

Московский Государственный Университет им. М.В. Ломоносова
Факультет вычислительной математики и кибернетики

В. Г. Баула

**Введение в архитектуру ЭВМ и
системы программирования**

Москва 2003

Предисловие

Данная книга представляет собой учебное пособие по архитектуре ЭВМ и системам программирования. Книга написана по читаемому автором лекционному курсу "Архитектура ЭВМ и язык Ассемблера" для студентов первого курса факультета Вычислительной математики и кибернетики МГУ им. М.В. Ломоносова. По данному курсу существует достаточно обширная литература, посвящённая программированию на Ассемблере, однако явно недостаточно учебной литературы собственно по **архитектуре** ЭВМ и системам программирования. Эта книга пособие призвано восполнить этот пробел.

В данном курсе вместе с архитектурой ЭВМ изучается и язык низкого уровня (Ассемблер). Это связано с тем обстоятельством, что ЭВМ является исполнителем алгоритма на машинном языке, поэтому знание языка низкого уровня необходимо для понимания архитектуры ЭВМ, на Ассемблере приводятся и все примеры, иллюстрирующие те ли иные особенности архитектуры. В то же время в изложении материала по архитектуре ЭВМ язык Ассемблера выполняет вспомогательную роль, основное внимание уделяется не изучению этого языка, а способам отображения конструкций языка высокого уровня (в основном Паскаля) на язык низкого уровня. Поэтому для достаточно полного изучения языка Ассемблер студентам совершенно необходимо знакомство с соответствующими курсами по этому языку (рекомендуется книга [5]).

Изложение материала по архитектуре ЭВМ проводится сначала по возможности в общем виде, безотносительно к конкретным компьютерам, а затем приводятся примеры, как та или иная особенность архитектуры реализована в современных ЭВМ. Так как в настоящее время в большинстве массовых ЭВМ используются процессоры фирмы Intel, то архитектура именно этого процессора (в основном его младшей модели) используются для изучения в первую очередь.

Большое внимание уделяется обоснованию тех или иных архитектурных решений, показывается необходимость появления определённых аппаратных возможностей, их нацеленность на решение встающих перед вычислительной техникой проблем. Показывается историческое развитие основных аппаратных решений, эволюция главных понятий от первых простых ЭВМ до современных компьютеров.

Изучение архитектуры ЭВМ неразрывно связано с выполнением машинных программ. Исходя из этого, в книге рассматриваются элементы системы программирования, показывается путь прохождения программы по всей цепочке от исходного текста, через объектный и загрузочный модули, до этапа счёта. Большое внимание приходится уделять модульному программированию, способам распределения памяти, статической и динамической загрузке и связыванию.

1. Понятие об архитектуре ЭВМ

Этот семестровый курс лекций называется "Архитектура ЭВМ и язык Ассемблера", хотя правильнее было бы назвать его всего лишь *введением* в архитектуру ЭВМ, так как в нашем курсе мы будем изучать только основы этого сложного предмета. Сначала нам нужно определить, что мы будем понимать под *архитектурой* компьютера. Обычно, определяя это понятие, говорят, что архитектура – это компоненты компьютера, их устройство, выполняемые ими функции, а также взаимосвязи между этими компонентами. Нас такое поверхностное определение не будет удовлетворять.

Дело в том, что понятие *архитектура* чего-либо существует не само по себе, а только в паре с другим понятием. Вы уже встречались с такой ситуацией в курсе из прошлого семестра "Алгоритмы и алгоритмические языки", где понятие *алгоритм* было неразрывно связано с понятием *исполнитель алгоритма*. При этом одна и та же запись для одного исполнителя была алгоритмом, а для другого – нет (например, если этот другой исполнитель не умел выполнять некоторые предписания в записи текста алгоритма).

Так и в нашем случае понятие архитектуры неразрывно связано с тем человеком (или теми людьми), которые изучают или рассматривают эту архитектуру. Ясно, что для разных людей архитектура одного и того же объекта может выглядеть совершенно по-разному. Так, например, обычный жилец многоэтажного дома видит, что этот дом состоит из фундамента, стен и крыши, имеет этажи, на каждом этаже есть квартиры, присутствует лестница, лифт и т.д. Совсем по-другому видит архитектуру этого же дома инженер, ответственный за его эксплуатацию. Он, например, знает, что некоторые перегородки между комнатами можно убрать при перепланировке квартиры, а другие перегородки являются несущими, если их убрать – дом рухнет. Инженер знает, где внутри стен проходят электрические провода, трубы водяного отопления, как обеспечивается противопожарная безопасность и многое другое.

Отсюда можно сделать вывод, что, изучая какой-либо объект, часто бывает удобно выделить различные **уровни** рассмотрения архитектуры этого объекта. Обычно выделяют три таких уровня: *внешний, концептуальный и внутренний*.¹ В качестве примера рассмотрим архитектуру какого-нибудь всем хорошо известного объекта, например, легкового автомобиля, на этих трёх уровнях.

1. **Внешний уровень.** На этом уровне видит архитектуру автомобиля обычный пассажир. Он знает, что машина имеет колёса, кузов, сиденья, мотор и другие части. Он понимает, что для работы автомобиля в него надо заливать бензин, знает назначение дворников на ветровом стекле, ремней безопасности и т.д. И этого ему вполне достаточно, чтобы успешно пользоваться машиной, главное – правильно назвать водителю нужный адрес.
2. **Концептуальный уровень.** Примерно на этом уровне видит архитектуру машины её водитель. В отличие от пассажира он знает, что в *его* автомобиль нужно заливать вовсе не бензин, а дизельное топливо, кроме того, необходимо ещё заливать масло и воду. Водитель знает назначение всех органов управления машиной, марку топлива, температуру окружающего воздуха, ниже которой необходимо заливать в машину не обычную воду, а так называемый антифриз и т.д. Ясно, что водитель видит архитектуру автомобиля совсем иначе, нежели обычный пассажир.
3. **Внутренний уровень.** На этом уровне автомобиль видит автомобиль инженер-конструктор, ответственный за его разработку. Он знает марку металла, из которого изготавливаются цилиндры двигателя, зависимость отдаваемой мотором мощности от вида топлива, допустимую нагрузку на отдельные узлы автомобиля, антикоррозийные свойства внешнего корпуса и многое другое.

Не надо думать, что один уровень видения архитектуры "хороший", а другой – "плохой". Каждый из них необходим и достаточен для конкретного применения рассматриваемого объекта. Знать объект на более глубоком уровне часто бывает даже вредно, так как получить эти знания обычно достаточно трудно, и все усилия пропадут, если в дальнейшем эти знания не понадобятся.

Необходимо, чтобы выпускники нашего университета, изучая какой-либо объект, достаточно ясно представляли себе, на каком уровне они его рассматривают и достаточен ли этот уровень для

¹ В области компьютеров и программного обеспечения такие уровни используются, например, при описании структуры баз данных, где описания данных (схемы данных) могут рассматриваться на внешнем, концептуальном и внутреннем уровнях [12].

практической работы с этим объектом. При необходимости, разумеется, надо перейти на более глубокий уровень рассмотрения.

Перейдём теперь ближе к предмету нашего курса – архитектуре компьютеров. Все люди, которые используют компьютеры в своей работе, обычно называются **пользователями**. Ясно, что в зависимости от того, на каком уровне они видят архитектуру компьютера, всех пользователей можно разделить на уровни или группы. Как правило, в научной литературе выделяют следующие группы пользователей.

1. **Конечные пользователи** (пользователи-непрограммисты). Обычно это специалисты в конкретных предметных областях – физики, биологи, лингвисты, финансовые работники и т.д., либо люди, использующие компьютеры в сфере образования, досуга и развлечений (они имеют дело с обучающими программами, компьютерными играми и т.д.). В своей работе все они используют компьютер, снабжённый соответствующим, как говорят, *прикладным программным обеспечением*. Это различные базы данных, текстовые редакторы, пакеты прикладных программ, системы автоматического перевода, обучающие, игровые и музыкальные программы. Этим пользователям достаточно видеть архитектуру компьютеров на *внешнем* уровне, их абсолютное большинство, примерно 90% от общего числа. Вообще говоря, компьютеры разрабатываются и выпускаются для нужд этих пользователей.
2. **Прикладные программисты**. Эти пользователи разрабатывают для конечных пользователей прикладное программное обеспечение. В своей работе они используют различные языки программирования высокого уровня (Паскаль, Фортран, Си, языки баз данных и т.д.) и соответствующие *системы программирования* (с этим понятием мы будем знакомиться в нашем курсе). Прикладным программистам достаточно видеть архитектуру компьютеров на *концептуальном* уровне. Можно примерно считать, что прикладных программистов примерно 8–9% от числа всех пользователей.
3. **Системные программисты**. Это самая небольшая (порядка 1%) группа пользователей, которая видит архитектуру ЭВМ на *внутреннем* уровне. Основная деятельность системных программистов заключается в разработке *системного программного обеспечения*. Курс лекций по системному программному обеспечению будет у Вас в следующем семестре, пока достаточно знать, что сюда относятся и системы программирования – тот инструмент, с помощью которого прикладные программисты пишут свои программы для конечных пользователей.

Разумеется, можно выделить и другие уровни видения архитектуры компьютера, не связанные с его *использованием*. В качестве примера можно указать уровень инженера-разработчика аппаратуры компьютера, уровень физика, исследующего новые материалы для построения схем ЭВМ и т.д. Эти уровни изучаются на других специальностях, и непосредственно нас интересовать не будут. В нашем курсе изучаются в основном первые три уровня, но иногда, в качестве примеров, мы совсем немного будем рассматривать и эти другие уровни видения архитектуры ЭВМ.

Далее укажем те **способы**, с помощью которых мы будем описывать архитектуру компьютера в нашем курсе. Можно выделить следующие основные способы описания архитектуры ЭВМ.

1. Словесные описания, использование чертежей, графиков, блок-схем и т.д. Именно таким способом в научной литературе описывается архитектура ЭВМ *для пользователей*.
2. В качестве другого способа описания архитектуры компьютера на *внутреннем* уровне можно с успехом использовать *язык машины* и близкий к нему *язык Ассемблера*. Дело в том, что компьютер является *исполнителем* алгоритма на языке машины и архитектуру компьютера легче понять, если знать язык, на котором записываются эти алгоритмы. В нашем курсе мы будем изучать язык Ассемблера в основном именно для лучшего понимания архитектуры ЭВМ. Для этого нам понадобится не полный язык Ассемблера, а лишь относительно небольшое подмножество этого языка.
3. Можно проводить описание архитектуры ЭВМ и с помощью *формальных языков*. Из курса предыдущего семестра Вы знаете, как важна *формализация* некоторого понятия, что позволяет значительно поднять строгость его описания и устранить различное понимание этого понятия разными людьми. В основном формальные языки используются для описания архитектуры ЭВМ на инженерном уровне, эти языки весьма сложны и их изучение выходит за рамки нашего предмета. Мы, однако, попробуем дать почти формальное описание архитектуры, но не "настоящего" компьютера, а некоторой *учебной* ЭВМ. Эта ЭВМ будет, с

одной стороны, достаточно проста, чтобы её формальное описание не было слишком сложным, а, с другой стороны, она должна быть *универсальной* (т.е. пригодной для реализации любых алгоритмов, для выполнения которых хватает аппаратных ресурсов компьютера).

Вы, конечно, уже знаете, что сейчас производятся самые разные компьютеры. Определим теперь, архитектуру **каких** именно ЭВМ мы будем изучать в нашем курсе.

1. Сначала мы рассмотрим архитектуру некоторой **абстрактной** машины (машины фон Неймана).
2. Далее мы изучим специальную **учебную ЭВМ**, которая по своей архитектуре близка к самым первым из выпускавшихся компьютеров.
3. Затем мы достаточно подробно изучим архитектуру первой модели того **конкретного компьютера**, на котором Вы работаете в терминальном классе.
4. В заключение нашего курса мы проведём некоторый достаточно простой **сравнительный анализ** архитектуры основных классов универсальных ЭВМ.

2. Машина Фон Неймана

В 1946 Джон фон Нейман (с соавторами) описал архитектуру некоторого абстрактного вычислителя, который сейчас принято называть *машиной фон Неймана* [2]. Эта машина является *абстрактной моделью* ЭВМ, однако, эта абстракция отличается от абстрактных исполнителей алгоритмов (например, машины Тьюринга). Если машину Тьюринга принципиально нельзя реализовать из-за входящей в её архитектуру бесконечной ленты, то машина фон Неймана не поддаётся реализации, так как многие детали в архитектуре этой машины *не конкретизированы*. Это было сделано специально, чтобы не сковывать творческого подхода к делу у инженеров-разработчиков новых ЭВМ.

В некотором смысле машина фон Неймана подобна *абстрактным структурам данных*, которые Вы изучали в предыдущем семестре. Для таких структур данных, как Вы помните, для их использования необходимо было произвести отображение на структуры данных хранения и реализовать соответствующие операции над этими данными.

Можно сказать, что в машине фон Неймана зафиксированы те особенности архитектуры, которые в той или иной степени должны быть присущи, по мнению авторов этой абстрактной машины, всем компьютерам. Разумеется, практически все современные ЭВМ по своей архитектуре отличаются от машины фон Неймана, однако эти отличия удобно изучать именно как *отличия*, проводя сравнения и сопоставления с машиной фон Неймана. При нашем рассмотрении данной машины будет обращено внимание на отличия архитектуры машины фон Неймана от современных ЭВМ. основополагающие свойства архитектуры машины фон Неймана будут сформулированы в виде **принципов фон Неймана**. Эти принципы многие годы определяли основные черты архитектуры ЭВМ нескольких поколений [3].

На рис. 2.1 приведена схема машины фон Неймана, как она изображается в большинстве учебников, посвящённых архитектуре ЭВМ. На этом рисунке толстыми стрелками показаны *потоки команд и данных*, а тонкими – передача между устройствами *управляющих сигналов*. Машина фон Неймана состоит из памяти, устройств ввода/вывода и *центрального процессора* (ЦП). Центральный процессор, в свою очередь, состоит из *устройства управления* (УУ) и *арифметико-логического устройства* (АЛУ). Рассмотрим последовательно устройства машины фон Неймана и выполняемые ими функции.

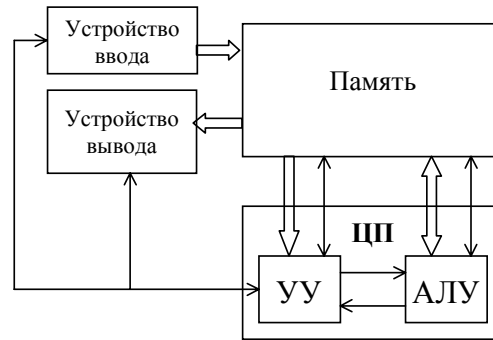


Рис. 2.1. Схема машины фон Неймана.

2.1. Память

Принцип линейности и однородности памяти.

Память – линейная (упорядоченная) однородная последовательность некоторых элементов, называемых *ячейками*. В любую ячейку памяти другие устройства машины (по толстым стрелкам) могут записать и считать информацию, причём время чтения из любой ячейки одинаково для всех ячеек. Время записи в любую ячейку тоже одинаково (это и есть принцип *однородности* памяти).¹ Такая память в современных компьютерах называется *памятью с произвольным доступом* (Random Access Memory, RAM). На практике многие ЭВМ могут иметь участки памяти разных видов, одни из которых поддерживают только чтение информации (Read Only Memory, ROM), другие могут допускать запись, но за большее время, чем в остальную память (это так называемая *полупостоянная* память) и др.

Ячейки памяти в машине фон Неймана нумеруются от нуля до некоторого положительного числа N , которое обычно является степенью двойки. *Адресом ячейки* называется её номер. Каждая ячейка состоит из более мелких частей, именуемых *разрядами* и нумеруемых также от нуля и до определённого числа. Количество разрядов в ячейке обозначает *разрядность памяти*. Каждый разряд может хранить цифру в некоторой системе счисления. В большинстве ЭВМ используется двоичная система счисления, т.к. это более выгодно с точки зрения аппаратной реализации, в этом случае каждый разряд хранит один *бит* информации. Восемь бит составляет один *байт*.

Содержимое ячейки называется *машинным словом*. С точки зрения архитектуры, машинное слово – это минимальный объём данных, которым могут обмениваться различные узлы машины (не надо, однако, забывать о передаче управляющих сигналов по тонким стрелкам). Из каждой ячейки памяти можно считать *копию* машинного слова и передать её в другую часть памяти, при этом оригинал не меняется. При записи в память старое содержимое ячейки пропадает и заменяется новым машинным словом.

Заметим, что на практике решение задачи сохранения исходного машинного слова при чтении из ячейки для некоторых видов памяти является нетривиальным и достаточно трудоёмким, так как в этой памяти (она называется *динамической* памятью) при чтении оригинал разрушается. Приведём типичные *характеристики памяти* современных ЭВМ.

1. Объём памяти – сотни миллионов ячеек (обычно восьмиразрядных).
2. Скорость работы памяти: *время доступа* (минимальная задержка на чтение слова) и *время цикла* (минимальная задержка на чтение из одной и той же ячейки двух слов) – порядка единиц и десятков наносекунд ($1 \text{ секунда} = 10^9 \text{ наносекунд}$). Заметим, что для упомянутой выше динамической памяти время цикла *больше*, чем время доступа, так как надо ещё восстановить разрушенное при чтении содержимое ячейки.
3. Стоимость. Для основной памяти ЭВМ пока достаточно знать, что чем быстрее такая память, тем она, естественно, дороже. Конкретные значения стоимости памяти не представляют интереса в рамках наших лекций.

Принцип неразличимости команд и данных. Машинное слово представляет собой либо команду, либо подлежащее обработке данное (число, символьная информация, элемент изображения

¹ Разумеется, время чтения из ячейки памяти может не совпадать с временем записи в неё.

и т.д.). Для краткости в дальнейшем будем называть такую информацию "числами". Данный принцип фон Неймана заключается в том, что числа и команды *неотличимы* друг от друга – в памяти и те и другое представляются некоторым набором разрядов, причём по внешнему виду машинного слова нельзя определить, что оно представляет – команду или число.

Из этого принципа вытекает очевидное следствие – **принцип хранимой программы**. Этот принцип является очень важным, его суть состоит в том, что программа хранится в памяти вместе с числами, а значит, может изменяться во время счёта этой программы. Говорят также, что программа может *самомодифицироваться* во время счёта. Заметим, что, когда фон Нейман писал свою работу, большинство тогдашних ЭВМ хранили программу в памяти одного вида, а числа – в памяти другого вида. В современных ЭВМ и программы, и данные хранятся в одной и той же памяти.

2.2. Устройство Управления

Как ясно из самого названия, устройство управления (УУ) *управляет* всеми остальными устройствами ЭВМ. Оно осуществляет это путём посылки *управляющих сигналов*, подчиняясь которым остальные устройства производят определённые действия, предписанные этими сигналами. Это устройство является единственным, от которого на рис. 2.1 отходят тонкие стрелки ко всем другим устройствам. Остальные устройства могут "командовать" только памятью, делая ей запросы на чтение и запись машинных слов.

Принцип автоматической работы. Машина, выполняя записанную в её памяти *программу*, функционирует автоматически, без участия человека.¹ *Программа* – набор записанных в памяти (не обязательно последовательно) машинных команд, описывающих шаги работы алгоритма. Таким образом, программа – это запись алгоритма на *языке машины*. *Язык машины* – набор всех возможных команд.

Принцип последовательного выполнения команд. Устройство управления выполняет некоторую команду от начала до конца, а затем по определённому правилу выбирает следующую команду для выполнения, затем следующую и т.д. Этот процесс продолжается, пока не будет выполнена специальная команда останова, либо при выполнении очередной команды не возникнет *аварийная ситуация* (например, деление на ноль). Аварийная ситуация – это аналог *безрезультативного* останова алгоритма.

2.3. Арифметико–Логическое Устройство

В архитектуре машины фон Неймана арифметико-логическое устройство (АЛУ) может выполнить следующие действия.

1. Считать содержимое некоторой ячейки памяти – поместить копию машинного слова из этой ячейки в ячейку, расположенную в самом АЛУ. Такие ячейки, расположенные не в памяти, а в других устройствах ЭВМ, называются *регистровой памятью* или просто *регистрами*.
2. Записать в некоторую ячейку памяти – поместить копию содержимого регистра АЛУ в ячейку памяти. Когда не имеет значения, какая операция (чтение или запись) производится, говорят, что происходит *обмен* машинным словом между регистром и памятью.
3. АЛУ может также выполнять различные *операции* над данными в своих регистрах, например, сложить содержимое двух регистров (обычно называемых регистрами первого R1 и второго R2 операндов), и поместить результат на третий регистр (называемый, как правило, сумматором S).

2.4. Взаимодействие УУ и АЛУ

Революционность идей фон Неймана заключалась в *специализации*: каждое устройство отвечает за выполнение только своих функций. Если раньше, например, память часто не только хранила данные, но и могла производить операции над ними, то теперь было предложено, чтобы память только хранила данные, АЛУ производило арифметико-логические операции над ними, устройство ввода только вводило данные из "внешнего мира" в память и т.д. Фон Нейман распределил функции между различными устройствами, что существенно упростило схему машины.

¹ Если только такое участие не предусмотрено в самой программе, например, при вводе данных с клавиатуры. Пример устройства, которое может выполнять команды, но не в автоматическом режиме – обычный (непрограммируемый) калькулятор.

Устройство управления тоже имеет свои регистры, оно может считывать команды из памяти на специальный *регистр команд* (RK), на котором всегда хранится *текущая* выполняемая команда. Регистр УУ с именем RA называется *счётчиком адреса*, при выполнении текущей команды в него записывается адрес *следующей* команды (первую букву в сокращении слова регистр будем записывать латинской буквой R).

Рассмотрим, например, операцию сложения двух чисел $z := x + y$ (здесь x , y и z – адреса ячеек памяти, в которых хранятся, соответственно, операнды и результат сложения). При получении такой команды УУ последовательно посылает управляющие сигналы в АЛУ, предписывая ему сначала считать операнды x и y из памяти и поместить их на регистры R1 и R2. Затем по следующему управляющему сигналу АЛУ производит операцию сложения чисел на регистрах R1 и R2 и записывает результат на регистр S. По следующему управляющему сигналу АЛУ пересылает копию регистра S в ячейку памяти с адресом z . Ниже приведена иллюстрация описанного примера на языке Паскаль, где R1, R2 и S – регистры АЛУ, ПАМ – массив, условно обозначающий память ЭВМ, а \otimes – операция (в нашем случае это сложение, т.е. $\otimes = +$).

```
R1 := ПАМ[x]; R2 := ПАМ[y]; S := R1  $\otimes$  R2; ПАМ[z] := S;
```

В дальнейшем конструкция ПАМ[A] для краткости будет обозначаться как <A>, тогда наш пример переписывается так:

```
R1 := <x>; R2 := <y>; S := R1  $\otimes$  R2; <z> := S;
```

Опишем теперь более формально шаги выполнения одной команды в машине фон Неймана:

```
RK := <RA>; считать из памяти очередную команду на регистр команд;
```

```
RA := RA + 1; увеличить счётчик адреса на единицу;
```

```
Выполнить очередную команду.
```

Затем выполняется следующая команда и т.д. Итак, если машинное слово попадает на регистр команд, то оно *интерпретируется* УУ как команда, а если слово попадает в АЛУ, то оно *по определению* считается числом. Это позволяет, например, складывать команды программы как числа, либо выполнить некоторое число как команду. Разумеется, обычно такая ситуация является семантической ошибкой, если только специально не предусмотрена программистом для каких-то целей (мы иногда будем оперировать с командами, как с числами, в нашей учебной машине).

Современные ЭВМ в той или иной степени нарушают **все** принципы фон Неймана. Например, существуют компьютеры, которые различают команды и данные. В них каждая ячейка памяти кроме собственно машинного слова содержит ещё специальный признак, называемый *тэгом*, который и определяет, чем является машинное слово. В этой архитектуре при попытке выполнить число как команду, либо складывать команды как числа, будет зафиксирована ошибка. Так нарушается принцип неразличимости команд и чисел.

Практически все современные ЭВМ нарушают принцип однородности и линейности памяти. Память может быть, например, двумерной, когда адрес ячейки задаётся не одним, а двумя числами, либо ячейки памяти могут вообще не иметь адресов (такая память называется *ассоциативной*) и т.д.

Достаточно мощные компьютеры нарушают и принцип последовательного выполнения команд: они одновременно могут выполнять несколько команд как из одной программы, так, иногда, и из разных программ (такие компьютеры могут иметь несколько центральных процессоров, а также быть так называемыми *конвейерными* ЭВМ, их мы рассмотрим в конце нашего курса).

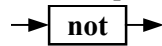
Особо следует отметить, что в архитектуре машины фон Неймана реализованы и другие принципы, которые самим фон Нейманом явно не формулировались, так как считались самоочевидными. Так, например, предполагается, что во время выполнения программы не меняется число узлов компьютера и взаимосвязи между ними. В то же время сейчас существуют ЭВМ, которые нарушают и этот принцип. Во время работы одни устройства могут, как говорят, отбраковываться (например, отключаться для ремонта), другие – автоматически подключаться, появляются новые связи между элементами ЭВМ (например, в так называемых *трансьютерах*) и т.д.

На этом мы закончим краткое описание машины фон Неймана и принципов её работы. И в заключение этого раздела мы совсем немного рассмотрим архитектуру ЭВМ на уровне инженера-конструктора. Это будет сделано исключительно для того, чтобы снять тот покров таинственности с работы центрального процессора, который есть сейчас у некоторых студентов: как же машины может выполнять различные операции, неужели она такая умная?

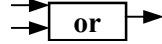
Аппаратура современных ЭВМ состоит из некоторых элементарных конструктивных элементов, называемых *вентильями*. Каждый вентиль реализует одну из логических операций, у него есть один или два *входа* и один *выход*. На входах и выходе могут быть электрические сигналы двух видов:

низкое напряжения (трактуются как ноль или логическое значение **false**) и высокое (ему соответствует единица или логическое значение **true**). Основные вентили следующие.

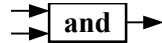
1. *Отрицание*, этот вентиль имеет один вход и один выход, если на входе значение **true**, то на выходе значение **false** и наоборот. Будем изображать этот вентиль так:



2. *Дизъюнкция* или логическое сложение, реализует хорошо известную Вам операцию Паскаля **or**, будем изображать его как



3. **И**, наконец, *конъюнкция* или логическое умножение, изображаемое как



Каждый вентиль срабатывает (т.е. преобразует входные сигналы в выходные) не непрерывно, а только тогда, когда на вентиль по специальному управляющему проводу приходит так называемый *такты́ый импульс*. Заметим, что по этому принципу работают ЭВМ, которые называются *дискретными*, в отличие от *аналоговых* компьютеров, схемы в которых работают непрерывно. Подавляющее число современных ЭВМ являются дискретными, только их мы и будем изучать. Более подробно об этом можно прочесть в книгах [1,3].

Из вентилях строятся так называемые *интегральные схемы* – это набор вентилях, соединённых проводами и такими радиотехническими элементами, как сопротивления, конденсаторы и индуктивности. Каждая интегральная схема тоже имеет свои входы и выходы и реализует какую-нибудь функцию узла компьютера. В специальной литературе интегральные схемы, которые содержат порядка 1000 вентилях, называются малыми интегральными схемами (МИС), порядка 10000 вентилях – средними (СИС), порядка 100000 – большими (БИС) и более 100000 вентилях – сверхбольшими интегральными схемами (СБИС).

Большинство современных интегральных схем собираются на одной небольшой прямоугольной пластинке полупроводника с размерами порядка сантиметра. Под микроскопом такая пластинка СБИС похожа на план большого города. Интегральная схема имеет от нескольких десятков до нескольких сотен внешних контактов.

Для того, чтобы реализовать простые электронные часы, необходимо порядка 1000 вентилях, из 10000 вентилях уже можно собрать простейший центральный процессор, а современные мощные ЭВМ состоят из миллионов вентилях.

В качестве примера рассмотрим интегральную схему, которая реализует функцию сложение двух одноразрядных двоичных чисел. Входными данными этой схемы являются значения переменных x и y , а результатом – их сумма, которая, в общем случае, является двухразрядным числом (обозначим разряды этого числа как a и b), формирующиеся как результат сложения $x+y$. Запишем таблицу истинности для этой функции от двух переменных:

x	y	b	a
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Легко вычислить, что величины a и b будут определяться формулами:

$$a = x \oplus y = (x \text{ or } y) \text{ and not } (x \text{ and } y)$$

$$b = x \text{ and } y$$

Реализуем нашу интегральную схему как набор вентилях, связанных проводниками (рис. 2.2. а).

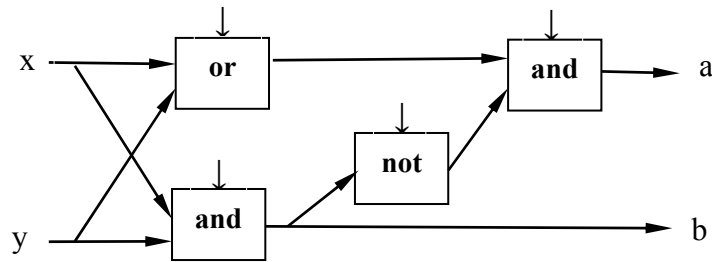


Рис. 2.2. а) Сборка двоичного сумматора из вентилях, ↓ – тактовые импульсы.

Наша интегральная схема (см. рис. 2.2 б) будет иметь не менее 7-ми внешних контактов: входные x и y , выходные a и b , один контакт для подачи тактовых импульсов, два контакта для подачи электрического питания (ясно, что без энергии ничего работать не будет) и, возможно, другие контакты. Суммирование чисел x и y в приведенной выше схеме осуществляется после прихода трёх тактовых импульсов (как говорят, за три такта). Современные компьютеры обычно реализуют более сложные схемы суммирования, срабатывающие за один такт.

Скорость работы интегральной схемы зависит от частоты прихода тактовых импульсов, называемой *такты́й частотой*. У современных ЭВМ тактовые импульсы приходят на схемы основной *памяти* с частотой примерно в сто миллионов раз в секунду, а на схемы *центрального процессора* – ещё примерно в 10 раз чаще.



Рис. 2.2б. Интегральная схема двоичного сумматора.

3. Учебная машина

Рассмотрим конкретизацию абстрактной машины фон Неймана на примере учебной машины, которую будем называть УМ-3 (смысл этого названия – учебная машина трёхадресная). Наша учебная машина будет удовлетворять *всем* принципам фон Неймана.

Память учебной машины состоит из 512 ячеек, имеющих адреса от 0 до 511, по 32 двоичных разряда каждая. В каждой ячейке может быть записано целое или вещественное *число* (представляются они по-разному) или *команда*. Команда в ячейке будет представляться в следующей форме:

КОП	A1	A2	A3
5 разрядов	9 разрядов	9 разрядов	9 разрядов

Здесь КОП – это число от 0 до 31, которое задаёт номер (код) операции, а A1, A2 и A3 – адреса операндов. Таким образом, в каждой команде задаются адреса аргументов (это A2 и A3) и адрес результата операции A1. Конкретизируем регистры устройства управления:

- RA – регистр, называемый *счётчиком адреса*, он имеет 9 разрядов и хранит адрес команды, которая будет выполняться вслед за текущей командой;
- RK – регистр команд имеет 32 разряда и содержит текущую выполняемую команду (код операции КОП и адреса операндов A1, A2 и A3);
- w – регистр «омега», в который после выполнения некоторых команд (у нас это будут арифметические команды сложения, вычитания, умножения и деления) записывается число от 0 до 2 по правилу (s – результат арифметической операции):

$$w := \begin{cases} 0, & s = 0, \\ 1, & s < 0, \\ 2, & s > 0; \end{cases}$$


- Err – регистр ошибки, содержащий нуль в случае успешного выполнения очередной команды и единицу в противном случае.

В таблице 3.1 приведены все команды учебной машины УМ–3.

3.1. Схема выполнения команд

Все *бинарные* операции (т.е. те, которые имеют два аргумента и один результат) выполняются в нашей учебной машине по схеме: $\langle A1 \rangle := \langle A2 \rangle \otimes \langle A3 \rangle$ (\otimes – любая бинарная операция). Каждая команда выполняется по следующему алгоритму:

1. RK := <RA>; чтение очередной команды на регистр команд УУ.
2. RA := RA + 1.
3. Выполнение операции, заданной в коде операции (КОП). При ошибочном КОП выполняется Err := 1.
4. **if** (Err=0) **and** (КОП<>СТОП) **then goto** 1 **else** КОНЕЦ.

Теперь нам осталось определить условие начала работы программы. Для загрузки программы в память и формирования начальных значений регистров в устройстве управления на устройстве ввода имеется специальная кнопка ПУСК ( Пуск ☺). При нажатии этой кнопки **устройство ввода** самостоятельно (без сигналов со стороны устройства управления) производит следующую последовательность действий:

1. Производится ввод расположенного на устройстве ввода массива машинных слов в память, начиная с *первой ячейки*; этот массив машинных слов заканчивается специальным признаком *конца массива*.
2. RA := 1
3. w := 0
4. Err := 0

Далее всё готово для автоматической работы центрального процессора по загруженной в память программе. Таким образом, мы полностью определили условия начала и конца работы нашей алгоритмической системы (вспомним курс "Алгоритмы и алгоритмические языки").

Таблица 3.1. Команды учебной машины.

КОП	Операция и её мнемоническое обозначение
01	СЛВ – сложение вещественных чисел
11	СЛЦ – сложение целых чисел
02	ВЧВ – вычитание вещественных чисел
12	ВЧЦ – вычитание целых чисел
03	УМВ – умножение вещественных чисел
13	УМЦ – умножение целых чисел
04	ДЕВ – деление вещественных чисел
14	ДЕЦ – деление целых чисел (то же, что и div в Паскале)
24	МОД – остаток от деления (то же, что и mod в Паскале)
00	ПЕР – пересылка: $\langle A1 \rangle := \langle A3 \rangle$
10	ЦЕЛ – вещественное в целое: $\langle A1 \rangle := \text{Round}(\langle A3 \rangle)$
20	ВЕЩ – целое в вещественное: $\langle A1 \rangle := \text{Real}(\langle A3 \rangle)$
09	БЕЗ – безусловный переход: goto A2, т.е. RA:=A2
19	УСЛ – условный переход: Case w of 0: goto A1; 1: goto A2; 2: goto A3 end
31	СТОП – остановка выполнения программы
05	ВВВ – ввод A2 вещественных чисел в память, начиная с адреса A1

15	ВЫВ – вывод вещественных чисел, аналогично ВВВ
06	ВВЦ – ввод целых чисел, аналогично ВВВ
16	ВЫЦ – вывод целых чисел, аналогично ВВВ

По своей архитектуре наша учебная машина очень похожа на первые ЭВМ, построенные в соответствии с принципами фон Неймана, например, на отечественную ЭВМ СТРЕЛА [3], выпускавшуюся в середине прошлого века.

3.2. Примеры программ для учебной машины.

3.2.1. Пример 1. Оператор присваивания.

Составим программу, которая реализует арифметический оператор присваивания.

$$y := (x+1)^2 \text{ mod } (x-1)^2.$$

Сначала необходимо решить, в каких ячейках памяти будут располагаться наши переменные x и y . Эта работа называется *распределением памяти* под хранение переменных. При программировании на Паскале эту работу выполняла за нас Паскаль-машина, когда видела описания переменных:

```
Var x, y: integer;
```

Теперь нам придётся распределять память самим. Сделаем естественное предположение, что наша программа будет занимать не более 100 ячеек памяти (напомним, что программа вводится, начиная с первой ячейки памяти при нажатии кнопки ПУСК). Тогда, начиная со 101 ячейки, память будет свободна. Пусть для хранения значения переменной x мы выделим 101 ячейку, а переменной y – 102 ячейку. Остальные переменные при необходимости будем размещать в последующих ячейках памяти. В приведенном примере нам понадобятся дополнительные (как говорят, *рабочие*) переменные $r1$ и $r2$, которые мы разместим в ячейках 103 и 104 соответственно.

При программировании на машинном языке, в отличие от Паскаля, у нас не будут существовать константы. Действительно, в какой бы ячейке мы не поместили значение константы, ничто не мешает нам записать в эту ячейку новое значение. Поэтому мы будем называть *константами* такие переменные, которые имеют начальные значения, и которые мы не планируем изменять в ходе выполнения программы. Отсюда следует, что такие константы, как и переменные с начальным значением, следует располагать в *тексте программы* и загружать в память вместе с программой при нажатии кнопки ПУСК. Разумеется, такие переменные с начальными значениями следует располагать в таком месте программы, чтобы устройство управления не начало бы выполнять их как команды. Чтобы избежать этого мы будем размещать их в конце программы, после команды СТОП.

Следует обратить внимание и на тот факт, что изначально программист не знает, сколько ячеек в памяти будет занимать его программа. Поэтому адреса программы, ссылающиеся на переменные с начальным значением, до завершения написания программы остаются незаполненными, и уже потом, разместив эти переменные в памяти **сразу же вслед за командами программы**, следует указать их адреса в тексте программы. В приведённом примере те адреса программы, которые заполняются в последнюю очередь, будут обозначаться подчёркиванием.

Запись программы состоит из строк, каждая строка снабжается номером ячейки, куда будет помещаться это машинное слово (команда или переменная с начальным значением) при загрузке программы. Вслед за номером задаются все поля команды, затем программист может указать комментарий. Номера ячеек, кодов операций и адреса операндов будем записывать в десятичном виде, хотя первые программисты использовали для этого 8-ую или 16-ую системы счисления. Кроме того, так как числа неотличимы по внешнему виду от команд, то будем записывать их тоже чаще всего в виде команд. Текст нашей первой программы с комментариями приведён на рис. 3.1.

№	Команда				Комментарий
001	06	101	001	000	Ввод x
2	11	103	101	<u>009</u>	$r1 := (x+1)$
3	13	103	103	103	$r1 := (x+1)^2$
4	12	104	101	<u>009</u>	$r2 := (x-1)$
5	13	104	104	104	$r2 := (x-1)^2$
6	24	102	103	104	$y := r1 \text{ mod } r2$

7	16	102	001	000	Вывод y
8	31	000	000	000	Стоп
9	00	000	000	001	Целая константа 1

Рис 3.1. Текст программы первого примера.

После написания программы осталось поместить на устройство ввода два массива – саму программу (9 машинных слов) и число x (одно машинное слово) и нажать кнопку ПУСК. Как мы уже говорили, первый массив заканчивался специальной строкой – признаком конца ввода, так что устройство ввода знает, сколько машинных слов надо ввести в память по кнопке ПУСК.

3.2.2. Пример 2. Условный оператор.

Составим теперь программу, реализующую условный оператор присваивания. Пусть целочисленная переменная y принимает значение в зависимости от вводимой целочисленной переменной x в соответствии с правилом:

$$y := \begin{cases} x+2, & \text{при } x < 2, \\ 2, & \text{при } x = 2, \\ 2^*(x+2), & \text{при } x > 2; \end{cases}$$

В данном примере при записи программы на месте кода операции мы будем для удобства вместо числа указывать его мнемоническое обозначение. Разумеется, потом, перед вводом программы необходимо будет заменить эти мнемонические обозначения соответствующими им числами.

Для определения того, является ли значение переменной x больше, меньше или равным константе 2, мы будем выполнять операцию вычитания $x-2$, получая в регистре w значение 0 при $x=2$, 1 при $x<2$ и 2 при $x>2$. При этом сам результат операции вычитания нам не нужен, но по нашему формату команд указание адреса ячейки для записи результата является обязательным. Для записи таких ненужных значений мы будем чаще всего использовать ячейку с номером 0. В соответствии с принципом однородности памяти, эта ячейка ничем не отличается от других, то есть, доступна как для записи, так и для чтения данных. В некоторых реальных ЭВМ этот принцип нарушается: при считывании из этой ячейки всегда возвращался нуль, а запись в ячейку с адресом ноль физически не осуществляется (на практике такой принцип работы с этой ячейкой иногда удобнее).

Для хранения переменных x и y выделим ячейки 100 и 101 соответственно. Программист сам определяет порядок размещения в программе трёх ветвей нашего условного оператора присваивания. Мы будем сначала располагать вторую ветвь ($x=2$), затем первую ($x<2$), а потом третью ($x>2$). На рис. 3.2 приведён текст этой программы.

№	Команда	Комментарий
001	ВВЦ 100 001 000	Read(x)
2	СЛЦ 101 100 <u>011</u>	$y := x+2$
3	ВЧЦ 000 100 <u>011</u>	$\langle 000 \rangle := x-2$; формирование w
4	УСЛ 005 <u>007</u> 009	Case w of 0: goto 005; 1: goto 007; 2: goto 009 end
5	ВЫЦ <u>011</u> 001 000	Write(2)
6	СТОП 000 000 000	Конец работы
7	ВЫЦ 101 001 000	Write(y)
8	СТОП 000 000 000	Конец работы
9	УМЦ 101 <u>011</u> 101	$y := 2 * y$
010	БЕЗ 000 007 000	Goto 007
1	00 000 000 002	Целая константа 2

Рис 3.2. Текст программы второго примера.

Обратите внимание, что константа 2 неотличима от команды пересылки содержимого второй ячейки памяти в нулевую ячейку, именно такая команда и будет выполняться, если эта константа будет выбрана на регистр команд устройства управления.

3.2.3. Пример 3. Реализация цикла.

В качестве следующего примера напишем программу для вычисления начального отрезка гармонического ряда:

$$y = \sum_{i=1}^n 1/i$$

Для хранения переменных n , y и i выделим ячейки 100, 101 и 102 соответственно. В этом алгоритме мы реализуем *цикл с предусловием*, поэтому при вводе $n < 1$ тело цикла не будет выполняться ни одного раза, и наша программа будет выдавать нулевой результат. На рис. 3.3 приведена возможная программа для решения этой задачи.

Сделаем некоторые замечания к этой программе. В нашем языке у нас нет команды деления целого числа на вещественное, поэтому при вычислении величины $1.0/i$ нам пришлось отдельной командой

```
ВЕЩ 000 000 102
```

преобразовать значение целой переменной i в вещественное значение. Обратите также внимание, что для нашей учебной машины мы ещё не определили формат представления вещественных чисел, поэтому в ячейке с адресом 14 стоит пока просто условное обозначение константы 1.0, а не её машинное представление.

№	Команда	Комментарий
001	ВВЦ 100 001 000	Read(n)
2	ВЧВ 101 101 101	$y := 0.0$
3	ПЕР 102 000 013	$i := 1$
4	ВЦЦ 000 102 100	$i := i - n$; формирование w
5	УСЛ 006 006 011	If $i > n$ then goto 011
6	ВЕЩ 000 000 102	$\langle 000 \rangle := \text{Real}(i)$
7	ДЕВ 000 014 000	$\langle 000 \rangle := 1.0 / \langle 000 \rangle$
8	СЛВ 101 101 000	$y := y + \langle 000 \rangle$
9	СЛЦ 102 102 013	$i := i + 1$
010	БЕЗ 000 004 000	Следующая итерация цикла
1	ВЫВ 101 001 000	Write(y)
2	СТОП 000 000 000	Стоп
3	00 000 000 001	Целая константа 1
4	$\langle 1.0 \rangle$	Вещественная константа 1.0

Рис 3.3. Текст программы третьего примера.

3.2.4. Пример 4. Работа с массивами.

Пусть требуется написать программу для ввода массива x из 100 вещественных чисел и вычисления суммы всех элементов этого массива:

$$S = \sum_{i=1}^{100} x[i]$$

Будем предполагать, что длина программы не превышает 200 ячеек, и поместим массив x , начиная с 200-ой ячейки памяти. Вещественную переменную S с начальным значением 0.0 и целую переменную i с начальным значением 100 разместим в конце текста программы. На рис. 3.4 приведён текст этой программы.

№	Команда	Комментарий
001	ВВВ 200 100 000	Read(x); массив x в ячейках 200+299
2	СЛВ 008 200 008	$S := S + x[1]$

3	СЛЦ	002	002	011	Модификация команды в ячейке 2
4	ВЧЦ	010	010	009	n := n-1
5	УСЛ	006	006	002	Следующая итерация цикла
6	ВЫВ	008	001	000	Write(S)
7	СТОП	000	000	000	Стоп
8		<0.0>			Переменная S = 0.0
9	00	000	000	001	Целая константа 1
010	00	000	000	100	Переменная n с начальным значением 100
1	00	000	001	000	Константа переадресации

Рис 3.4. Текст программы четвертого примера.

Рассматриваемая программа выделяется своим новым приёмом программирования и может быть названа *самомодифицирующейся программой*. Обратим внимание на третью строку программы. Содержащаяся в ней команда изменяет исходный код программы (команду в ячейке 2) для организации цикла перебора элементов массива. Модифицируемая команда рассматривается как целое число, которое складывается со специально подобранное *константой переадресации*. Согласно одному из принципов фон Неймана, числа и команды в учебной машине неотличимы друг от друга, а, значит, изменяя числовое представление команды, мы можем изменять и её суть.

У такого метода программирования есть один существенный недостаток: модификация кода программы внутри её самой может привести к путанице и вызвать появление ошибок. Кроме того, самомодифицирующуюся программу трудно понимать и вносить в неё изменения. В нашей учебной машине это, однако, *единственный* способ обработки массивов. В других архитектурах ЭВМ, с которыми мы познакомимся несколько позже, есть и другие, более эффективные способы работы с массивами, поэтому метод с модификацией команд не используется.

3.3. Формальное описание учебной машины

При описании архитектуры учебной ЭВМ на естественном языке многие вопросы остались нераскрытыми. Что, например, будет после выполнения команды из ячейки с адресом 511? Какое значение после нажатия кнопки ПУСК имеют ячейки, расположенные вне введённой программы? Как представляются целые и вещественные числа? Для ответа на почти все такие вопросы мы приведём формальное описание нашей учебной машины. В качестве метаязыка мы будем использовать Турбо-Паскаль, на котором Вы работаете. Другими словами, мы напишем программу, выполнение которой *моделирует* работу нашей учебной машины, т.е. наша машина, по определению, работает "почти так же", как и написанная нами программа на Паскале.

Ниже приведена реализация учебной машины на языке Турбо-Паскаль:

```

program VM_3(input, output);
const
  N = 511;
type
  Address = 0..N;
  Tag = (kom, int, fl); {В машинном слове может храниться команда, целое
                        или вещественное число}
  Komanda = packed record
    KOP: 0..31;
    A1, A2, A3: Address;
  end;
  Slovo = packed record
    case Tag of
      kom: (k: Komanda);
      int: (i: LongInt);
      fl: (f: Single);
    end
  Memory = array[0..N] of Slovo;
var

```

```

Mem: Memory;
S, R1, R2: Slovo; {Регистры АЛУ}
RK: Komanda; {Регистр команд}
RA: Address; {Счётчик адреса}
Om: 0..2; {Регистр w}
Err: Boolean;
begin
Input_Program; {Эта процедура должна вводить текст программы с устройства
                ввода в память по кнопке ПУСК}
Om := 0; Err := False; RA := 1; {Начальная установка регистров}
with RK do
repeat {Основной цикл выполнения команд}
RK := Mem[RA].k;
RA := (RA+1) mod (N+1);
case KOP of {Анализ кода операции}
00: { ПЕР }
begin R1 := Mem[A3]; Mem[A1] := R1 end;
01: { СЛБ }
begin
R1 := Mem[A2]; R2 := Mem[A3]; S.f := R1.f + R2.f;
if S.f = 0.0 then OM := 0 else
if S.f < 0.0 then OM := 1 else OM := 2;
Mem[A1] := S; { Err := ? }
end;
09: { БЕЗ }
RA := A2;
24: { МОД }
begin
R1 := Mem[A2]; R2 := Mem[A3];
if R2.i = 0 then Err := True else begin
S.i := R1.i mod R2.i; Mem[A1] := S;
if S.i = 0 then OM := 0 else
if S.i < 0 then OM := 1 else OM := 2;
end
end;
13: { СТОП } ;
{ Реализация остальных кодов операций }
else
Err := True;
end; { case }
until Err or (KOP = 31)
end.

```

Для хранения машинных слов мы описали тип *Slovo*, который является записью с вариантами языка Турбо-Паскаль. В такой записи на одном и том же месте памяти могут располагаться команды, длинные (32-битные) целые числа или же 32-битные вещественные числа типа *Single*.¹

Наша программа ведёт себя почти так же, как учебная машина. Одно из немногих мест, где это поведение расходится, показано в тексте программы, например, при реализации команды сложения вещественных чисел. Программа на Паскале при переполнении (когда результат сложения не помещается в переменную *S*) производит аварийное завершение программы, а учебная машина просто присваивает регистру *Err* значение 1. Наше формальное описание отвечает и на вопрос о том, как в учебной машине представляются целые и вещественные числа: точно так же, как в переменных на Паскале. Это представление мы изучим в нашем курсе несколько позже.

Заметим также, что память учебной машины как бы замкнута в кольцо: после выполнения команды из ячейки с адресом 511 (если это не команда перехода) следующая команда будет выполняться из ячейки с адресом ноль. Такая организация памяти типична для многих современных ЭВМ.

¹ Тип *real* Турбо-Паскаля не подходит, потому что имеет длину 48 бит, а не 32, как нам нужно.

4. Введение в архитектуру ЭВМ

4.1. Адресность ЭВМ

Как мы уже упоминали, число адресов в команде называется *адресностью* ЭВМ. Разнообразие архитектур ЭВМ предполагает, в частности, и различную адресность команд. Рассмотрим схему выполнения команд с различным числом адресов операндов. Будем предполагать, что для хранения кода операции в команде отводится один байт (8 разрядов), а для хранения каждого из адресов – 3 байта (это обеспечивает объём памяти 2^{24} ячеек). Ниже приведены форматы команд для ЭВМ различной адресности и схемы выполнения этих команд для случая бинарных операций (у таких операций два операнда и один результат).

- **Трёхадресная машина.**

КОП	A1	A2	A3
-----	----	----	----

8 разрядов 24 разряда 24 разряда 24 разряда = 10 байт

Схема выполнения команд такой машины нам уже известна:

$R1 := \langle A2 \rangle; R2 := \langle A3 \rangle; S := R1 \otimes R2; \langle A1 \rangle := S; \{\otimes - \text{операция}\}$

- **Двухадресная машина.**

КОП	A1	A2
-----	----	----

8 разрядов 24 разряда 24 разряда = 7 байт

Схема выполнения команд:

$R1 := \langle A1 \rangle; R2 := \langle A2 \rangle; S := R1 \otimes R2; \langle A1 \rangle := S;$

Заметим, что теперь для выполнения бинарной операции первый и второй операнды задаются *явно* в качестве адресов в команде, а местоположение результата операции задаётся неявно или, как говорят, *по умолчанию*. В рассмотренном выше случае результат операции по умолчанию помещается на место первого операнда, уничтожая его.

- **Одноадресная машина.**

КОП	A1
-----	----

8 разрядов 24 разряда = 4 байта

Схема выполнения команд:

$R1 := \langle A1 \rangle; S := S \otimes R1;$

Для работы в одноадресной машине необходимы ещё две команды, которые имеют один операнд и один результат и выполняются по другим схемам. Это команда чтения числа из памяти на регистр сумматора:

СЧ A1

Она выполняется по схеме

$S := \langle A1 \rangle$

и команда записи значения из сумматора в память:

ЗП A1

Она выполняется по схеме

$\langle A1 \rangle := S$

При выполнении бинарных операций в одноадресной ЭВМ только один второй операнд задаётся в команде явно, а первый операнд и результат задаются неявно – это регистр сумматора.

- **Безадресная машина.**

КОП

8 разрядов = 1 байт

В отличие от других рассмотренных выше машин, безадресная машина использует при работе аппаратно реализованный в компьютере стек, для чего вводятся две дополнительные *одноадресные* команды: записи из памяти в стек

ВСТЕК A1

которая выполняется по схеме

$R1 := \langle A1 \rangle; \text{ВСТЕК}(R1)$

и команда чтения из стека

ИЗСТЕКА A1

которая выполняется по схеме

ИЗСТЕКА (R1); <A1> := R1

Таким образом, за исключение двух указанных выше одноадресных команд, которые имеют длину 4 байта, все остальные команды являются безадресными, имеют длину 1 байт и выполняются по схеме:

R1 := ИЗСТЕКА; R2 := ИЗСТЕКА; S := R1 ⊗ R2; ВСТЕК(S)

Как видно, для безадресных команд при выполнении бинарных операций уже все аргументы (два операнда и результат) задаются неявно и располагаются в стеке. Отсюда понятно, почему часто машины этой архитектуры называются *стековыми ЭВМ*.

Кроме рассмотренных видов машин, существовали и другие архитектуры ЭВМ, например, **четырёхадресные**, в четвёртом адресе которых дополнительно хранится ещё и адрес следующей выполняемой команды. Собственно, адресов может быть и больше, с помощью таких команд можно, например, реализовать функции от многих переменных.

Существуют архитектуры ЭВМ, которые различаются не только количеством адресов в команде, но и наличием в команде нескольких *кодов операций*. Такие ЭВМ называются машинами с *очень длинным командным словом* (VLIW – very large instruction word). В этих компьютерах, например, указанные команды могут реализовывать оператор присваивания вида $z := k * (x + y)$ по схеме:

R1 := <x>; R2 := <y>; S := R1 + R2;
R1 := <k>; S := S * R1; <z> := S

В компьютерах с такой архитектурой команда содержит два кода операции и четыре адреса аргументов:

КОП1	КОП2	A1	A2	A3	A4
------	------	----	----	----	----

Такие команды могут выполняться, например, по схеме:

R1 := <A2>; R2 := <A3>; S := R1 КОП1 R2;
R1 := <A4>; S := S КОП2 R1; <A1> := S

4.2. Сравнительный анализ ЭВМ различной адресности

При изучении ЭВМ с разным количеством адресов естественно встаёт вопрос, какая архитектура лучше, например, даёт программы, занимающие меньше места в памяти (что было весьма актуально для первых ЭВМ). Исследуем этот вопрос, составив небольшой фрагмент программы для ЭВМ с различной адресностью. В качестве примера рассмотрим реализацию оператора присваивания, который содержит типичный набор операций: $x := a / (a + b)^2$. В наших примерах мы будем использовать мнемонические коды операций и мнемонические имена для номеров ячеек памяти, в которых хранятся переменные (т.е. мы не будем производить явного распределения памяти, так как это несущественно для нашего исследования). Кроме того, не будем конкретизировать тип величин, это тоже не влияет на размер программы.

- **Трёхадресная машина.**

СЛ	x	a	b	X := a + b
УМН	x	x	X	X := (a + b) ²
ДЕЛ	x	a	x	X := a / (a + b) ²

Длина программы: 3*10 = 30 байт.

- **Двухадресная машина.**

ПЕР	R	a	R := a
СЛ	R	b	R := a + b
УМН	R	R	R := (a + b) ²
ПЕР	X	a	x := a;
ДЕЛ	X	R	x := a / (a + b) ²

Длина программы: 5*7 = 35 байт.

- **Одноадресная машина.**

СЧ	A	S := a
СЛ	B	S := a + b
ЗП	X	x := a + b

УМН	X	$x := (a+b)^2$
ЗП	X	
СЧ	A	$S := a/(a+b)^2$
ДЕЛ	X	
ЗП	X	

Длина программы: $8 \cdot 4 = 32$ байта.

- **Безадресная машина.**

ВСТЕК	A	Поместить a в стек
ВСТЕК		Дублировать вершину стека
ВСТЕК	B	Теперь в стеке 3 числа: b, a, a
СЛ		В стеке два числа: b+a, a
ВСТЕК		Дублировать вершину стека, в стеке b+a, b+a, a
УМН		В стеке $(a+b)^2, a$
ОБМЕН		Поменять местами два верхних элемента стека
ДЕЛ		В стеке $a/(a+b)^2$
ИЗСТЕКА	X	Запись результата из стека в x

В данной программе использовались команды разной длины (безадресные и одноадресные). Длина программы: $3 \cdot 4 + 6 \cdot 1 = 18$ байт.

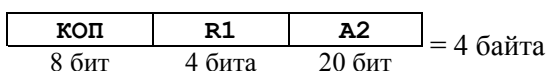
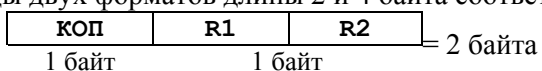
Наше небольшое исследование показало, что архитектура ЭВМ с безадресными командами даёт наиболее компактные программы. В начале развития вычислительной техники такие компьютеры были весьма распространены, их, в частности, выпускала известная фирма Барроуз (Burroughs). Однако в дальнейшем были предложены ЭВМ с другой архитектурой, которая позволила писать не менее компактные программы, и в настоящее время стековые ЭВМ используются редко.

4.3. Дробно-адресная архитектура

Далее мы рассмотрим архитектуру ЭВМ, которые называются компьютерами с адресуемыми регистрами, в русскоязычной литературе они часто называются *дробно-адресными* [3,4] (смысл этого названия мы скоро выясним). Эти компьютеры должны давать возможность писать такие же компактные программы, как и компьютеры с безадресной системой команд, но при этом они обладают рядом дополнительных достоинств.

Компьютеры дробно-адресной архитектуры нарушают один из принципов фон Неймана – принцип однородности памяти. Будем считать, что память, к которой может непосредственно обращаться центральный процессор, состоит из двух частей, каждая со своей независимой нумерацией ячеек. Одна из этих частей называется *адресуемой регистровой памятью* и имеет небольшой объём (порядка десятков ячеек), а другая называется *основной (оперативной) памятью* большого объёма. Ячейка каждого из видов памяти имеет свой адрес, но в случае с маленькой регистровой памятью этот адрес имеет размер в несколько раз меньший, чем адрес ячейки основной памяти.

Например, рассмотрим двухадресную ЭВМ, в которой регистровая память состоит из 16 ячеек. В этом случае адрес каждого регистра лежит в диапазоне $0 \div 15$, и будет помещаться в 4 бита, а основная память содержит 2^{20} ячеек и адрес каждой ячейки занимает 20 двоичных разрядов. В такой ЭВМ в качестве адресов операндов каждой команды могут быть или адреса двух регистров, или адрес регистра и адрес ячейки основной памяти. Адреса регистров на схемах команд будем обозначать R1 и R2, а адрес основной памяти A1 или A2. Первый вид команд будем называть командами *формата регистр-регистр* (обозначается RR), а вторые – *формата регистр-память* (обозначается RX). В этом случае для одного кода операции (например, сложения) мы получим команды двух форматов длины 2 и 4 байта соответственно:



В качестве преимущества этой архитектуры нужно отметить, что ячейки регистровой памяти размещаются *внутри* центрального процессора, и, имея статус регистров, позволяют производить на них арифметические и логические операции (что, как мы помним, в основной памяти невозможно). Кроме того, это обеспечивает быстрый доступ к хранимым на регистрах данным (не требуется делать обмен с расположенной отдельно от центрального процессора основной памятью).

Скажем теперь, что такая архитектура получила название *дробно-адресной* потому, что адрес ячейки регистровой памяти составляет какую-то часть адреса ячейки большой основной памяти. В нашем примере соответствующее отношение равно правильной дроби 1/5.

Из рассмотренного выше можно сделать вывод, что при программировании на ЭВМ с такой архитектурой желательно как можно чаще оперировать с регистровой памятью и как можно реже обращаться к большой основной памяти, такого принципа мы и будем придерживаться. Теперь для нашей дробно-адресной машины составим фрагмент программы, который реализует, как и в предыдущих примерах, арифметический оператор присваивания $x := a / (a+b)^2$. Мнемонические коды операций задают арифметические операции с обычным смыслом. Точка с запятой, как это принято в языке Ассемблера, задаёт *комментарий* к команде:

```

...
СЧ R1, a; R1 := a
СЧ R2, b; R2 := b
СЛ R2, R1; R2 := b+a=a+b
УМН R2, R2; R2 := (a+b)2
ДЕЛ R1, R2; R1 := a / (a+b)2
ЗП x, R1; x := R1 = a / (a+b)2
...

```

Длина этого фрагмента программы равна $3 \cdot 4 + 3 \cdot 2 = 18$ байт. Как видим, данная архитектура не уступает стековой (безадресной) архитектуре по длине получаемых программ.

Рассмотрим теперь недостатки дробно-адресной архитектуры ЭВМ. Если ранее для каждой арифметической операции было необходимо реализовать по одной команде для целых и вещественных чисел, то теперь число этих команд возросло вдвое из-за необходимости реализовывать эти команды как в формате RR, так и в формате RX. Это приводит к существенному усложнению устройства управления, которое отныне должно поддерживать большее количество операций.

Однако преимущества дробно-адресной архитектуры настолько очевидны, что её имеют большинство современных машин. Разумеется, в них есть и много новых особенностей, некоторые из которых мы рассмотрим далее в нашем курсе.

При работе с дробно-адресной архитектурой мы встречаемся с командами разного формата (и, соответственно, разной длины). Как говорится, современные ЭВМ обладают *многообразием форматов команд*. Например, на тех компьютерах, на которых Вы сейчас выполняете свои практические работы, реализованы около десяти форматов, а длина команд составляет от 1 до 6 байт.

4.4. Способы адресации

Введём следующее определение. *Способ адресации* – это способ задания операндов внутри машинной команды. Другими словами это правила, по которым заданные в команде (двоичные) числа определяют местонахождение и значение *операндов* для данной команды. Как правило, способ адресации операндов определяется только кодом операции команды. Для лучшего понимания этого понятия рассмотрим операцию сложения двух чисел в одноадресной ЭВМ. Мнемоника кодов операций будет указывать на способ адресации.

- **Прямой способ адресации.**

СЛ	2	S := S + <2>
----	---	--------------

При этом способе адресации (только этот способ мы использовали до сих пор) число на месте операнда задаёт *адрес* ячейки основной памяти, в котором и содержится необходимый в команде операнд. Мы будем в угловых скобках обозначать *содержимое* ячейки основной памяти с данным адресом. Так, в приведённом выше примере <2> обозначает содержимое ячейки с адресом 2. В этой ячейке, конечно же, скорее всего не хранится *число* 2.

- **Непосредственный способ адресации.**

СЛН	2	S := S + 2
-----	---	------------

При таком способе адресации поле адреса команды содержит, как говорят, *непосредственный* операнд. Разумеется, такие операнды могут быть только (неотрицательными) целыми числами, по длине не превышающими максимального значения в поле адреса.

- **Косвенный способ адресации.**

СЛК	2	$S := S + \langle\langle 2 \rangle\rangle$
-----	---	--

Здесь число на месте операнда задаёт *адрес* ячейки памяти, содержимое которой, в свою очередь, трактуется как целое число – адрес необходимого операнда в памяти ЭВМ.

В качестве примера выполним несколько команд сложения с различными способами адресации для одноадресной ЭВМ и рассмотрим значение регистра-сумматора S после выполнения этих команд (см. рис. 4.1). Справа на этом рисунке показаны первые ячейки памяти и хранимые в них целые числа.

	. . .	Адрес	Значение
СЧ	0; S := 0	000	0
СЛ	2; S := 3	001	2
СЛН	2; S := 5	002	3
СЛК	2; S := 13	003	8

Рис. 4.1. Значение регистра сумматора после выполнения команд сложения с различными способами адресации.

Упражнение. Добавьте в язык учебной машины УМ-3 новую команду пересылки, которая использует косвенную адресацию по своему третьему адресу, и покажите, что в этом случае можно обрабатывать массивы без использования самомодифицирующийся программ.

4.5. Многообразие форматов данных

Современные ЭВМ позволяют совершать операции над целыми и вещественными числами разной длины. Это вызвано чисто практическими соображениями. Например, если нужно нам целое число помещается в один байт, но неэкономно использовать под его хранение два или более байта. Во избежание такого неоправданного расхода памяти введены соответствующие *форматы данных*, отражающие представление в памяти ЭВМ чисел разной длины. В зависимости от размера числа, оно может занимать 1, 2, 4 и более байт. Приведённая ниже таблица иллюстрирует *многообразие форматов данных* (для представления целых чисел).

Размер (байт)	Название формата
1	Короткое
2	Длинное
4	Сверхдлинное

Многообразие форматов данных требует усложнения архитектуры регистровой памяти. Теперь регистры должны уметь хранить и обрабатывать данные разной длины.

4.6. Форматы команд

Для операций с разными способами адресации и разными форматами данных необходимо введение различных *форматов команд*, которые, естественно, имеют разную длину. Обычно это такие форматы команд (в скобках указано их мнемоническое обозначение):

- регистр – регистр (RR);
- регистр – память, память – регистр (RX);
- регистр – непосредственный операнд в команде (RI);
- память – непосредственный операнд в команде (SI);
- память – память, т.е. оба операнда в основной памяти (SS).

Многообразие форматов команд и данных позволяет писать более компактные и эффективные программы на языке машины, однако, как уже упоминалось, сильно усложняют центральный процессор ЭВМ.

4.7. Базирование адресов

Для дальнейшего уменьшения объёма программы современные ЭВМ используют *базирование адресов*. Изучение этого понятия проведём на следующем примере. Пусть в программе на одноадресной машине необходимо реализовать арифметический оператор присваивания $X := (A+B)^2$. Ниже приведена эта часть программы с соответствующими комментариями (напомним, что S – это регистр сумматора одноадресной ЭВМ):

```

...
СЧ А; S:=A
СЛ В; S:=A+B
ЗП R; R:=A+B – запись в рабочую переменную
УМ R; S:=(A+B)2
ЗП X; X:=(A+B)2
...

```

Так как в нашем примере одноадресная ЭВМ имеет 2^{24} (примерно 16 миллионов) ячеек памяти, то будем считать, что наш фрагмент программы располагается где-то примерно в середине памяти. Пусть, например, наши переменные располагаются соответственно в следующих ячейках памяти:

```

А – в ячейке с адресом 10 000 000
В – в ячейке с адресом 10 000 001
Х – в ячейке с адресом 10 000 002
R – в ячейке с адресом 10 000 003

```

Тогда приведённый выше фрагмент программы будут выглядеть следующим образом:

```

...
СЧ 10 000 000; S:=A
СЛ 10 000 001; S:=A+B
ЗП 10 000 003; R:=A+B
УМ 10 000 003; S:=(A+B)2
ЗП 10 000 002; X:=(A+B)2
...

```

Из этого примера видно, что большинство адресов в нашей программе имеют вид $V+\Delta$, где число V назовём *базовым адресом* программы или просто *базой* (в нашем случае $V=10\ 000\ 000$), а Δ – *смещением* адреса относительно этой базы. Здесь налицо существенная *избыточность информации*. Очевидно, что в каждой команде можно указывать только короткое *смещение* Δ , а базу хранить отдельно (обычно на каком-то специальном *базовом* регистре центрального процессора). Исходя из этих соображений, предусмотрим в машинном языке команду *загрузки базы* (длина этой команды 4 байта):

ЗГБ	A1
8 бит	24 бита

Тогда наш фрагмент программы будет иметь такой вид:

```

...
ЗГБ 10 000 000
...
СЧ 000; S:=A
СЛ 001; S:=A+B
ЗП 003; R:=A+B
УМ 003; S:=(A+B)2
ЗП 002; X:=(A+B)2
...

```

Теперь, однако, при выполнении *каждого* обращения за операндом в основную память, центральный процессор должен *вычислять* значение адреса этого операнда адреса по формуле $A=V+\Delta$. Это вычисление производится в устройстве управления и, естественно, усложняет его. Например, адрес переменной $A=100000001=V+\Delta=10^7+1$.

Осталось выбрать длину смещения Δ . Вернёмся к рассмотрению *дробноадресной* ЭВМ, для которой реализовано базирование адресов. Например, пусть под запись смещения выделим в команде поле длиной в 12 бит. Будем, как и раньше, обозначать операнд в памяти $A1$ или $A2$, но помним, что теперь это только *смещение* относительно базы. Тогда все команды, которые обращаются за операндом в основную память, будут в нашей *дробноадресной* ЭВМ более короткими:

КОП	R1	A2
------------	-----------	-----------

8 бит	4 бита	12 бит
-------	--------	--------

Схема выполнения такой команды для формата регистр-память:

$$\langle R1 \rangle := \langle R1 \rangle \otimes \langle B+A2 \rangle$$

или для формата память-регистр:

$$\langle B+A2 \rangle := \langle B+A2 \rangle \otimes \langle R1 \rangle$$

Область, в которой находятся вычисляемые относительно базы ячейки основной памяти, обычно называется *сегментом* памяти – это сплошной участок памяти, начало которого задаётся в некотором регистре, называемом *базовым*, или *сегментным*. Будем далее для определённости называть такие регистры сегментными, а сам приём – *сегментированием* памяти.

Сегментирование позволяет уменьшить объём памяти для хранения программ, но оно имеет и один существенный недостаток: теперь каждая команда может обращаться не к любой ячейке оперативной памяти, а только к тем из них, до которых "дотягивается" смещение. В нашем примере каждая команда может обращаться к диапазону адресов от значения сегментного регистра B до $B+2^{12}-1$. Для доступа к другим ячейкам памяти необходимо записать в сегментный регистр новое значение (как говорят, *перезагрузить* сегментный регистр). Несмотря на указанный недостаток, практически все современные ЭВМ производят сегментирование памяти. Заметим также, что этот недостаток в большинстве архитектур современных ЭВМ исправляется путём реализации *переменной* длины смещения (например, разрешается смещение в 1, 2 или 4 байта), что, однако ещё более увеличивает набор команд и усложняет центральный процессор.

Итак, для осуществления доступа к памяти ЭВМ необходимо, чтобы ячейка, к которой осуществляется доступ, находилась в сегменте, на начало которого указывает сегментный регистр. Современные ЭВМ обеспечивают одновременную работу с несколькими сегментами памяти и, соответственно, имеют несколько сегментных регистров.

В некоторых архитектурах регистры центрального процессора являются *универсальными*, т.е. каждый из них может быть использован как сегментный или для выполнения любых операций над данными. Сложность центрального процессора при этом существенно повышается, так что во многих архитектурах используются *специализированные* регистры, т.е. определённые регистры являются сегментными, на других могут производиться операции и т.д.

5. Понятие семейства ЭВМ

Компьютеры могут применяться в самых различных областях человеческой деятельности (эти области часто называются *предметными областями*). В качестве примеров можно привести область научно-технических расчётов (там много операций с вещественными числами), область экономических расчётов (там, в основном, выполняются операции над целыми числами и обработка символьной информации), мультимедийная область (обработка звука, изображения и т.д.), область управления различными сложными устройствами (ракетами, доменными печами и др.)

Компьютеры, архитектура которых ориентирована на какую-то одну предметную область, называются *специализированными*, в отличие от *универсальных* ЭВМ, которые более или менее успешно можно использовать во всех предметных областях. Мы в нашем курсе будем изучать архитектуру только универсальных ЭВМ.

Говорят, что компьютеры образуют *семейство*, если выполняются следующие требования:

1. Одновременно выпускаются и используются несколько *моделей* семейства с различными производительностью и ценой (моделями называются компьютеры-члены семейства).
2. Модели обладают *программной совместимостью*:
 - 1) снизу-вверх – старшие модели поддерживают все команды младших (любая программа, написанная для младшей модели, безошибочно выполняется и на старшей);
 - 2) сверху-вниз – на младших моделях выполняются программы, написанные для старших, если выполнены условия:
 - наличие у младшей модели достаточного количества ресурсов (например, памяти);
 - программа состоит только из поддерживаемых младшей моделью команд.
3. Присутствует *унификация* устройств, то есть их аппаратная совместимость между моделями (например, печатающее устройство для младшей модели должно работать и на старшей).

4. Модели организованы по принципу *модульности*, что позволяет в определённых пределах расширять возможности ЭВМ, увеличивая, например, объём памяти или повышая быстродействие центрального процессора.
5. Стандартизировано системное программное обеспечение (например, компилятор с языка Турбо-Паскаль может работать на всех моделях семейства).

Большинство выпускаемых в наше время ЭВМ содержатся в каких-либо семействах. В нашем курсе для упрощения изложения будут рассматриваться в основном младшие модели семейства ЭВМ компании Intel. Соответственно все примеры программ должны выполняться для всех моделей этого семейства, поэтому мы ограничимся лишь архитектурой и системой команд самой младшей модели этого семейства [9].

6. Архитектура младшей модели семейства Intel

6.1. Память

Архитектура рассматриваемого компьютера является дробно-адресной, поэтому адресуемая память состоит из регистровой и основной памяти. В младшей модели семейства основная память имеет объём 2^{20} ячеек по 8 бит каждая. Регистровая память будет рассмотрена немного позже.

6.2. Форматы данных

- **Целые числа.**

Целые числа могут занимать 8 бит (короткое целое), 16 бит (длинное целое) и 32 бита (сверхдлинное целое). Длинное целое принято называть *машинным словом* (не путать с машинным словом в Учебной Машине!).

Как видим, в этой архитектуре есть многообразие форматов целых чисел, что позволяет писать более компактные программы. Для других архитектур это может оказаться несущественно, например, в некоторых современных супер-ЭВМ идёт работа с малым количеством целых чисел, поэтому вводится только один формат – сверхдлинное целое.

- **Символьные данные.**

В качестве символов используются короткие целые числа, которые трактуются как неотрицательные (беззнаковые) числа, задающие номер символа в некотором алфавите.¹ Заметим, что как таковой символьный тип данных (в смысле языка Паскаль) в Ассемблере отсутствует, а запись 'А' обозначает не символьный тип данных, а эквивалентна выражению языка Паскаль `Ord('A')`.

- **Массивы (строки).**

Массивы могут состоять из коротких или длинных целых чисел. Массив коротких целых чисел может рассматриваться как *символьная строка*. В машинном языке присутствуют команды для обработки *элементов* таких массивов, если такую команду поставить в цикл, то образуются удобное средство для работы с массивами.

- **Вещественные числа.**

Чаще всего используются три формата вещественных чисел: короткие, длинные и сверхдлинные вещественные числа. Стоит отметить следующий важный факт. Если целые числа в различных ЭВМ по чисто историческим причинам иногда имеют разное внутреннее представление, то на момент массового выпуска ЭВМ с командами для работы с вещественными числами уже существовал определённый стандарт на внутреннее представление этих чисел – IEEE (Institute of Electrical and Electronics Engineers), и почти все современные машины этого стандарта придерживаются.

6.3. Вещественные числа

Рассмотрим представление короткого вещественного числа. Такое число имеет длину 32 бита и содержит три поля:

±	Е	М
1 бит	8 бит	23 бита

¹ В настоящее время существуют алфавиты, содержащие большое количество (порядка 32000) символов, для их представления, естественно, используются *длинные* беззнаковые целые числа.

Первое поле из одного бита определяет знак числа (знак "плюс" кодируется нулём, "минус" – единицей). Остальные биты, отведённые под хранение вещественного числа, разбиваются на два поля: машинный порядок E и мантиссу M , которая по модулю меньше единицы. Каждое представимое вещественное число A (кроме числа 0.0) может быть записано в виде: $A = \pm 1.M * 2^{E-127}$. Такие вещественные числа называются *нормализованными*: первый сомножитель удовлетворяет неравенству $1.0 \leq 1.M < 2.0$. Нормализация необходима для однозначного представления вещественного числа в виде двух сомножителей. Нулевое число представляется нулями во всех позициях, за исключением, быть может, первой позиции знака числа.

В качестве примера переведем десятичное число -13.25 во внутреннее машинное представление. Сначала переведем его в двоичную систему счисления:

$$-13.25_{10} = -1101.01_2$$

Затем нормализуем это число:

$$-1101.01_2 = -1.10101_2 * 2^3$$

Следовательно, мантисса будет иметь вид 10101000000000000000_2 , осталось вычислить машинный порядок: $3 = E - 127$; $E = 130 = 128 + 2 = 10000010_2$. Теперь, учитывая знак, получаем вид внутреннего машинного представления числа -13.25_{10} :

$$1100\ 0001\ 0101\ 0100\ 0000\ 0000\ 0000\ 0000_2 = C1500000_{16}$$

Шестнадцатеричные числа в языке Ассемблера принято записывать с буквой h на конце:

$$C1500000_{16} = C1500000h$$

Таков формат короткого вещественного числа. Согласно его виду, E изменяется от 0 до 255, следовательно, диапазон порядков коротких вещественных чисел равен $2^{-127} \dots 2^{128} \approx 10^{-38} \dots 10^{38}$. Как и для целых чисел, машинное представление которых мы рассмотрим чуть позже, число представимых вещественных чисел *конечно*. Заметим также, что, в отличие от целых чисел, в представлении вещественных чисел используется *симметричная* числовая ось, то есть для любого положительного числа найдётся соответствующее ему отрицательное (и наоборот).

Некоторые комбинации нулей и единиц в памяти, отведённой под вещественное число, собственно числа не задают, а используются для служебных целей. В частности, $E=255$ обозначает специальное значение "*не число*" (NaN – not a number). При попытке производить арифметические операции над такими "числами" возникает аварийная ситуация. Например, значение "не число" может быть присвоено вещественной переменной при её порождении, если эта переменная не имеет начального значения (как говорят, *не инициализирована*). Такой приём позволяет избежать тяжёлых семантических ошибок, которые могут возникать при работе с неинициализированными переменными, которые при порождении, как правило, имеют случайные значения.

Отметим ещё две специальные комбинации нулей и единиц, которые будем обозначать $\pm\epsilon$. Эти значения присваиваются результату операции с вещественными числами, если этот результат, хотя и не равен нулю, но не представим в виде вещественного числа, то есть $\pm\epsilon$ меньше самого маленького представимого положительного вещественного числа и больше самого большого отрицательного.

Аналогично существуют комбинации битов, задающие специальные значения $\pm\infty$. Эти значения выдаются в качестве результата, если этот результат такой большой по абсолютной величине, что не представим среди множества машинных вещественных чисел.

Центральный процессор "разумно" (по крайней мере с точки зрения математика) производит арифметические операции над такими "числами". Например, пусть A любое представимое вещественное число, тогда

$$A \pm \epsilon = A; \pm\epsilon * A = \pm\epsilon; A * \pm\infty = \pm\infty; \text{ и т.д. }^1$$

Для любознательных студентов заметим, что существует нетрадиционное построение математического анализа, в котором, как и в нашей ЭВМ, бесконечно малые величины $\pm\epsilon$ определяются не в виде пределов, как в обычном анализе, а существуют в виде "настоящих" вещественных чисел. Изложение нетрадиционного анализа можно посмотреть в книгах [13,14].

При изучении архитектуры ЭВМ вещественные числа не будут представлять для нас большого интереса и поэтому (а также из-за недостатка времени) операции над вещественными числами мы изучать не будем.

¹ При этом правильно учитывается знак числа, например, $(-8) * (-\epsilon) = +\epsilon$

6.4. Целые числа

Мы уже знаем, что хранимые в памяти машинные слова (наборы битов) могут трактоваться по-разному. При вызове в устройство управления этот набор битов трактуется как команда, а при вызове в арифметико-логическое устройство – как число. В дополнении к этому в рассматриваемой нами архитектуре каждое хранимое в памяти целое число может трактоваться программистом как *знаковое* или *беззнаковое* (неотрицательное). По внешнему виду *невозможно* определить, какое число хранится в определённом месте памяти, только сам программист может знать, как он *рассматривает* это число. Таким образом, определены две машинные *системы счисления* для представления знаковых и беззнаковых чисел соответственно.

Беззнаковые (неотрицательные) числа представляются в уже известной Вам двоичной системе счисления, такое представление называется *прямым кодом* неотрицательного числа. Например, десятичное число 13, хранимое в одном байте, будет записано как прямой код 00001101.

Если *инвертировать* прямой код (т.е. заменить все "1" на "0", а все "0" на "1"), то получим так называемый *обратный* код числа. Например, обратный код числа 13 равен 11110010.

Для представления *отрицательных* знаковых чисел используется так называемый *дополнительный* код, который можно получить из дополнительного кода прибавлением единицы. Например, получим дополнительный код числа -13 :

$$\begin{array}{r}
 \text{Прямой код} \quad = 00001101 \\
 \text{Обратный код} \quad = 11110010 \\
 \quad \quad \quad \quad + \quad \quad \quad \quad \underline{\quad \quad \quad 1} \\
 \text{Дополнительный код} = 11110011
 \end{array}$$

Существует и другой алгоритм преобразования отрицательного числа X в дополнительный код. Для этого необходимо записать в прямом коде значение $2^N - |X|$, где значение N равно максимальному числу бит в представлении числа (в нашем примере $N=8$).

Итак, в знаковой системе счисления отрицательные числа представляются в дополнительном коде, а неотрицательные – в прямом коде. Заметим, что при знаковой трактовке целых чисел крайний правый бит определяет знак числа ("1" для отрицательных чисел). Этот бит называется *знаковым* битом целого числа. Для знаковых чисел числовая ось несимметрична: количество отрицательных чисел на единицу больше, чем количество положительных чисел (докажите это!).

Процесс перехода от прямого кода к дополнительному коду и обратно с технической точки зрения очень прост и может быть легко реализован в центральном процессоре.

Очень важно понять, что все арифметические операции над знаковыми и беззнаковыми целыми числами производятся абсолютно по одинаковым алгоритмам, что естественно, потому что центральный процессор "не знает", какие это числа "на самом деле". В то же время, с точки зрения программиста, результаты таких операций могут быть разными для знаковых и беззнаковых чисел. Рассмотрим примеры сложения двух чисел длиной в байт. В первом столбике записано внутреннее двоичное представление чисел, а во втором и третьем – беззнаковое и знаковое десятичное представления этих же чисел.

- **Пример 1.**

	Б/з.	Знак.
11111100	252	-4
00000101	5	5
100000001	1	1

Из этого примера видно, что для знаковой трактовки чисел операция сложения выполнена правильно, а при рассмотрении чисел как беззнаковые результат будет неправильным, так как мы получим девятизначное двоичное число, не "умещающееся" в один байт. Так как центральный процессор "не знает", как программист будет трактовать складываемые числа, то он "на всякий случай" сигнализирует о том, что при сложении беззнаковых чисел произошла ошибка.

Для обозначения таких (и некоторых других) ситуаций в архитектуре компьютера введено понятие *флагов*. Каждый флаг занимает один бит в специальном *регистре флагов* (FLAGS). В данном случае флаг CF (carry flag) примет значение, равное единице (иногда говорят – флаг *поднят*). Рассматривая результат в знаковых числах, мы получили правильный ответ, поэтому соответствующий флаг OF (overflow flag) будет положен равным нулю (*опущен*).

- **Пример 2.**

	Б/з.	Знак.
01111001	121	121
00001011	11	11
10000100	132	-124

В данном примере ошибка будет, наоборот, в случае со знаковой трактовкой складываемых чисел, поэтому флаги CF и OF принимают соответственно значения 0 и 1.

- **Пример 3.**

	Б/з.	Знак.
11110110	246	-10
10001001	137	-119
101111111	383	+127

В данном случае результат будет ошибочен как при беззнаковой, так и при знаковой трактовке складываемых чисел. Содержимое флагов: CF = OF = 1. Легко придумать пример, когда результат сложения правильный как для знаковых, так и для беззнаковых чисел (сделайте это самостоятельно!).

Кроме формирования флагов CF и OF команда сложения целых чисел меняет и значения некоторых других флагов в регистре флагов FLAGS. Для нас будет важен флаг SF, в который заносится

знаковый (крайний правый) бит результата, и флаг ZF, который устанавливается в 1, если результат равен нулю, в противном случае этот флаг устанавливается в 0.

Представление отрицательных чисел в дополнительном коде не очень удобно для программистов, однако, позволяет существенно упростить арифметико-логическое устройство. Заметим, например, что вычитание можно выполнять как сложение с дополнительным кодом числа.

Основная причина использования двух систем счисления для представления целых чисел заключается в том, что при использовании *обеих* систем счисления диапазон представимых целых чисел увеличивается в полтора раза. Это было весьма существенно для первых ЭВМ с их небольшим объёмом памяти.

6.5. Сегментация памяти

Память нашей ЭВМ имеет уже знакомую нам сегментную организацию. В любой момент времени для младшей модели определены четыре сегмента (хотя для старших моделей число сегментов может быть и больше). Это означает, что есть четыре *сегментных* регистра, которые указывают на определённые области памяти. Каждый сегментный регистр имеет длину 16 разрядов, поэтому для того, чтобы сегмент мог располагаться на любом месте оперативной памяти, адрес начала сегмента получается после умножения значения сегментного регистра на число 16. Правда, при таком способе задания начала сегмента, он может начинаться не с любого места оперативной памяти, а только с адресов, кратных 16.

Итак, извлекать из памяти числа или команды можно только относительно одного из них этих сегментных регистров. Таким образом, физический адрес числа или команды вычисляется по формуле

$$A_{\text{физ}} := (SR * 16 + A) \bmod 2^{20},$$

где SR – значение сегментного регистра, а A – *смещение*. Физический адрес берётся по модулю 2^{20} , чтобы он не вышел за максимальный адрес памяти.

В качестве мнемонических обозначений сегментных регистров выбраны следующие двухбуквенные служебные¹ имена: кодовый сегментный регистр (CS), сегментный регистр данных (DS), сегментный регистр стека (SS) и дополнительный сегментный регистр (ES). Каждый из них может адресовать сегмент памяти длиной от 1 до 2^{16} байт (напомним, что вся память состоит из 2^{20} ячеек). Так как физический адрес в приведённой выше формуле берётся по модулю 2^{20} , то очевидно, что память "замкнута в кольцо". Таким образом, в одном сегменте могут находиться ячейки с самыми большими и самыми маленькими адресами основной памяти. На рис. 6.1 показан пример расположения сегментов в памяти.

Стоит отметить, что сегментные регистры являются специализированными, предназначенными только для хранения адресов сегментов, поэтому арифметические операции над их содержимым не предусмотрены.

Заметим, что даже если все сегменты не перекрываются и имеют максимальный размер, то и в этом случае центральный процессор в каждый момент времени имеет доступ только к одной четвёртой от общего объёма оперативной памяти.

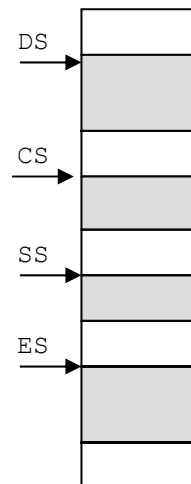
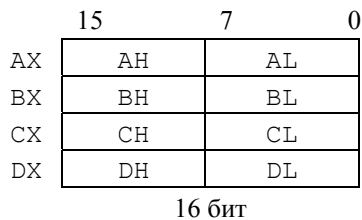


Рис. 6.1. Пример расположения сегментов в

6.6. Мнемонические обозначения регистров

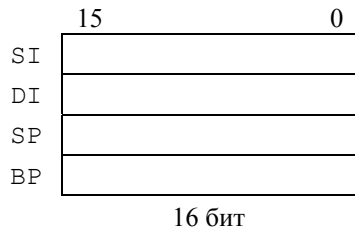
В силу того, что в ЭВМ все регистры имеют безликие двоичные обозначения, программисты предпочитают использовать мнемонические названия регистров. Регистры общего назначения, каждый из которых может складывать, вычитать и просто хранить данные, а некоторые – умножать и делить, обозначают следующими именами: AX, BX, CX, DX. Для обеспечения многообразия форматов данных каждый из них разбит на две части по 8 бит каждая (биты нумеруются немного непривычно справа налево, начиная с нуля):

¹ Эти имена (как и имена всех остальных регистров нашей ЭВМ) являются служебными в языке Ассемблера. Напомним, что служебные имена нельзя использовать ни в каком другом смысле.



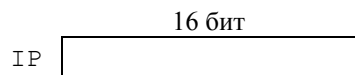
Каждый из регистров AH, AL, BH, BL, CH, CL, DH и DL может быть использован как самостоятельный регистр, на которых возможно выполнять операции сложения и вычитания.

Существуют также четыре регистра SI, DI, SP и BP, которые также могут использоваться для проведения сложения и вычитания, но уже не делятся на половинки:

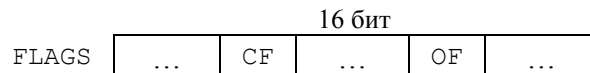


В основном эти четыре регистра используются как *индексные*, т.е. на них хранятся положение конкретного элемента в некотором массиве.

Кроме перечисленных выше регистров программист имеет дело с регистром IP (instruction pointer), который называется счётчиком адреса и содержит адрес следующей исполняемой команды (точнее, содержит *смещение* относительно начала кодового сегмента, адрес начала этого сегмента, как мы уже знаем, равен значению сегментного регистра CS, умноженному на 16).



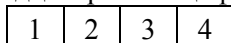
И, наконец, как уже упоминалось, архитектурой изучаемой ЭВМ предусмотрен регистр флагов FLAGS. Он содержит шестнадцать одноразрядных флагов, например, ранее упоминавшиеся флаги CF и OF. Конкретные номера битов, содержащих тот или иной флаг, для понимания архитектуры несущественны, и приводиться здесь не будут.



Все биты в регистрах пронумерованы справа налево: в шестнадцатибитных – от 0 до 15, в восьмибитных – от 0 до 7.

Все упомянутые имена регистров являются служебными в языке Ассемблера. Как и в языке Паскаль, в языке Ассемблера принято соглашение по синтаксису имён: регистр символов не различается, таким образом, AX, Ax, aX и ax обозначают *один и тот же* регистр.

Рассмотрим теперь способ хранения чисел в памяти ЭВМ. Запишем, например, шестнадцатеричное число 1234h в какой-нибудь 16-тиразрядный регистр (каждая шестнадцатеричная цифра занимает по 4 бита):



Теперь поместим это число в память в ячейки с номерами, например, 100 и 101. Так вот: в ячейку с номером 100 запишется число 34h, а в ячейку 101 – число 12h. Говорят, что число представлено в основной памяти (в отличие от регистров) в *перевернутом* виде. Это связано с тем, что в младших моделях ЭВМ при каждом обращении к памяти читался один байт. Чтобы считать слово, было необходимо дважды обратиться к памяти, поэтому было удобно (например, для проведения операция сложения "в столбик") получать сначала младшие цифры числа, а затем – старшие. В современной архитектуре за одно обращение из памяти получают сразу 4, 8 или 16 байт, но из-за совместимости моделей семейства пришлось оставить *перевернутое* представление чисел. Заметим, что в отличие от чисел, *команды* хранятся в памяти в обычном (не перевернутом) виде.

6.7. Структура команд

Теперь рассмотрим структуру машинных команд самых распространённых форматов регистр-регистр и регистр-память.

• **Формат регистр–регистр.**

6 бит	1 бит	1 бит	1 бит	1 бит	3 бита	3 бита
КОП	d	w	1	1	R1	R2

Команды этого формата занимают 2 байта. Первая часть команды – код операции – занимает в команде 6 бит, за ним следуют однобитные поля d и w, где d – бит *направления*, а w – бит *размера аргумента*, последующие два бита для этого формата равны 1, а последние две части по 3 бита каждая указывают на регистры – операнды.

Стоит подробнее рассмотреть назначение битов d и w. Бит d задаёт направление выполнения команды, а именно:

<R1> := <R1> ⊗ <R2> при d = 0

<R2> := <R2> ⊗ <R1> при d = 1.

Бит w задаёт размер регистров-операндов, имена которых можно определить по следующей схеме:

R _{1,2}	w = 1	w = 0
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

В младших моделях ЭВМ нашего семейства в таком формате возможна работа лишь с упомянутыми в таблице регистрами. В последующих же моделях возможности этого формата были расширены, но за счёт увеличения длины команды. Мы будем пользоваться и другими видами формата регистр-регистр, например, командой `mov ds, ax`, но выписывать их внутреннее представление не будем.

Как видно из таблицы, архитектурой не предусмотрены операции формата r8–r16, т.е. операции над регистрами разной длины запрещены, например, команды типа `add AL, BX` являются некорректными. Поэтому появляется необходимость *преобразования типов* из короткого целого в длинное, и из длинного в сверхдлинное. Такое преобразование зависит от трактовки числа – знаковое или беззнаковое. В первом случае число всегда расширяется слева нулями, а во втором – размножается знаковый бит (для знаковых чисел незначащими двоичными цифрами будут 0 для неотрицательных и 1 для отрицательных значений). Для этого в языке машины предусмотрены безадресные команды, имеющие в Ассемблере такую мнемонику:

cbw (convert byte to word)

и

cwd (convert word to double),

которые производят *знаковое* расширение соответственно регистра AL до AX и AX до пары регистров <DX, AX>, которые в этом случае рассматриваются как один длинный 32 битный регистр.

Преобразование целого значения из более длинного формата в более короткий (усечение) производится путём отбрасывания соответствующего числа *левых* битов целого числа. Усечённое число получится правильным, если будут отброшены только *незначащие* биты. Для беззнаковых чисел это всегда нулевые биты, а для знаковых – биты, совпадающие со знаковым битом *усечённого* числа.

• **Формат регистр–память (и память-регистр).**

КОП	R1	A2
-----	----	----

Операнд A2 может иметь один из приведённых ниже трёх видов:

1. A2 = A,
2. A2 = A[M1],
3. A2 = A[M1][M2].

Здесь A – задаваемое в команде смещение длиной 1 или 2 байта (заметим, что нулевой смещение может и не занимать места в команде), M1 и M2 – так называемые *регистры-модификаторы*. Как мы

сейчас увидим, значение адреса второго операнда A2 будет *вычисляться* по определённым правилам, поэтому этот адрес часто называют *исполнительным* адресом.

Стоит отметить один факт. До сих пор *адресом* мы называли физический номер ячейки в памяти машины. В языке Ассемблера *адресом* принято называть *смещение* ячейки относительно начала того сегмента, в котором она находится. Для обозначения полного адреса будем употреблять термин *физический адрес*.

Рассмотрим подробнее каждый их трёх возможных видов операнда A2. При $A2 = A$ физический адрес вычисляется центральным процессором по формуле:

$$A_{\text{физ}} := (B * 16 + A) \bmod 2^{20},$$

где B, как обычно, обозначает значение одного из сегментных регистров. Запись $A2 = A[M1]$ означает использование *регистра-модификатора*, которым может быть любой из следующих регистров: BP, BX, SI, DI. В этом случае физический адрес вычисляется по формуле

$$A_{\text{физ}} := (B * 16 + (A + \langle M1 \rangle) \bmod 2^{16}) \bmod 2^{20},$$

где вместо $\langle M1 \rangle$ подставляется содержимое регистра-модификатора (одного из четырёх указанных). Запись $A2 = A[M1][M2]$ ¹ обозначает вычисление физического адреса по формуле:

$$A_{\text{физ}} := (B * 16 + (A + \langle M1 \rangle + \langle M2 \rangle) \bmod 2^{16}) \bmod 2^{20},$$

где используются сразу два регистра-модификатора. На месте M1 можно указывать любой из регистров BX или BP, а на месте M2 – любой из регистров SI или DI. Использование, например, регистров BX и BP (как и SI и DI) одновременно в качестве модификаторов запрещено. В старших моделях почти все ограничения на использование регистров модификаторов также было снято (за счёт увеличения длины команды).

В качестве примера вычислим физический адрес второго операнда команды сложения формата RX, на языке Ассемблера эту команду можно записать в виде `add ax, 6[bx][di]`. Пусть регистры имеют следующие значения (в шестнадцатеричном виде перед числом записывается ноль, если оно начинается с цифр A–F):

$$bx = 0FA00h, di = 0880h, ds = 2000h$$

Тогда

$$A_{\text{физ}} := (2000h * 16 + (6 + 0FA00h + 0880h) \bmod 2^{16}) \bmod 2^{20} = \\ (20000h + 0286) \bmod 2^{20} = 20286h$$

Если, например, в байте с адресом 20286h хранится число 56h, а в байте с адресом 20287h – число 32h, то наша команда реализует операцию сложения $ax := ax + 3256h$.

Рассмотрим теперь внутреннее представление формата команды регистр–память. Длина этой команды 2, 3 или 4 байт:

8 бит		2 бита		3 бита	3 бита	8 бит	8 бит
КОП	d	w	Mod	R1	Mem	a8	a8->a16

где mod – трёх битовое поле модификатора, mem – двух битовое поле способа адресации, a8 и a16 – это обозначения для одно- или двухбайтного смещение. Биты d и w знакомы нам из предыдущего формата регистр–регистр. Все возможные комбинации mod и mem приведены в таблице 6.1.

Таблица 6.1. Значения полей mod и mem в формате регистр–память.

Mem \ mod	00	01	10	11
	0 доп. Байт.	1 доп. байт	2 доп. байта	Формат RR
000	[BX+SI]	[BX+SI]+a8	[BX+SI]+a16	
001	[BX+DI]	[BX+DI]+a8	[BX+DI]+a16	
010	[BP+SI]	[BP+SI]+a8	[BP+SI]+a16	
011	[BP+DI]	[BP+DI]+a8	[BP+DI]+a16	
100	[SI]	[SI]+a8	[SI]+a16	
101	[DI]	[DI]+a8	[DI]+a16	
110	a16	[BP]+a8	[BP]+a16	
111	[BX]	[BX]+a8	[BX]+a16	

¹ В некоторых ассемблерах допускается эквивалентная запись выражения $A[M1][M2]$ в виде $A[M1+M2]$ и даже в виде $A+M1+M2$.

Данная таблица показывает, как зависит способ адресации от полей `mem` и `mod`. Как видим, она полностью объясняет ограничения на выбор регистров-модификаторов, которые мы сформулировали ранее. Мы не будем рассматривать машинный вид остальных форматов команд, будем изучать их только на языке Ассемблера. Напомним, что это такие форматы команд:

- RR – (регистр – регистр);
- RX – (регистр – память, память – регистр);
- RI – (регистр – непосредственный операнд в команде);
- SI – (память – непосредственный операнд в команде);
- SS – (память – память, т.е. оба операнда в основной памяти).

6.8. Команды языка машины

Далее мы будем изучать синтаксис машинных команд и семантику их выполнения центральным процессором. Для удобства команды будем записывать так, как это принято в языке Ассемблер (можно считать, что мы уже начали понемногу изучать этот язык).

6.8.1. Команды пересылки

Команды пересылки – одни из самых распространённых команд в языке машины. Все они пересылают один или два байта из одного места памяти в другое. Для более компактного описания синтаксиса команд введём следующие условные обозначения:

`r8` – любой короткий регистр `AH, AL, BH, BL, CH, CL, DH, DL`;

`r16` – любой из длинных регистров `AX, BX, CX, DX, SI, DI, SP, BP`;

`m8, m16` – операнды в основной памяти длиной 1 и 2 байта соответственно;

`i8, i16` – непосредственные операнды в самой команде длиной 1 и 2 байта соответственно;

`SR` – один из сегментных регистров `SS, DS, ES`;

`m32` – операнд в основной памяти длиной 4 байта.

Общий вид команды пересылки в нашей двухадресной ЭВМ такой (после точки с запятой будем записывать, как это принято в Ассемблере, *комментарий* к команде):

`mov op1, op2; op1 := op2`

Существуют следующие допустимые форматы операндов команды пересылки:

op1	op2
R8	r8, m8, i8
R16	r16, m16, i16, SR, CS
M8	r8, i8
M16	r16, i16, SR, CS
SR	r16, m16

Команды пересылки не меняют флаги в регистре `FLAGS`.

6.8.2. Арифметические команды

Изучение команд для выполнения арифметических операций начнём с команд сложения и вычитания целых чисел. Определим вид и допустимые операнды у команд сложения и вычитания:

`КОП op1, op2`, где `КОП = add, sub, adc, sbb`.

`add` – сложение,

`sub` – вычитание:

`op1 := op1 ± op2`

`adc` – сложение с учётом флага переноса,

`sbb` – вычитание с учётом флага переноса:

`op1 := op1 ± op2 ± CF`

Таблица допустимых операндов для этих команд:

op1	op2
R8	r8, m8, i8
M8	r8, i8
R16	r16, m16, i16
M16	r16, i16

В результате выполнения операций изменяются флаги CF, OF, ZF, SF, которые отмечают соответственно за перенос, переполнение, нулевой результат и знак результата (флагу SF всегда присваивается знаковый бит результата). Эти команды меняют и некоторые другие флаги (см. [5,9]), но это нас интересовать не будет.

Далее рассмотрим команды умножения и деления целых чисел. Формат этих команд накладывает сильные ограничения на месторасположение операндов. Первый операнд всех команд этого класса явно в команде не указывается и находится в фиксированном регистре, принимаемом *по умолчанию*. В младшей модели семейства есть следующие команды умножения и деления, в них явно задаётся только второй операнд (второй сомножитель или делитель):

```
mul op2 – беззнаковое умножение,
imul op2 – знаковое умножение,
div op2 – беззнаковое целочисленное деление,
idiv op2 – знаковое целочисленное деление.
```

Как видим, в отличие от команд сложения и вычитания, умножение и деление знаковых и беззнаковых целых чисел выполняются *разными* командами (по разным алгоритмам).

В случае с короткими целыми операндами при умножении вычисление производится по формуле:

```
AX := AL * op2
```

При делении (операции **div** и **mod** понимаются в смысле языка Паскаль):

```
AL := AX div op2
AH := AX mod op2
```

В случае с длинными операндами при умножении вычисление производится по формуле:

```
(DX,AX) := AX * op2
```

При делении:

```
AX := (DX,AX) div op2
DX := (DX,AX) mod op2
```

В этих командах операнд op2 может иметь формат r8, r16, m8 или m16.

Как видим, команды умножения *всегда* дают точный результат, так как под хранение произведения выделяется в два раза больше места, чем под каждый из сомножителей. Команды деления могут вызывать аварийную ситуацию, если частное не помещается в отведённое для него место, т.е. в регистры AL и AX соответственно. Заметим, что остаток от деления всегда помещается в отводимое для него место на регистрах AH или DX соответственно (докажите это!).

После выполнения команд умножения устанавливаются некоторые флаги, из которых для программиста представляют интерес только флаги переполнения и переноса (CF и OF). Эти флаги устанавливаются по следующему правилу. CF=OF=1, если в произведении столько значащих (двоичных) цифр, что они не помещаются в младшей половине произведения. На практике это означает, что при CF=OF=1 произведение коротких целых чисел не помещается в регистр AL и частично "переползает" в регистр AH, а произведение длинных целых чисел – не помещается в регистре AX и "на самом деле" занимает оба регистра (DX, AX). И наоборот, если CF=OF=0, то в старшей половине произведения (соответственно в регистрах AH и DX) находятся только *незначащие* двоичные цифры произведения. Другими словами, при CF=OF=0 в качестве результата произведения можно взять его младшую половину.

Команды деления после своего выполнения как-то устанавливают некоторые флаги, но никакой полезной информации из значения этих флагов программист извлечь не может. Можно сказать, что деление "портит" некоторые флаги.

Для написания программ на Ассемблере нам будут полезны также следующие *унарные* арифметические операции.

```
neg op1 – взятие обратной величины знакового числа, op1 := -op1;
inc op1 – увеличение (инкремент) аргумента на единицу, op1 := op1+1;
dec op1 – уменьшение (декремент) аргумента на единицу, op1 := op1-1;
```

Применение этих команд вместо соответствующих по действию команд вычитания и сложения приводит к более компактным программам. Необходимо также отметить, что команды **inc** и **dec**, в отличие от эквивалентных им команд **add** и **sub** никогда не меняют флаг CF.¹

¹ Это сделано специально, так как схемы центрального процессора, выполняющие инкремент и декремент операнда используются и при выполнении некоторых других команд, которые не должны менять этот флаг.

7. Язык Ассемблера

7.1. Понятие о языке Ассемблера

Наличие большого количества форматов данных и команд в современных ЭВМ приводит к существенным трудностям при программировании на машинном языке. Для упрощения процесса написания программ для ЭВМ был разработан язык-посредник, названный *Ассемблером*, который, с одной стороны, должен быть машинно-ориентированным (допускать написание любых машинных программ), а с другой стороны – позволять автоматизировать процесс составления программ в машинном коде. Для перевода с языка Ассемблера на язык машины используется специальная программа-переводчик, также называемая *Ассемблером* (от английского слова “assembler” – “сборщик”). В зависимости от контекста, в разных случаях под словом “Ассемблер” будет пониматься или язык программирования, или программа-переводчик с этого языка на язык машины.

В нашем курсе мы не будем рассматривать все особенности языка Ассемблера, для этого надо обязательно изучить хотя бы один из учебников [5–8]. Заметим также, что для целей изучения архитектуры ЭВМ нам понадобится только некоторое достаточно небольшое подмножество языка Ассемблера, только оно и будет использоваться на наших лекциях.

Рассмотрим, что, например, должна делать программа Ассемблер при переводе с языка Ассемблера на язык машины.¹

- заменять мнемонические обозначения кодов операций на соответствующие машинные коды операций (например, для нашей учебной машины, *вцц* → 002);
- автоматически распределять память под хранение переменных, что позволяет программисту не заботиться о конкретном адресе переменной, если ему всё равно, где она будет расположена;
- преобразовывать числа, написанные в программе в различных системах счисления во внутреннее машинное представление (в машинную систему счисления).

В конкретном Ассемблере обычно существуют много дополнительных возможностей для более удобного написания программ, однако при этом должны выполняться следующие требования (они вытекают из принципов Фон Неймана):

- возможность помещать в любое определённое программистом место памяти любую команду или любые данные;
- возможность выполнять любые данные как команды и работать с командами, как с данными (например, складывать команды как числа).

7.2. Применение языка Ассемблера

Общеизвестно, что программировать на Ассемблере трудно. Как Вы знаете, сейчас существует много различных языков *высокого уровня*, которые позволяют затрачивать много меньше усилий при написании программ. Естественно, возникает вопрос, когда у программиста может появиться необходимость использовать Ассемблер при написании программ. В настоящее время можно указать две области, в которых использование языка Ассемблера оправдано, а зачастую и необходимо.

Во-первых, это так называемые машинно-зависимые системные программы, обычно они управляют различными устройствами компьютера (такие программы называются драйверами). В этих системных программах используются специальные машинные команды, которые нет необходимости применять в обычных (или, как говорят *прикладных*) программах. Эти команды невозможно или весьма затруднительно задать в языке высокого уровня.

Вторая область применения Ассемблера связана с оптимизацией выполнения программ. Очень часто программы-переводчики (компиляторы) с языков высокого уровня дают весьма неэффективную программу на машинном языке. Обычно это касается программ вычислительного характера, в которых большую часть времени выполняется очень небольшой (порядка 3-5%) участок программы (главный цикл). Для решения этой проблемы могут использоваться так называемые многоязыковые системы программирования, которые позволяют записывать части программы на различных языках. Обычно основная часть программы записывается на языке программирования высокого уровня (Фортране,

¹ Как мы узнаем позже, на самом деле производится перевод не на язык машины, а на специальный промежуточный *объектный* язык.

Паскале, С и т.д.), а критические по времени выполнения участки программы – на Ассемблере. Скорость работы всей программы при этом может значительно увеличиться. Часто это единственный способ заставить программу дать результат за приемлемое время.

При дальнейшем изучении архитектуры компьютера нам придётся писать как фрагменты, так и полные программы на машинном языке. Для написания этих программ мы будем использовать одну из версий языка Ассемблера, так называемый Макроассемблер версии 4.0 (MASM-4.0). Достаточно полное описание этого языка приведено в учебнике [5], изучения этого учебника (или аналогичных учебников по языку Ассемблера [6-8]) является **обязательным** для хорошего понимания материала по нашему курсу. На лекциях мы подробно будем изучать только те особенности и тонкие свойства языка Ассемблера, которые недостаточно полно описаны в указанных учебниках.

Изучение языка Ассемблера начнём с рассмотрения общей структуры программы на этом языке. Программа на языке Ассемблера состоит из одного или более независимых *модулей*. В каком смысле модуль является *независимой* единицей языка Ассемблер, мы выясним несколько позже, когда будем изучать тему "Модульное программирование". Наши первые программы будут содержать всего один модуль, но позже будут рассмотрены и многомодульные программы.

Каждый модуль обычно содержит описание одного или нескольких *сегментов* памяти. Напомним, что в нашей архитектуре для работы программы каждая команда и каждое данное должны располагаться в каких-либо сегментах памяти. Как мы уже знаем, в младшей модели нашего семейства ЭВМ в каждый момент времени определены четыре *активных* (или *текущих*) сегмента памяти, на которые указывают соответствующие сегментные регистры CS, DS, SS и ES.

Таким образом, перед непосредственной работой с содержимым любого сегмента требуется установить на его начало определённый сегментный регистр, до этого нельзя ни писать в этот сегмент, ни читать из него. С другими сегментами, кроме этих четырёх текущих (если они есть в программе), работать в этот момент нельзя, при необходимости доступа к ним нужно менять (*перезагружать*) содержимое соответствующих сегментных регистров.

Стоит заметить, что сегменты могут перекрываться в памяти ЭВМ и даже полностью совпадать (накладываться друг на друга). Однако максимальный размер сегмента в младшей модели нашего семейства ЭВМ равен 64К, и, если сегменты будут перекрываться, то одновременно для работы будет доступно меньшее количество оперативной памяти. Заметим, что пересечение сегментов никак не влияет на логику работы центрального процессора.¹

В соответствии с принципом фон Неймана, мы имеем право размещать в любом из сегментов памяти как числа, так и команды. Но такой подход ведёт к плохому стилю программирования, программа перестаёт легко читаться и пониматься программистами. Будем поэтому стараться размещать команды программы в одном сегментах, а данные – в других. Весьма редко программисту будет выгодно размещать данные среди команд, один такой случай будет рассмотрен позже в нашем курсе.

На текущий сегмент команд должен указывать регистр CS, а на сегмент данных – регистр DS. Дело в том, что эти регистры *специализированные*. В частности, устройство управления может выбирать команды для выполнения *только* из сегмента, на который указывает регистр CS. Производить арифметические операции можно над числами из любого сегмента, однако в соответствии с принципом *умолчания* все переменные, если прямо не указано противное, сегментируются по регистру DS. Явное указание необходимости выбирать аргументы команды по другому сегментному регистру увеличивает длину команды на один байт (перед такой командой вставляется специальная однобайтная команда, которая называется *префиксом сегмента*).

Итак, модуль состоит из описаний сегментов. В сегментах находятся все команды и области памяти, используемые для хранения переменных. Вне сегментов могут располагаться только так называемые *директивы* языка Ассемблер, о которых мы будем говорить немного ниже. Пока лишь отметим, что чаще всего директивы не определяют в программе ни команд, ни переменных (поймите, что именно поэтому они и могут стоять *вне* сегментов).²

¹ Для особо любознательных студентов: для перекрывающихся сегментов можно ухитриться получить доступ к ячейкам одного сегмента через сегментный регистр, указывающий на другой сегмент. Мы, естественно, такими фокусами заниматься не будем.

² Исключением, например, является директива **include**, на место которой подставляется некоторый текстовый файл. Этот файл может содержать описание целого сегмента или же наборы фрагментов программ (так называемые макроопределения). С примером использования этой директивы мы вскоре познакомимся.

Описание каждого сегмента, в свою очередь, состоит из *предложений* (statement) языка Ассемблера. Каждое предложение языка Ассемблера занимает отдельную строчку программы, исключение из этого правила будет отмечено особо. Далее рассмотрим различные классы предложений Ассемблера.

7.3. Классификация предложений языка Ассемблер

- **Многострочные комментарии.** Это единственная конструкция Ассемблера, которая может занимать несколько строк текста программы. Будем для унификации терминов считать её неким частным типом предложения, хотя не все авторы учебников по Ассемблеру придерживаются этой точки зрения. Способ записи этих комментариев:

```
COMMENT *
    < строки – комментарии >
*
```

здесь символ * задаёт границы комментария, он не должен встречаться внутри самого комментария. При необходимости использовать символ * внутри комментария, надо вместо этого символа в качестве границ комментария выбрать какой-нибудь другой подходящий символ, например, &.

- **Команды.** Почти каждому такому предложению языка Ассемблера будет соответствовать одна команда на языке машины (в редких случаях получаются две "тесно связанных" команды). Как уже отмечалось, вне описания сегмента такое предложение встречаться не может.

- **Резервирование памяти.** Эти предложения отводят в том сегменте, где они записаны, области памяти для хранения переменных. Это некоторый аналог описания переменных языка Паскаль. Способ записи таких предложений надо посмотреть в учебнике [5], мы приведём лишь некоторые примеры с комментариями.

Предложение	Количество памяти
A db ?	1 байт
B dw ?	2 байта (слово)
C dd ?	4 байта (двойное слово)

В этих примерах описаны переменные с именами А, В и С разной длины, которые, как мы уже привыкли в языке Паскаль, не будут иметь конкретных начальных значений, что отмечено символом вопросительного знака. Однако, по принципу Фон Неймана ничто не мешает нам работать напрямую с одним или несколькими байтами, расположенными в любом месте памяти. Например, команда

```
mov ax, B+1
```

будет читать на регистр ax слово, *второй* байт которого располагается в конце переменной В, а *первый* – в начале переменной С (помним о "перевёрнутом" хранении слов в памяти!). Поэтому следует быть осторожными и не считать А, В и С отдельными, "независимыми" переменными в смысле языка Паскаль, это просто именованные области памяти. Разумеется, в понятно написанной программе эти области используются так, как они описаны с помощью присвоенных им имён.

Предложение

```
D dw 20 dup (?)
```

резервирует в сегменте 20 подряд расположенных слов с неопределёнными начальными значениями. Это можно назвать резервированием памяти под массив из 20 элементов, но при этом мы также не теряем возможности работать с произвольными байтами и словами из области памяти, зарезервированной под массив.

- **Директивы или команды Ассемблеру.** Эти предложения, как уже упоминалось, не порождают в машинной программе никакого кода, т.е. команд или переменных (редким исключением является директива **include**, о которой мы будем говорить при написании полных программ). Директивы используются программистом для того, чтобы давать программе Ассемблер определённые указания, управлять работой Ассемблера при компиляции (переводе) программы на язык машины. В качестве примера рассмотрим директивы объявления начала и конца описания сегмента с именем А:

```
A segment
```

```
...
```

A ends

Частным случаем директивы является и предложение-метка, она приписывает имя (метку) следующему за ней предложению. Так, в приведённом ниже примере метка `Next_Statement_Name` является именем следующего за ней предложения, таким образом, у этого предложения две метки:

```
Next_Statement_Name:
L:   mov ax, 2
```

• **Макрокоманды.** Этот класс предложений Ассемблера относится к *макросредствам* языка, и будут подробно изучаться далее в нашем курсе. Пока лишь скажем, что на место макрокоманды по определённым правилам подставляется некоторый набор (возможно и пустой) предложений Ассемблера.

Теперь рассмотрим структуру одного предложения. За редким исключением, каждое предложение может содержать от одного до четырёх *полей*: поле *метки*, поле *кода операции*, поле *операндов* и поле *комментария* (как обычно, квадратные скобки указывают на необязательность заключённой в них конструкции):

```
[<метка>[:]] КОП [<операнды>] [; комментарий]
```

Как видно, все поля предложения, кроме кода операции, являются необязательными и могут отсутствовать в конкретном предложении. Операнды, если их в предложении несколько, отделяются друг от друга запятыми (в *макрокоманде* операнды могут разделяться и пробелами). Если после метки стоит двоеточие, то это указание на то, что данное предложение может рассматриваться как *команда*, т.е. выбираться для исполнения в устройство управления.

В очень редких случаях предложения языка Ассемблера имеют другую структуру, например, *директива* присваивания значения *переменной периода генерации* (с этими переменными мы познакомимся при изучении макросредств языка):

```
K = K+1
```

Другим примером может служить строка-комментарий, такие строки начинаются с символа точки с запятой, перед которой могут стоять только символы пробелов:

```
; это строка-комментарий
```

7.4. Пример полной программы на Ассемблере

Прежде, чем написать нашу первую полную программу на Ассемблере, нам необходимо научиться выполнять операции ввода/вывода, без которых ни одна сколько-нибудь серьёзная программа обойтись не может. В самом языке машины, в отличие от языка нашей учебной машины УМ-3, нет команда ввода/вывода,¹ чтобы, например, ввести целое число, необходима достаточно большая *программа* на машинном языке.

Для организации ввода/вывода мы в наших примерах будем использовать макрокоманды из учебника [5]. Вместо каждой макрокоманды Ассемблер будет подставлять соответствующий этой макрокоманде набор команд и констант (этот набор, как мы узнаем позже, называется *макрорасширением* для макрокоманды).

Нам понадобятся следующие макрокоманды ввода/вывода.

- Макрокоманда вывода символа на экран

```
outch op1
```

где операнд `op1` может быть в формате `i8`, `r8` или `m8`. Значение операнда трактуется как код символа, этот символ выводится в текущую позицию экрана. Для задания кода символа удобно использовать символьную константу языка Ассемблер, например, `'A'`. Такая константа преобразуется программой Ассемблера именно в код этого символа. Например, `outch '*'` выведет символ звёздочки на место курсора.

- Макрокоманда ввода символа с клавиатуры

```
inch op1
```

где операнд `op1` может быть в формате `r8` или `m8`. Код введённого символа записывается в место памяти, определяемое операндом.

¹ Точнее, в машинном языке есть только операции обмена байтами между центральным процессором и периферийными устройствами компьютера, с этими операциями мы познакомимся позже.

- Макрокоманды вывода на экран целого значения

outint op1 [, op2]

outword op1 [, op2]

Здесь, как всегда, квадратные скобки говорят о том, что второй операнд может быть опущен. В качестве первого операнда `op1` можно использовать `i16`, `r16` или `m16`, а второго – `i8`, `r8` или `m8`. Действие макрокоманды `outint op1, op2` полностью эквивалентно процедуре вывода языка Паскаль `write(op1:op2)`, а действие макрокоманды с именем **outword** отличается только тем, что первый операнд *трактруется* как беззнаковое (неотрицательное) число.

- Макрокоманда ввода целого числа

inint op1

где операнд `op1` может иметь формат `r16` или `m16`, производит ввод с клавиатуры на место первого операнда целого значения из диапазона $-2^{15} \dots +2^{16}$. Особо отметим, что операнды форматов `r8` и `m8` недопустимы.

- Макрокоманда без параметров

newline

предназначена для перехода курсора к началу следующей строки экрана и эквивалентна вызову процедуры без параметров `writeln` языка Паскаль. Этого же эффекта можно достичь, если вывести на экран служебные символы с кодами 10 и 13, т.е. выполнить, например, макрокоманды

outch 10

outch 13

- Макрокоманда без параметров

flush

предназначена для очистки буфера ввода и эквивалентна вызову процедуры без параметров `readln` языка Паскаль.

- Макрокоманда вывода на экран строки текста

outstr

Эта макрокоманда выводит на экран строку текста из того сегмента, на который указывает сегментный регистр `DS`, причём адрес начала этой строки в сегменте должен находиться в регистре `DX`. Таким образом, физический адрес начала выводимого текста определяется по формуле

$$A_{\text{физ}} = (DS * 16 + DX) \bmod 2^{20}$$

Заданный таким образом адрес принято записывать в виде так называемой *адресной пары* `<DS, DX>`. В качестве признака конца выводимой строки символов должен быть задан символ `$` (он рассматривается как служебный признак конца и сам не выводится). Например, если в сегменте данных есть текст

Data **segment**

. . .

T **db** 'Текст для вывода на экран\$'

. . .

data **ends**

то для вывода этого текста на экран можно выполнить следующий фрагмент программы

. . .

mov DX, **offset** T; DX:=адрес T

outstr

. . .

Рассмотрим теперь пример простой *полной* программы на Ассемблере. Эта программа должна вводить значение целой переменной `A` и реализовывать оператор присваивания (в смысле языка Паскаль)

`X := (2*A - 241 div (A+B)2) mod 7`

где `B` – *параметр*, т.е. значение, которое не вводится, а задаётся в самой программе. Пусть `A`, `B` и `C` – *знаковые* целые величины, описанные в сегменте данных так:

A **dw** ?

B **db** -8; это *параметр*, заданный программистом

X **dw** ?

Вообще говоря, результат, заносимый в переменную *X* *короткий* (это остаток от деления на 7), однако мы выбрали для *X* формат слова, т.к. его надо выдавать в качестве результата, а макрокоманда `outint` может выводить только *длинные* целые числа.

Наша программа будет содержать три сегмента с именами `data`, `code` и `stack` и выглядеть следующим образом:

```
include io.asm
; вставить в программу файл с макроопределениями
; для макрокоманд ввода-вывода
data segment
A    dw ?
B    db -8
X    dw ?
Data ends
stack segment stack
    db 128 dup (?)
stack ends
code segment
    assume cs:code, ds:data, ss:stack
start:mov ax,data; это команда формата r16,i16
    mov ds,ax ; загрузка сегментного регистра DS
    inint A ; макрокоманда ввода целого числа
    mov bx,A ; bx := A
    mov al,B ; al := B
    cbw ; ax := длинное B
    add ax,bx ; ax := B+A=A+B
    add bx,bx ; bx := 2*A
    imul ax ; (dx,ax) := (A+B)2
    mov cx,ax ; cx := младшая часть (A+B)2
    mov ax,241
    cwd ; <dx,ax> := сверхдлинное 241
    idiv cx ; ax := 241 div (A+B)2, dx := 241 mod (A+B)2
    sub bx,ax ; bx := 2*A - 241 div (A+B)2
    mov ax,bx
    cwd
    mov bx,7
    idiv bx ; dx := (2*A - 241 div (A+B)2) mod 7
    mov X,dx
    outint X
    finish
code ends
end start
```

Прокомментируем текст нашей программы. Во-первых, заметим, что сегмент стека мы нигде явно не используем, однако он *необходим в любой* программе. Как мы узнаем далее из нашего курса, во время выполнения любой программы возможно автоматическое (без нашего ведома) переключение на выполнение некоторой другой программы, при этом используется сегмент стека. Подробно этот вопрос мы рассмотрим при изучении *прерываний*.

В начале сегмента кода расположена директива **assume**, она говорит программе Ассемблера, на какие сегменты будут указывать соответствующие сегментные регистры при выполнении команд, *обращающихся* к этим сегментам. Сама эта директива не меняет значения ни одного сегментного регистра, подробно про неё необходимо прочитать в учебнике [5].

Заметим, что сегментные регистры *SS* и *CS* должны быть загружены *перед* выполнением *самой первой* команды нашей программы. Ясно, что сама наша программа этого сделать не в состоянии, так как для этого необходимо выполнить хотя бы одну команду, что требует доступа к сегменту кода, и, в свою очередь, уже установленного на этот сегмент регистра *CS*. Получается замкнутый круг, и единственным решением будет попросить какую-то *другую* программу загрузить значения этих

регистров, *перед вызовом* нашей программы. Как мы потом увидим, эту операцию будет делать служебная программа, которая называется *загрузчиком*.

Первые две команды нашей программы загружают значение сегментного регистра DS, в младшей модели для этого необходимы именно *две* команды, так как одна команда имела бы несуществующий формат:

```
mov ds,data; формат SR,i16 такого формата нет!
```

Пусть, например, при счёте нашей программы сегмент данных будет располагаться, начиная с адреса 100000_{10} оперативной памяти. Тогда команда

```
mov ax,data
```

будет во время счёта иметь вид

```
mov ax,6250 ; 100000 div 16 = 6250
```

Макрокоманда

```
inint A; макрокоманда ввода целого числа
```

вводит значение целого числа в переменную A.

Далее начнём непосредственное вычисление правой части оператора присваивания. Задача усложняется тем, что величины A и B имеют разную длину и непосредственно складывать их нельзя. Приходится командами

```
mov al,B ; al := B
cbw ; ax := длинное B
```

преобразовать короткое целое B, которое сейчас находится на регистре al, в длинное целое на регистре ax. Далее вычисляется значение выражения $(A+B)^2$ и можно приступить к выполнению деления. Так как делитель является длинным целым числом (мы поместили его на регистр cx), то необходимо применить операцию *длинного* деления, для чего делимое (число 241 на регистре ax) командой

```
cwd
```

преобразуем в сверхдлинное целое и помещаем на два регистра (dx, ax). Вот теперь всё готово для команды целочисленного деления

```
idiv cx; ax:= 241 div (A+B)2 , dx:= 241 mod (A+B)2
```

Далее мы присваиваем остаток от деления (он в регистре dx) переменной X и выводим значение этой переменной по макрокоманде

```
outint X
```

которая эквивалентна процедуре WriteLn(X) языка Паскаль. Последним предложением в сегменте кода является макрокоманда

```
finish
```

Эта макрокоманда заканчивает выполнение нашей программы, она эквивалентна выходу программы на Паскале на конечный **end**.

И, наконец, директива

```
end start
```

заканчивает описание всего модуля на Ассемблере. Обратите внимание на параметр этой директивы – метку start. Она указывает *входную точку* программы, т.е. её первую выполняемую команду программы.

Сделаем теперь важные замечания к нашей программе. Во-первых, мы не проверяли, что команды сложения и вычитания дают правильный результат (для этого, как мы знаем, после выполнения этих команд нам было бы необходимо проверить флаг переполнения OF, т.к. наши числа мы считаем *знаковыми*). Во-вторых, команда длинного умножения располагает свой результат в двух регистрах (dx, ax), а в нашей программе мы брали результат произведения только из регистра ax, предполагая, что на регистре dx находятся только *незначущие* цифры произведения. По-хорошему надо было бы проверить, что в dx содержатся только нулевые биты, если $ax \geq 0$, и только двоичные “1”, если

$ax < 0$. Другими словами, знак числа в регистре dx должен совпадать со знаком числа в регистре ax, для *знаковых* чисел это и есть признак того, что в регистре dx содержится *незначущая* часть произведения. И, наконец, мы не проверили, что не производим деления на ноль (в нашем случае что $A \neq 8$). В наших *учебных* программах мы иногда не будем делать таких проверок, но в “настоящих”

программах, которые Вы будете создавать на компьютерах и предъявлять преподавателям, эти проверки являются обязательными.

Продолжая знакомство с языком Ассемблера, решим следующую задачу. Напишем фрагмент программы, в котором увеличивается на единицу целое число, расположенное в 234567_{10} байте оперативной памяти. Мы уже знаем, что запись в любой байт памяти возможна только тогда, когда этот байт расположен в одном из четырёх текущих сегментах. Сделаем, например, так, чтобы наш байт располагался в сегменте данных. Главное здесь – не путать сегменты данных, которые мы описываем в программе на Ассемблере, с активными сегментами, на начала которых установлены сегментные регистры. Описываемые в программе сегменты обычно размещаются загрузчиком на свободных участках оперативной памяти, и, как правило, при написании текста программы неизвестно их будущего месторасположение.¹ Однако ничто не мешает нам любой участок оперативной памяти сделать сегментом, установив на него какой-либо сегментный регистр. Так мы и сделаем для решения нашей задачи, установив сегментный регистр DS на начало ближайшего сегмента, в котором будет находиться наш байт с адресом 234567_{10} . Так как в сегментный регистр загружается адрес начала сегмента, делённый на 16, то нужное нам значение сегментного регистра можно вычислить по формуле: $DS := 234567 \text{ div } 16 = 14660$. При этом адрес A нашего байта в сегменте (его смещение от начала сегмента) вычисляется по формуле: $A := 234567 \text{ mod } 16 = 7$. Таким образом, для решения нашей задачи можно предложить следующий фрагмент программы:

```
mov ax,14660
mov ds,ax; Начало сегмента
mov bx,7; Смещение
inc byte ptr [bx]
```

Теперь, после изучения арифметических операций, перейдём к рассмотрению команд *переходов*, которые понадобятся нам для программирования условных операторов и циклов. После изучения нашего курса мы должны уметь отображать на Ассемблер любые конструкции языка Паскаль.

7.5. Переходы

В большинстве современных компьютеров реализован принцип *последовательного выполнения команд*. Это значит, что после выполнения текущей команды счётчик адреса будет указывать на следующую (ближайшую с большим адресом) команду в оперативной памяти.² Изменить последовательное выполнение команд можно с помощью *переходов*, при этом следующая команда может быть расположена в другом месте оперативной памяти. Ясно, что без переходов компьютеры функционировать не могут: скорость центрального процессора так велика, что он очень быстро может по одному разу выполнить все команда в оперативной памяти.

Понимание переходов очень важно при изучении архитектуры ЭВМ, они позволяют уяснить логику работы центрального процессора. Все переходы можно разделить на два вида.

- Переходы, вызванные выполнением центральным процессором специальных *команд переходов*.
- Переходы, которые *автоматически* выполняет центральный процессор при наступлении определённых событий в центральной части компьютера или в его периферийных устройствах (устройствах ввода/вывода).

Начнём последовательное рассмотрение переходов для компьютеров нашей архитектуры. Напомним, что физический адрес начала следующей выполняемой команды зависит от значений двух регистров: сегментного регистра CS и счётчика адреса IP и вычисляется по формуле:

$$A_{\text{физ}} := (CS * 16 + IP) \text{ mod } 2^{20}$$

Следовательно, для осуществления перехода необходимо в один или оба эти регистра занести новые значения. Отсюда будем выводить первую классификацию переходов: будем называть переход

¹ Можно дать загрузчику явное указание на размещение конкретного сегмента с заданного адреса оперативной памяти (мы изучим это позднее в нашем курсе), но это редко когда нужно программисту. Наоборот, лучше писать программу, которая будет правильно работать при *любом* размещении её сегментов в оперативной памяти.

² При достижении в программе конца оперативной памяти (или конца сегмента при сегментной организации памяти) обычно выполняется команда, расположенная в начале памяти или в начале этого сегмента.

близким переходом, если при этом меняется *только* значение регистра IP, если же при переходе меняются значения обоих регистров, то такой переход будем называть *дальним* (межсегментным) переходом.¹

Следующей основой для классификации переходов будет служить способ изменения значения регистра. При *относительном* переходе происходит *знаковое* сложение содержимого регистра с некоторой константой, например,

$$IP := (IP + Const)_{\text{mod } 2^{16}}$$

При *абсолютном* переходе происходит просто присваивание соответствующему регистру нового значения, например,

$$CS := Const$$

Опять же из соображений ценности практического использования в программировании, для сегментного регистра CS реализован только абсолютный переход, в то время как для счётчика адреса IP возможен как абсолютный, так и относительный переходы.

Далее будем классифицировать *относительные* переходы по величине той константы, которая прибавляется к значению счётчика адреса IP: при *коротком* переходе величина этой *знаковой* константы (напомним, что мы обозначаем её *i8*) не превышает по размеру одного байта (т.е. лежит в диапазоне от -128 до $+127$):

$$IP := (IP + i8)_{\text{mod } 2^{16}},$$

а при *длинном* переходе эта константа имеет размер слова (двух байт):

$$IP := (IP + i16)_{\text{mod } 2^{16}}$$

Кроме того, величина, используемая при абсолютном переходе для задания нового значения какого-либо из этих регистров, может быть *прямой* и *косвенной*. Прямая величина является просто числом (в нашей терминологии это *непосредственный* адрес), а косвенная – является *адресом* некоторой области памяти, откуда и будет извлекаться необходимое число, например,

$$IP := [m16]$$

Здесь на регистр IP заносится число, содержащееся в двух байтах памяти по адресу *m16*, т.е. это *близкий длинный абсолютный косвенный* переход.

Таким образом, каждый переход можно классифицировать по его свойствам: близкий – дальний, относительный – абсолютный, короткий – длинный, прямой – косвенный. Разумеется, не все из этих переходов реализуются в компьютере, так, мы уже знаем, что короткими или длинными бывают только относительные переходы, а относительные переходы бывают только прямыми.

7.6. Команды переходов

Изучим сначала команды переходов. Эти команды предназначены *только* для передачи управления в другое место программы, они *не меняют* никаких флагов.

7.6.1. Команды безусловного перехода

Рассмотрим сначала команды безусловного перехода, которые всегда передают управление в указанную в них точку программы. На языке Ассемблера все эти команды записываются в виде

jmp op1

Здесь op1 может иметь следующие форматы:

op1	Способ выполнения	Вид перехода
i8	$IP := (IP + i8)_{\text{mod } 2^{16}}$	Близкий относительный короткий
i16	$IP := (IP + i16)_{\text{mod } 2^{16}}$	Близкий относительный длинный
r16	$IP := [r16]$	Близкий абсолютный косвенный
m16	$IP := [m16]$	Близкий абсолютный косвенный
m32	$IP := [m32], CS := [m32+2]$	Дальний абсолютный косвенный
seg:off	$IP := off, CS := seg$	Дальний абсолютный прямой

Здесь *seg:off* – это мнемоническое обозначение двух операндов в формате *i16*, разделённых двоеточием. Как видно из этой таблицы, многие потенциально возможные виды безусловного

¹ В принципе переход возможен и при изменении значения только одного регистра CS, однако в практике программирования такие переходы практически не имеют смысла и не реализуются.

перехода (например, близкие абсолютные прямые, близкие абсолютные короткие и др.) не реализованы в нашей архитектуре. Это сделано исключительно для упрощения центрального процессора (не нужно реализовывать в нём эти команды) и для уменьшения размера программы (чтобы длина поля кода операции в командах не была слишком большой).

Рассмотрим теперь, как на языке Ассемблера задаются эти операнды команд безусловного перехода. Для указания близкого относительного перехода в команде обычно записывается метка команды, на которую необходимо выполнить переход, например:

```
jmp L; Перейти на команду, помеченную меткой L
```

Напомним, что вслед за меткой команды, в отличие от метки области памяти, ставится двоеточие. Так как значением метки является её смещение в том сегменте, где эта метка описана, то программе Ассемблера приходится самой вычислять необходимое смещение *i8* или *i16*, которое необходимо записать на место операнда в команде на машинном языке ¹, например:

```
L: add   bx,bx ;
      . . .
      . . .
      . . .
jmp   L; L = i8 или i16
```

← i8 или i16 (со знаком !)

Здесь формат операнда (*i8* или *i16*) выбирается программой Ассемблера автоматически, в зависимости от расстояния в программе между командой перехода и меткой. Если же метка *L* располагается в программе *после* команды перехода, то Ассемблер, ещё не зная истинного расстояния до этой метки, "на всякий случай" заменяет эту метку на операнд размера *i16*. Поэтому для тех программистов, которые знают, что смещение должно быть формата *i8* и хотят сэкономить один байт памяти, Ассемблер предоставляет возможность задать размер операнда в явном виде:

```
jmp short L
```

Ясно, что это нужно делать только при острой нехватке оперативной памяти для программы. ² Для явного указания дальнего перехода программист должен использовать оператор **far ptr**, например:

```
jmp far ptr L
```

Приведём фрагмент программы с различными видами командам безусловного перехода, в этом фрагменте описаны два кодовых сегмента (для иллюстрации дальних переходов) и один сегмент данных:

```
data segment
A1 dw L2;           Смещение команды с меткой L2 в своём сегменте
A2 dd Code1:L1;    Это seg:off
      . . .
data ends
code1 segment
      . . .
L1: mov ax,bx
      . . .
code1 ends
code2 segment
      assume cs:code2, ds:data
start:mov ax,data
      mov ds,ax ;      загрузка сегментного регистра DS
L2: jmp far ptr L1;  дальний прямой абсолютный переход, op1=seg:off
      . . .
      jmp L1;        ошибка т.к. без far ptr
      jmp L2;        близкий относительный переход, op1=i8 или i16
      jmp A1;        близкий абсолютный косвенный переход, op1=m16
```

¹ Необходимо учитывать, что в момент выполнения команды перехода счётчик адреса IP уже указывает на *следующую* команду, что, конечно, существенно при вычислении величины смещения. Но, так как эту работу выполняет программа Ассемблера, мы на эту особенность не будем обращать внимания.

² Например, при написании *встроенных* программ для управления различными устройствами (стиральными машинами, видеоманитофонами и т.д.), либо программ для автоматических космических аппаратов, где память относительно небольшого объёма, т.к. особая, способная выдерживать космическое излучение.

```

jmp  A2;    дальний абсолютный косвенный переход, op1=m32
jmp  bx;    близкий абсолютный косвенный переход, op1=r16
jmp  [bx];  ошибка, нет выбора: op1=m16 или m32 ?
mov  bx, A2
jmp  dword ptr [bx]; дальний абсолютный косвенный переход op1=m32
. . .
code2 ends

```

Отметим одно важное преимущество относительных переходов перед абсолютными. Значение `i8` или `i16` в команде относительного перехода зависит только от расстояния в байтах между командой перехода и точкой, в которую производится переход. При любом изменении в сегменте кода *вне* этого диапазона команд значения `i8` или `i16` не меняются.

Как видим, архитектура нашего компьютера обеспечивает большой спектр команд безусловного перехода. Напомним, что в нашей учебной машине УМ-3 была только одна команда безусловного перехода. На этом мы закончим наше краткое рассмотрение команд безусловного перехода. Напомним, что для усвоения материала по курсу Вам необходимо изучить соответствующий раздел учебника по Ассемблеру.

7.6.2. Команды условного перехода

Все команды условного перехода выполняются по схеме

```
if <условие перехода> then goto L
```

и производят *близкий короткий относительный* переход, если выполнено некоторое условие перехода, в противном случае продолжается последовательное выполнение команд программы. На Паскале условие перехода чаще всего задают в виде условного оператора

```
if op1 <отношение> op2 then goto L
```

где отношение – один из знаков операции отношения = (равно), <> (не равно), > (больше), < (меньше), <= (меньше или равно), >= (больше или равно). Если обозначить $rez = op1 - op2$, то оператор условного перехода можно записать в эквивалентном виде сравнения с нулём

```
if rez <отношение> 0 then goto L
```

Все машинные команды условного перехода, кроме одной, вычисляют условие перехода, анализируя один, два или три флага из регистра флагов, и лишь одна команда условного перехода вычисляет условие перехода, анализируя значение регистра `CX`. Команда условного перехода в языке Ассемблер имеет вид

```
j<мнемоника перехода> i8; IP := (IP + i8) mod 216
```

Мнемоника перехода (это от одной до трёх букв) связана со значением анализируемых флагов (или регистра `CX`), либо со *способом формирования* этих флагов. Чаще всего программисты формируют флаги, проверяя отношение между двумя операндами `op1 <отношение> op2`, для чего выполняется команда *вычитания* или команда *сравнения*. Команда сравнения имеет мнемонический код операции `cmp` и такой же формат, как и команда вычитания:

```
cmp op1, op2
```

Она и выполняется точно так же, как команда вычитания за исключением того, что разность *не записывается* на место первого операнда. Таким образом, единственным результатом команды сравнения является формирование флагов, которые устанавливаются так же, как и при выполнении команды вычитания. Вспомним, что программист может *трактовать* результат вычитания (сравнения) как производимый над *знаковыми* или же *беззнаковыми* числами. Как мы уже знаем, от этой трактовки зависит и то, будет ли один операнд больше другого или же нет. Так, например, рассмотрим два коротких целых числа `0FFh` и `01h`. Как *знаковые* числа $0FFh = -1 < 01h = 1$, а как *беззнаковые* числа $0FFh = 255 > 01h = 1$.

Исходя из этого, принята следующая терминология: при сравнении *знаковых* целых чисел первый операнд может быть *больше* (greater) или *меньше* (less) второго операнда. При сравнении же *беззнаковых* чисел будем говорить, что первый операнд *выше* (above) или *ниже* (below) второго. Ясно, что действию "выполнить переход, если первый операнд больше второго" будут соответствовать разные машинные команды, если трактовать операнды как знаковые или же беззнаковые целые числа. Это учитывается в различных мнемониках этих команд.

Ниже в Таблице 7.1 приведены мнемоники команд условного перехода. Некоторые команды имеют разную мнемонику, но выполняются одинаково (переводятся программой Ассемблера в одну и ту же машинную команду), такие команды указаны в одной строке таблицы.

Таблица 7.1. Мнемоника команд условного перехода

КОП	Условие перехода	
	Логическое условие перехода	Результат (rez) команды вычитания или соотношение операндов op1 и op2 команды сравнения
je jz	ZF = 1	Rez = 0 или op1 = op2 (результат = 0, операнды равны)
jne jnz	ZF = 0	rez <> 0 или op1 <> op2 Результат <> 0, операнды не равны
jg jnle	(SF=OF) and (ZF=0)	rez > 0 или op1 > op2 Знаковый результат > 0, op1 больше op2
jge jnl	SF = OF	rez >= 0 или op1 >= op2 Знаковый результат >= 0, т.е. op1 больше или равен (не меньше) op2
jl jnge	SF <> OF	rez < 0 или op1 < op2 Знаковый результат < 0, т.е. op1 меньше (не больше или равен) op2
jle jng	(SF<>OF) or (ZF=1)	rez <= 0 или op1 <= op2 Знаковый результат <= 0, т.е. op1 меньше или равен (не больше) op2
ja jnbe	(CF=0) and (ZF=0)	rez > 0 или op1 > op2 Беззнаковый результат > 0, т.е. op1 выше (не ниже или равен) op2
jae jnb jnc	CF = 0	rez >= 0 или op1 >= op2 Беззнаковый результат >= 0, т.е. op1 выше или равен (не ниже) op2
jb jnae jc	CF = 1	rez < 0 или op1 < op2 Беззнаковый результат < 0, т.е. op1 ниже (не выше или равен) op2
jbe jna	(CF=1) or (ZF=1)	rez >= 0 или op1 >= op2 Беззнаковый результат >= 0, т.е. op1 ниже или равен (не выше) op2
js	SF = 1	Знаковый бит результата (7-й или 15-ый, в зависимости от размера) равен единице
jns	SF = 0	Знаковый бит результата (7-й или 15-ый, в зависимости от размера) равен нулю
jo	OF = 1	Флаг переполнения равен единице
jno	OF = 0	Флаг переполнения равен нулю
jp jpe	PF = 1	Флаг чётности ¹ равен единице
jnp jpo	PF = 0	Флаг чётности равен нулю
jcxz	CX = 0	Значение регистра CX равно нулю

В качестве примера рассмотрим, почему условному переходу `jl/jnge` соответствует логическое условие перехода `SF<>OF`. При выполнении команды сравнения `cmp op1, op2` или команды вычитания `sub op1, op2` нас будет интересовать трактовка операндов как знаковых целых чисел, поэтому возможны два случая, когда первый операнд меньше второго. Во-первых, если при выполнении операции вычитания `op1-op2` результат получился правильным, т.е. не было переполнения (`OF=0`), то бит знака у правильного результата равен единице (`SF=1`). Во-вторых, при вычитании мог получиться неправильный результат, т.е. было переполнение (`OF=0`), но в этом случае знаковый бит результата будет неправильным, т.е. равным нулю. Видно, что в обоих случаях эти два

¹ Флаг чётности равен единице, если в восьми младших битах результата содержится чётное число двоичных единиц. Мы не будем работать с этим флагом.

флага не равны друг другу, т.е. должно выполняться условие $SF \neq OF$, что и указано в нашей таблице. Для тренировки разберите правила формирования и других условий переходов.

Как видим, команд условного перехода достаточно много, поэтому понятно, почему для них реализован только один формат – близкий короткий относительный переход. Реализация других форматов команд условного перехода привела бы к резкому увеличению числа команд в языке машины и, как следствие, к усложнению центрального процессора и росту объёма программ. В то же время наличие только одного формата команд условного перехода может приводить к плохому стилю программирования. Пусть, например, надо реализовать на Ассемблере условный оператор языка Паскаль

```
if X>Y then goto L;
```

Соответствующий фрагмент на языке Ассемблера, реализующий этот оператор для *знаковых* X, Y

```
mov ax,X
cmp ax,Y
jg L
. . .
```

L:

может быть неверным, если расстояние между меткой и командой условного перехода велико (не помещается в байт). В таком случае придётся использовать такой фрагмент на Ассемблере со вспомогательной меткой L1 и вспомогательной командой безусловного перехода:

```
mov ax,X
cmp ax,Y
jle L1
jmp L
```

L1:

. . .

L:

Таким образом, на самом деле мы *вынуждены* реализовывать такой фрагмент программы на языке Паскаль:

```
if X<=Y then goto L1; goto L; L1::;
```

Это, конечно, по необходимости, прививает плохой стиль программирования.

В качестве примера использования команд условного перехода рассмотрим программу, которая вводит знаковое число A в формате слова и вычисляет значение X по формуле

$$X := \begin{cases} (A+1) * (A-1), & \text{при } A > 2 \\ 4, & \text{при } A = 2 \\ (A+1) \bmod 7, & \text{при } A < 2 \end{cases}$$

```
include io.asm
; файл с макроопределениями для макрокоманд ввода-вывода
data segment
A dw ?
X dw ?
Diagn db 'Ошибка - большое значение!$'
Data ends
Stack segment stack
db 128 dup (?)
stack ends
code segment
assume cs:code, ds:data, ss:stack
start:mov ax,data; это команда формата r16,i16
mov ds,ax ; загрузка сегментного регистра DS
inint A ; ввод целого числа
mov ax,A ; ax := A
mov bx,ax ; bx := A
inc ax ; ax := A+1
jo Error
cmp bx,2 ; Сравнение A и 2
jle L1 ; Вниз по первой ветви вычисления X
dec bx ; bx := A-1
```

```

    jo    Error
    imul  bx    ; (dx,ax):=(A+1)*(A-1)
    jo    Error ; Произведение (A+1)*(A-1) не помещается в ax
L:    mov  X,ax ; Результат берётся только из ax
    outint X;    Вывод результата
    newline
    finish
L1:   jl    L2;    Вниз по второй ветви вычисления X
    mov  ax,4
    jmp  L;    На вывод результата
L2:   mov  bx,7;    Третья ветвь вычисления X
    cwd          ; (dx,ax):= длинное (A+1) - иначе нельзя!
    idiv bx;    dx:=(A+1) mod 7, ax:=(A+1) div 7
    mov  ax,dx
    jmp  L;    На вывод результата
Error:mov  dx,offset Diagn
    outstr
    newline
    finish
code  ends
end  start

```

В нашей программе мы сначала закодировали вычисление по первой ветви нашего алгоритма, затем по второй и, наконец, по третьей. Программист, однако, может выбрать и другую последовательность кодирования ветвей, это не влияет на суть дела. В нашей программе предусмотрена выдача аварийной диагностики, если результаты операций сложения $(A+1)$, вычитания $(A-1)$ или произведения $(A+1) * (A-1)$ слишком велики и не помещаются в одно слово.

Для увеличения и уменьшения операнда на единицу мы использовали команды

inc op1 и **dec** op1

Здесь op1 может иметь формат r8, r16, m8 и m16. Например, команда **inc ax** эквивалентна команде **add ax,1**, но *не меняет* флага CF. Таким образом, после этих команд нельзя проверить флаг переполнения, чтобы определить, правильно ли выполнены такие операции над *беззнаковыми числами*.

Обратите также внимание, что мы использовали команду *длинного* деления, попытка использовать здесь короткое деление, например

```

L2:   mov  bh,7;    Третья ветвь вычисления X
    idiv bh;    ah:=(A+1) mod 7, al:=(A+1) div 7

```

может привести к ошибке. Здесь остаток от деления $(A+1)$ на число 7 всегда поместится в регистр ah, однако *частное* $(A+1) \text{ div } 7$ может *не поместиться* в регистр al (пусть $A=27999$, тогда $(A+1) \text{ div } 7 = 4000$ – не поместится в регистр al).

При использовании команд условного перехода мы предполагали, что расстояние от точки перехода до нужной метки небольшое (формата i8), если это не так, то программа Ассемблера выдаст нам соответствующую диагностику об ошибке и нам придётся использовать "плохой стиль программирования", как объяснялось выше. В нашей программе это может случиться только тогда, когда суммарный размер кода, подставляемого вместо макрокоманд **outint** и **finish**, будет больше 128 байт (обязательно понять это!).

7.6.3. Команды цикла

Для организации циклов на Ассемблере вполне можно использовать команды условного перехода. Например, цикл языка Паскаль с предусловием **while X<0 do S;** можно реализовать в виде следующего фрагмента на Ассемблере

```

L:    cmp  X,0;    Сравнить X с нулём
    jge  L1
;    Здесь будет оператор S
    jmp  L
L1:   . . .

```


Оператор цикла с постусловием `repeat S1; S2; . . . Sk until X<0;` можно реализовать в виде фрагмента на Ассемблере

```
L:      ;      S1
        ;      S2
        . . .
        ;      Sk
        cmp  X,0;   Сравнить X с нулём
        jge  L
        . . .
```

В этих примерах мы считаем, что тело цикла по длине не превышает примерно 120 байт (это 30–40 машинных команд). Как видим, цикл с постусловием требует для своей реализации на одну команду меньше, чем цикл с предусловием.

Как мы знаем, если число повторений выполнения тела цикла известно до начала исполнения этого цикла, то в языке Паскаль наиболее естественно было использовать *цикл с параметром*. Для организации цикла с параметром в Ассемблере можно использовать специальные *команды цикла*. Команды цикла, по сути, тоже являются командами условного перехода и, как следствие, реализуют только *близкий короткий относительный* переход. Команда цикла

```
loop L;   Метка L заменится на операнд i8
```

использует неявный операнд – регистр CX и её выполнение может быть так описано с использованием Паскаля:

```
Dec(CX); {Это часть команды loop, поэтому флаги не меняются!}
if CX<>0 then goto L;
```

Как видим, регистр CX (который так и называется регистром счётчиком цикла – loop counter), используется этой командой именно как параметр цикла. Лучше всего эта команда цикла подходит для реализации цикла с параметром языка Паскаль вида

```
for CX:=N downto 1 do S;
```

Этот оператор можно *эффективно* реализовать таким фрагментом на Ассемблере:

```
mov   CX,N
jcxz  L1
L:     . . .; Тело цикла -
        . . .; оператор S
        loop L
L1:    . . .
```

Обратите внимание, так как цикл с параметром языка Паскаль по существу является циклом с предусловием, то до начала его выполнения проверяется исчерпание значений для параметра цикла с помощью команды условного перехода `jcxz L1`, которая именно для этого и была введена в язык машины. Ещё раз напоминаем, что команды циклов *не меняют* флагов.

Описанная выше команда цикла выполняет тело цикла ровно N раз, где N – *беззнаковое* число, занесённое в регистр-счётчик цикла CX перед началом цикла. К сожалению, никакой другой регистр *нельзя* использовать для этой цели (т.к. это неявный параметр команды цикла). Кроме того, в приведённом выше примере реализации цикла тело этого не может быть слишком большим, иначе команда `loop L` *не сможет* передать управление на метку L.

В качестве примера использования команды цикла решим следующую задачу. Требуется ввести *беззнаковое* число N≤500, затем ввести N *знаковых* целых чисел и вывести сумму тех из них, которые принадлежат диапазону –2000..5000. Можно предложить следующее решение этой задачи.

```
include io.asm
; файл с макроопределениями для макрокоманд ввода-вывода
data segment
N     dw   ?
S     dw   0; Начальное значение суммы = 0
T1    db   'Введите N<=500 $'
T2    db   'Ошибка - большое N!$'
T3    db   'Вводите целые числа',10,13,'$'
T4    db   'Ошибка - большая сумма!$'
```

```

data ends
stack segment stack
dw 64 dup (?)
stack ends
code segment
assume cs:code,ds:data,ss:stack
start:mov ax,data
mov ds,ax
mov dx, offset T1; Приглашение к вводу
outstr
inint N
cmp N,500
jbe L1
mov dx, offset T2; Диагностика от ошибке
Err:  outstr
      newline
      finish
L1:   mov cx,N; Счётчик цикла
      jcxz Pech; На печать результата
      mov dx,offset T3; Приглашение к вводу
      ostr
      newline
L2:   inint ax; Ввод очередного числа
      cmp ax,-2000
      jl L3
      cmp ax,5000
      jg L3; Проверка диапазона
      add S,ax; Суммирование
      jno L3; Проверка на переполнение S
      mov dx,offset T4
      jmp Err
L3:   loop L2
Pech: outch 'S'
      outch '='
      outint S
      newline
      finish
code ends
end start

```

В качестве ещё одного примера рассмотрим использование циклов при обработке массивов. Пусть необходимо составить программу для решения следующей задачи. Задана константа $N=20000$, надо ввести массивы X и Y по N *беззнаковых* чисел в каждом массиве и вычислить выражение

$$S := \sum_{i=1}^N X[i] * Y[N - i + 1]$$

Для простоты будем предполагать, что каждое из произведений и вся сумма имеют формат **dw** (помещаются в слово). Ниже приведена программа, решающая эту задачу.

```

include io.asm
N equ 20000; Аналог Const N=20000; Паскаля
data1 segment
T1 db 'Вводите числа массива $'
T2 db 'Сумма = $'
T3 db 'Ошибка - большое значение!',10,13,'$'
S dw 0; искомая сумма
X dw N dup (?); 2*N байт
data1 ends
data2 segment
Y dw N dup (?); 2*N байт

```

```

data2 ends
st      segment stack
        dw      64 dup(?)
st      ends
code    segment
        assume cs:code,ds:data1,es:date2,ss:st
begin_of_program:
        mov     ax,data1
        mov     ds,ax;      ds - на начало data1
        mov     ax,data2
        mov     es,ax;      es - на начало data2
        mov     dx, offset T1; Приглашение к вводу
        outstr
        outch  'X'
        newline
        mov     cx,N; счётчик цикла
        mov     bx,0; индекс массива
L1:     inint  X[bx];ввод очередного элемента X[i]
        add     bx,2; увеличение индекса, это i:=i+1
        loop   L1
        outstr; Приглашение к вводу
        outch  'Y'
        newline
        mov     cx,N; счётчик цикла
        mov     bx,0; индекс массива
L2:     inint  ax
        mov     Y[bx],ax; ввод очередного элемента es:Y[bx]
        add     bx,2; увеличение индекса
        loop   L2
        mov     bx,offset X; указатель на X[1]
        mov     si,offset Y+2*N-2; указатель на Y[N]
L3:     mov     ax,[bx]; первый сомножитель
        mul    word ptr es:[si]; умножение на Y[N-i+1]
        jc     Err; большое произведение
        add     S,ax
        jc     Err; большая сумма
        add     bx,type X; это bx:=bx+2
        sub     si,2; это i:=i-1
        loop   L3; цикл суммирования
        mov     dx, offset T2
        outstr
        outword S
        newline
        finish
Err:    mov     dx,T3
        outstr
        finish
code    ends
        end    begin_of_program

```

Подробно прокомментируем эту программу. Количество элементов массивов мы задали, используя директиву эквивалентности `N equ 20000`, это есть указание программе Ассемблера о том, что всюду в программе, где встретится имя N, надо подставить вместо него операнд этой директивы – число 20000. Таким образом, это почти полный аналог описания константы в языке Паскаль.¹ Под каждый из массивов директива `dw` зарезервирует $2 * N$ байт памяти.

¹ Если не принимать во внимание то, что константа в Паскале имеет *тип*, это позволяет контролировать её использование, а в Ассемблере это просто указание о *текстовой подстановке* вместо имени операнда директивы эквивалентности.

Заметим теперь, что оба массива *не поместятся* в один сегмент данных (в сегменте не более примерно 32000 слов, а у нас в сумме 40000 слов), поэтому массив X мы размещаем в сегменте data1, а массив Y – в сегменте data2. Директива **assume** говорит, что на начала этих сегментов будут соответственно указывать регистры ds и es, что мы и обеспечили в самом начале программы. При вводе массивов мы использовали индексный регистр bx, в котором находится *смещение* текущего элемента массива от начала этого массива.

При вводе массива Y мы для *учебных* целей вместо предложения

```
L2:  inint Y[bx]; ввод очередного элемента
```

записали два предложения

```
L2:  inint ax
      mov  Y[bx], ax; ввод очередного элемента
```

Это мы сделали, чтобы подчеркнуть: при доступе к элементам массива Y Ассемблер учитывает то, что имя Y описано в сегменте data2 и *автоматически* (используя информацию из директивы **assume**) поставит перед командой `mov Y[bx], ax` специальную однобайтную команду `es:`. Эту команду называют *префиксом программного сегмента*, так что на языке машины у нас будут две последовательные, тесно связанные команды:

```
es:  mov Y[bx], ax
```

В цикле суммирования произведений для доступа к элементам массивов мы использовали другой приём, чем при вводе – регистры-указатели bx и si, в этих регистрах находятся *адреса* очередных элементов массивов. Напомним, что адрес – это *смещение* элемента относительно *начала сегмента* (в отличие от индекса элемента – это смещение от *начала массива*).

При записи команды умножение

```
mul  word ptr es:[si]; умножение на Y[N-i+1]
```

мы вынуждены *явно* задать размер второго сомножителя и записать префикс программного сегмента es:, так как по виду операнда [si] Ассемблер не может сам "догадаться", что это элемент массива Y *размером в слово* и *из сегмента data2*.

В команде

```
add  bx, type X; это bx:=bx+2
```

для задания размера элемента массива мы использовали оператор **type**. Параметром этого оператора является имя из нашей программы, значением оператора `type <ИМЯ>` является целое число – *тип* данного имени. Для имён областей памяти это длина этой области в байтах (для массива это почти всегда длина одного элемента), для меток команд это отрицательное число -1, если метка расположена в том же сегменте, что и оператор **type**, или отрицательное число -2 для меток из других сегментов. Все остальные имена имеют тип ноль.

Вы, наверное, уже почувствовали, что программирование на Ассемблере сильно отличается от программирования на языке высокого уровня (например, на Паскале). Чтобы подчеркнуть это различие, рассмотрим пример задачи, связанной с обработкой матрицы, и решим её на Паскале и на Ассемблере.

Пусть дана прямоугольная матрица целых чисел и надо найти сумму элементов, которые расположены в строках, начинающихся с отрицательного значения. Для решения этой задачи на Паскале можно предложить следующий фрагмент программы

```
Const N=20; M=30;
Var   X: array[1..N,1..M] of integer;
       Sum,i,j: integer;
       . . .
       { Ввод матрицы X }
       Sum:=0;
       for i:=1 to N do
           if X[i,1]<0 then
               for j:=1 to M do Sum:=Sum+X[i,j];
```

Сначала обратим внимание на то, что переменные i и j несут в программе на Паскале двойную нагрузку: это одновременно и счётчики циклов, и индексы элементов массива. Такое совмещение функций упрощает понимание программы и делает её очень компактной по внешнему виду, но не проходит даром: чтобы по индексам элемента массива вычислить его адрес в сегменте, приходится выполнить достаточно сложные действия. Например, адрес элемента X[i, j] приходится вычислять так:

$$\text{Адрес}(X[i, j]) = \text{Адрес}(X[1, 1]) + 2 * M * (i - 1) + 2 * (j - 1)$$

Эту формулу легко понять, учитывая, что матрица хранится в памяти по строкам (сначала первая строка, затем вторая и т.д.), и каждая строка имеет длину $2 * M$ байт. Буквальное вычисление адресом элементов по приведённой выше формуле (а именно так чаще всего и делает Паскаль-машина) приводит к весьма неэффективной программе. При программировании на Ассемблере лучше всего разделить функции счётчика цикла и индекса элементов. В качестве счётчика лучше всего использовать регистр `cx` (он и специализирован для этой цели), а адреса лучше хранить в индексных регистрах (`bx`, `si` и `di`). Исходя из этих соображений, можно так переписать программу на Паскале, предвидя её будущий перенос на Ассемблер.

```

Const N=20; M=30;
Var X: array[1..N,1..M] of integer;
    Sum,cx,oldcx: integer; bx: ↑integer;
    . . .
    { Ввод матрицы X }
    Sum:=0; bx:=↑X[1,1]; {Так в Паскале нельзя}
    for cx:=N downto 1 do
        if bx↑<0 then begin oldcx:=cx;
            for cx:=M downto 1 do begin
                Sum:=Sum+bx↑; bx:=bx+2 {Так в Паскале нельзя}
            end;
            cx:=oldcx
        end
    else bx:=bx+2*M {Так в Паскале нельзя}

```

Теперь осталось переписать этот фрагмент программы на Ассемблере:

```

N      equ    20
M      equ    30
oldcx  equ    di
Data   segment
X      dw     N*M dup (?)
Sum    dw     ?
    . . .
Data   ends
    . . .
;      Ввод матрицы X
      mov    Sum,0
      mov    bx,offset X; Адрес X[1,1]
      mov    cx,N
L1:    cmp   word ptr [bx],0
      jge   L3
      mov   oldcx,cx
      mov   cx,M
L2:    mov   ax,[bx]
      add   Sum,ax
      add   bx,2
      loop  L2
      mov   cx,oldcx
      jmp   L4
L3:    add   bx,2*M
L4:    loop  L1

```

Приведённый пример очень хорошо иллюстрирует стиль мышления программиста на Ассемблере. Для доступа к элементам обрабатываемых данных применяются указатели (ссылочные переменные, адреса), и используются операции над этими адресами (адресная арифметика). Получающиеся программы могут максимально эффективно использовать все особенности архитектуры используемого компьютера. Применение адресом и адресной арифметики свойственно и

некоторым языкам высокого уровня (например, языку C), который ориентирован на использование особенности машинной архитектуры для написания более эффективных программ.

Вернёмся к описанию команд цикла. В языке Ассемблера есть и другие команды цикла, которые могут производить досрочный (до исчерпания счётчика цикла) выход из цикла. Как и для команд условного перехода, для мнемонических имен некоторых из них существуют синонимы, которые мы будем разделять в описании этих команд символом /.

Команда

loopz/loope L

выполняется по схеме

Dec (CX) ; **if** (CX<>0) **and** (ZF=1) **then goto** L;

А команда

loopnz/loopne L

выполняется по схеме

Dec (CX) ; **if** (CX<>0) **and** (ZF=0) **then goto** L;

В этих командах необходимо учитывать, что операция Dec (CX) является частью команды цикла и *не меняет* флага ZF.

Как видим, досрочный выход из таких циклов может произойти при соответствующих значениях флага нуля ZF. Такие команды используются в основном при работе с массивами, для усвоения этого материала Вам необходимо изучить соответствующий раздел учебника по Ассемблеру.

7.7. Работа со стеком

Прежде, чем двигаться дальше в описании команд перехода, нам необходимо изучить понятие *стека* и рассмотреть команды работы со стеком.

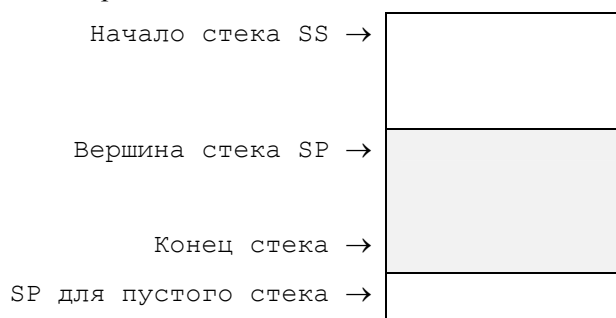
Стеком называется сегмент памяти, на начало которого указывает сегментный регистр SS. При работе программы в регистр SS можно последовательно загружать адреса начал нескольких сегментов, поэтому иногда говорят, что в программе несколько стеков. Однако в каждый момент стек только один – тот, на который сейчас указывает регистр SS. Именно этот стек мы и будем иметь в виду.

Кроме начала, у стека есть текущая позиция – *вершина* стека, её смещение от начала сегмента стека записано в регистре SP (stack pointer). Следовательно, как мы уже знаем, физический адрес вершины стека можно получить по формуле $A_{\text{физ}} = (SS * 16 + SP) \bmod 2^{20}$.

Стек есть аппаратная реализация абстрактной структуры данных *стек*, с которой Вы познакомились в прошлом семестре. В стек можно записывать (и, соответственно, читать из него) только машинные *слова*, чтение и запись байтов не предусмотрена в архитектуре рассматриваемого нами компьютера. Это, конечно, не значит, что в стеке нельзя *хранить* байты, двойные слова и т.д., просто нет машинных *команд* для записи в стек и чтения из стека данных этих форматов.

В соответствие с определением понятия стек последнее записанное в него слово будет читаться из стека первым. Это так называемое правило "последний пришёл – первый вышел" (английское сокращение LIFO).¹ Обычно стек принято изображать "растущим" снизу-вверх. Как следствие получается, что конец стека фиксирован и расположен снизу, а вершина движется вверх (при записи в стек) и вниз (при чтении из стека).

В каждый момент времени регистр SP указывает на *последнее слово*, записанное в стек. Обычно стек изображают, как показано на рис. 7.1.



¹ Вообще говоря, это же правило можно записать и как "первый пришёл – последний вышел" (английское сокращение FILO). В литературе встречаются оба этих правила и их сокращения.

Рис. 7.1. Так мы будем изображать стек.

На нашем рисунке, как обычно, стек растёт снизу-вверх, занятая часть стека закрашена. В начале работы программы, когда стек пустой, регистр *SP* указывает на первое слово за концом стека. Особым является случай, когда стек имеет максимальный размер 2^{16} байт, в этом случае значение регистра *SP* для пустого стека равно нулю, т.е. совпадает со значением этого регистра и для полного стека, поэтому стеки максимального размера использовать не рекомендуется, так как будет затруднён контроль пустоты и переполнения стека.

Обычно для резервирования памяти под стек на языке Ассемблера описывается специальный *сегмент стека*. В наших предыдущих программах мы делали это таким образом:

```
stack segment stack
    dw 64 dup (?)
stack ends
```

Имя сегмента стека и способ резервирования памяти может быть любым, например, можно описать такой стек:

```
st_1 segment stack
    db 128 dup (?)
st_1 ends
```

То, что этот сегмент будет при выполнении программы использоваться именно как сегмент стека, указывается параметром **stack** директивы **segment**. Этот параметр является служебным словом языка Ассемблера и, вообще говоря, не должен употребляться ни в каком другом смысле.¹

В нашем последнем примере размер сегмента стека установлен в *64 слова*, поэтому в начале работы регистр *SP* будет иметь значение 128, т.е., как мы и говорили ранее, указывает на первое слово за концом стека. Области памяти в стеке обычно не имеют имён, так как доступ к ним, как правило, производится только с использованием регистров.

Обратим здесь внимание на важное обстоятельство. Перед началом работы со стеком необходимо загрузить в регистры *SS* и *SP* требуемые значения, однако сама программа это сделать не может, т.к. при выполнении *самой первой* команды программы стек уже должен быть доступен (почему это так мы узнаем в нашем курсе позже, когда будем изучать механизм прерываний). Поэтому в рассмотренных выше примерах программ мы *сами* не загружали в регистры *SS* и *SP* никаких начальных значений. Как мы узнаем позже, перед началом выполнения нашей программы этим регистрам присвоит значения специальная системная программа *загрузчик*, которая размещает нашу программу в памяти и передаёт управление на команду, помеченную той меткой, которая указана в конце нашего модуля в качестве параметра директивы **end**. Разумеется, позже при работе программы мы и сами можем загрузить в регистр *SS* новое значение, это будет *переключением* на другой сегмент стека.

Рассмотрим сначала те команды работы со стеком, которые *не являются* командами перехода. Команда

```
push op1
```

где *op1* может иметь форматы *r16, m16, CS, DS, SS, ES*, записывает в стек слово, определяемое своим операндом. Это команда выполняется по правилу:

$$SP := (SP - 2)_{\text{mod } 2^{16}}; \langle SS, SP \rangle := op1$$

Здесь запись $\langle SS, SP \rangle$ обозначает адрес в стеке, вычисляемый по формуле

$$A_{\text{физ}} = (SS * 16 + SP)_{\text{mod } 2^{20}}.$$

Особым случаем является команда

```
push SP
```

В *младших* моделях нашего семейства она выполняется, как описано выше, а в *старших* – по схеме

$$\langle SS, SP \rangle := SP; SP := (SP - 2)_{\text{mod } 2^{16}}$$

¹ Иногда в наших примерах мы, следуя учебнику [5], называли так же и сам сегмент стека. Некоторые компиляторы с Ассемблера (например, MASM-4.0) допускают это, если по контексту могут определить, что это именно имя пользователя, а не служебное слово. Другие компиляторы (например, Турбо-Ассемблер) подходят к этому вопросу более строго и не допускают использование служебных слов в качестве имён пользователя. Все служебные слова Ассемблера мы выделяем жирным шрифтом.

Следовательно, если мы хотим, чтобы наша программа правильно работала на всех моделях семейства, надо с осторожностью использовать в программе команду `push SP`.

Команда

pop op1

где op1 может иметь форматы r16, m16, SS, DS, ES, читает из стека слово и записывает его в место памяти, определяемое своим операндом. Это команда выполняется по правилу:

$op1 := \langle SS, SP \rangle; SP := (SP + 2) \bmod 2^{16}$

Команда

pushf

записывает в стек регистр флагов FLAGS, а команда

popf

наоборот, читает из стека слово и записывает его в регистр флагов FLAGS. Эти команды удобны для сохранения в стеке и восстановления значения регистра флагов.

В старших моделях нашего семейства появились две новые удобные команды работы со стеком.

Команда

pusha

последовательно записывает в стек регистры AX, CX, DX, BX, SP (этот регистр записывается до его изменения), BP, SI и DI. Команда

popa

последовательно считывает из стека и записывает значения в эти же регистры (но, естественно, в обратном порядке). Эти команды предназначены для сохранения в стеке и восстановления значений сразу всех этих регистров.

Команды записи в стек не проверяют того, что стек уже полон, для надёжного программирования это должен делать сам программист. Например, для проверки того, что стек уже полон, и писать в него нельзя, можно использовать команду сравнения

cmp SP, 0; стек уже полон ?

и выполнить условный переход, если регистр SP равен нулю. Особым случаем здесь будет стек максимального размера 2^{16} байт, для него значение регистра SP=0 как для полного, так и для пустого стека (обязательно понять это!), поэтому не рекомендуется использовать стек максимального размера.

Аналогично для проверки того, что стек уже пуст, и читать из него нельзя, следует использовать команду сравнения

cmp SP, K; стек пуст ?

где K – чётное число – размер стека в байтах. Если размер стека в байтах нечётный, то стек полон при SP=1, т.е. в общем случае необходима проверка SP<2. Обычно избегают задавать стеки нечётной длины, для них труднее проверить и пустоту стека.

В качестве примера использования стека рассмотрим программу для решения следующей задачи. Необходимо вводить целые *беззнаковые* числа до тех пор, пока не будет введено число ноль (признак конца ввода). Затем следует вывести в обратном порядке то из введённых чисел, которые принадлежат диапазону [2..100] (сделаем спецификацию, что таких чисел может быть не более 300). Ниже приведено возможное решение этой задачи.

```
include io.asm
st      segment stack
        db    128 dup (?); это для системных нужд
        dw    300 dup (?); это для хранения наших чисел
st      ends
code    segment
        assume cs:code, ds:code, ss:st
T1      db    'Вводите числа до нуля$'
T2      db    'Числа в обратном порядке:', 10, 13, '$'
T3      db    'Ошибка - много чисел!', 10, 13, '$'
program_start:
        mov   ax, code
        mov   ds, ax
        mov   dx, offset T1; Приглашение к вводу
```



```

    outstr
    newline
    sub    cx,cx; хороший способ для cx:=0
L:      inint ax
    cmp    ax,0; проверка конца ввода
    je     Pech; на вывод результата
    cmp    ax,2
    jb     L
    cmp    ax,100
    ja     L;    проверка диапазона
    cmp    cx,300; в стеке уже 300 чисел ?
    je     Err
    push   ax; запись числа в стек
    inc    cx; счетчик количества чисел в стеке
    jmp    L
Pech:   jcxz Kon; нет чисел в стеке
    mov    dx, offset T2
    outstr
L1:     pop    ax
    outword ax,10; ширина поля вывода=10
    loop  L1
Kon:    finish
Err:    mov    dx,T3
    outstr
    finish
code    ends
end program_start

```

Заметим, что в нашей программе нет собственно переменных, а только строковые константы, поэтому мы не описали отдельный сегмент данных, а разместили эти строковые константы в кодовом сегменте. Можно считать, что сегменты данных и кода в нашей программе *совмещены*. Мы разместили строковые константы в *начале* сегмента кода, перед входной точкой программы, но с таким же успехом можно разместить эти строки и в *конце* кодового сегмента после последней макрокоманды **finish**.

Обратите внимание, как мы выбрали размер стека: 128 байт мы зарезервировали для системных нужд (как уже упоминалось, стеком будут пользоваться и другие программы, подробнее об этом будет рассказано далее) и 300 слов мы отвели для хранения введенных нами чисел. При реализации этой программы может возникнуть желание определять, что введено слишком много чисел, анализируя переполнение стека. Другими словами, вместо проверки

```

    cmp    cx,300; в стеке уже 300 чисел ?
    je     Err

```

казалось бы, можно было поставить проверку исчерпания стека

```

    cmp    SP,2; стек уже полон ?
    jb     Err

```

Это, однако, может повлечь за собой тяжёлую ошибку. Дело в том, что в стеке может остаться совсем мало места, а, как мы знаем, стек использует не только наша, но и другие программы, которые в этом случае будут работать неправильно.

Теперь, после того, как мы научились работать со стеком, вернёмся к дальнейшему рассмотрению команд перехода.

7.8. Команды вызова процедуры и возврата из процедуры

Команды вызова процедуры по-другому называются командами *перехода с возвратом*, что более правильно, так как их можно использовать и без процедур. По своей сути это команды *безусловного перехода*, которые перед передачей управления в другое место программы запоминают в стеке адрес следующей за ними команды. На языке Ассемблера эти команды имеют следующий вид:

```

call op1

```

где `op1` может иметь следующие форматы: `i16`, `r16`, `m16`, `m32` и `i32`. Как видим, по сравнению с командой безусловного перехода здесь не реализован только близкий короткий относительный переход `call i8`, он практически бесполезен в практике программирования, так как почти всегда тело процедуры находится достаточно далеко от точки вызова этой процедуры. Таким образом, как и команды безусловного перехода, команды вызова процедуры бывают *близкими* (внутрисегментными) и *дальними* (межсегментными). Близкий вызов процедуры выполняется по следующей схеме:

```
Встек (IP); jmp op1
```

Здесь запись **Встек** (IP) обозначает действие "записать значение регистра IP в стек". Заметим, что отдельной команды `push IP` в языке машины нет. Дальний вызов процедуры выполняется по схеме:

```
Встек (CS); Встек (IP); jmp op1
```

Для возврата на команду программы, адрес которой находится на вершине стека, предназначена команда *возврата из процедуры*, по сути, это тоже команда безусловного перехода. Команда возврата из процедуры имеет следующий формат:

```
ret [i16]; Параметр может быть опущен
```

На языке машины у этой команды есть две модификации, отличающиеся кодами операций: близкий и дальний возврат из процедуры. Нужный код операции выбирается программой Ассемблера автоматически, по контексту использования команды возврата, о чём мы будем говорить далее. Если программист опускает параметр этой команды `i16`, то Ассемблер автоматически полагает `i16=0`.

Команда *близкого* возврата из процедуры выполняется по схеме:

```
Изстека (IP);  $SP:=(SP+i16)_{\text{mod } 2^{16}}$ 
```

Здесь, по аналогии с командой вызова процедуры, запись **Изстека** (IP) обозначает операцию "считать из стека слово и записать его в регистр IP".

Команда *дальнего* возврата из процедуры выполняется по схеме:

```
Изстека (IP); Изстека (CS);  $SP:=(SP+i16)_{\text{mod } 2^{16}}$ 
```

Действие $SP:=(SP+i16)_{\text{mod } 2^{16}}$ приводит к тому, что указатель вершины стека `SP` устанавливается на некоторое другое место в стеке. В большинстве случаев этот операнд имеет смысл только для *чётных* `i16>0` и `SP+i16<=K`, где `K` – размер стека. В этом случае из стека удаляются `i16 div 2` слов, что можно трактовать как *очистку* стека от данного количества слов (уничтожение соответствующего числа локальных переменных). Возможность очистки стека, как мы увидим, будет весьма полезной при программировании процедур на Ассемблере.

7.9. Программирование процедур на Ассемблере

В языке Ассемблера есть понятие процедуры – это участок программы, который начинается директивой

```
<имя процедуры> proc [<спецификация процедуры>]
```

и заканчивается директивой

```
<имя процедуры> endp
```

Вложенность процедур, в отличие от языка Паскаль, *не допускается*. Имя процедуры имеет тип метки, хотя за ним и не стоит двоеточие. Вызов процедуры обычно производится командой `call`, а возврат из процедуры – командой `ret`.

Спецификация процедуры – это константа `-2` (этой служебной константе в Ассемблере присвоено имя **far**) или `-1` (этой служебной константе в Ассемблере присвоено имя **near**).¹ Если спецификация опущена, то имеется в виду ближняя (**near**) процедура. Спецификация процедуры – это *единственный* способ повлиять на выбор Ассемблером конкретного кода операции для команды возврата `ret` *внутри* этой процедуры: для близкой процедуры это близкий возврат, а для дальней – дальний возврат. Отметим, что для команды `ret`, расположенной *вне* процедуры Ассемблером выбирается *ближний* возврат.

Изучение программирования процедур на Ассемблере начнём со следующей простой задачи: пусть надо ввести массивы `X` и `Y` *знаковых* целых чисел, массив `X` содержит 100 чисел, а массив `Y` содержит 200 чисел. Затем необходимо вычислить величину

¹ Отметим и другие полезные имена констант в языке Ассемблера: **byte**=1, **word**=2, **dword**=4, **abs**=0.

$$\text{Sum} := \sum_{i=1}^{100} X[i] + \sum_{i=1}^{200} Y[i]$$

Будем предполагать, что массивы находятся в одном сегменте данных, а переполнение результата при сложении будем для простоты игнорировать (не выдавать диагностику). Для данной программы естественно реализовать процедуру суммирования элементов массива и дважды вызывать эту процедуру для массивов X и Y. Текст нашей процедуры мы, как и в Паскале, будем располагать *перед* текстом основной программы (начало программы, как мы знаем, помечено меткой, указанной в директиве **end** нашего модуля).

Заметим, что, вообще говоря, мы будем писать *функцию*, но в языке Ассемблера различия между процедурой и функцией не синтаксическое, а чисто семантическое, т.е. о том, что это функция, а не процедура, знает программист, но не программа Ассемблера, поэтому далее мы часто будем использовать обобщённый термин *процедура*.

Перед тем, как писать процедуру, необходимо составить *соглашение о связях* между основной программой и процедурой.¹ Это соглашение включает в себя способ передачи параметров, возврата результата работы и некоторую другую информацию. Так, мы "договоримся" с процедурой, что суммируемый массив *слов* будет располагаться в сегменте данных, адрес первого элемента перед вызовом процедуры будет записан в регистр **bx**, а количество элементов – в регистр **cx**. Сумма элементов массива при возврате из процедуры должна находиться в регистре **ax**. При этих соглашениях о связях у нас получится следующая программа (для простоты вместо команд для ввода массивов вы указали только комментарий).

```
include io.asm
data segment
X    dw    100 dup(?)
Y    dw    200 dup(?)
Sum  dw    ?
data ends
stack segment stack
     dw    64 dup(?)
stack ends
code segment
     assume cs:code,ds:data,ss:stack
Summa proc
; соглашение о связях: bx – адрес первого элемента
; cx=количество элементов, ax – ответ (сумма)
     sub    ax,ax; сумма:=0
L:    add    ax,[bx]
     add    bx,2
     loop   L
     ret
Summa endp
start:mov    ax,data
     mov    ds,ax
; здесь команды для ввода массивов X и Y
     mov    bx, offset X; адрес начала X
     mov    cx,100; число элементов в X
     call   Summa
     mov    Sum,ax; сумма массива X
     mov    bx, offset Y; адрес начала Y
     mov    cx,200; число элементов в Y
     call   Summa
     add    Sum,ax; сумма массивов X и Y
     outint Sum
     newline
     finish
```

¹ Под основной программой мы имеем в виду то место нашей программы, где вызывается процедура, т.е. возможен и случай, когда одна процедура вызывает другую (в том числе рекурсивно саму себя).

```
code ends
end start
```

Если попытаться один к одному переписать эту программу на Турбо-Паскале, то получится примерно следующее:

```
Program S(input,output);
Var X: array[1..100] of integer;
    Y: array[1..200] of integer;
    bx: ↑integer; Sum,cx,ax: integer;
Procedure Summa;
    Label L;
Begin
    ax:=0;
L: ax := ax + bx↑; bx:=bx+2; {так в Паскале нельзя}
    dec(cx); if cx<>0 then goto L
End;
Begin {Ввод массивов X и Y}
    cx:=100; bx:=↑X[1]; {так в Паскале нельзя} 1
    Summa; Sum:=ax;
    cx:=200; bx:=↑Y[1]; {так в Паскале нельзя}
    Summa; Sum:=Sum+ax; Writeln(Sum)
End.
```

Как видим, это очень плохой стиль программирования, так как неявными параметрами процедуры являются глобальные переменные. Решая эту задачу на Паскале, мы бы написали, например, такую программу:

```
Program S(input,output);
Type Mas= array[1..N] of integer;
    {так в Паскале нельзя, N - не константа}
Var X,Y: Mas;
    Sum: integer;
Function Summa(Var A: Mas, N: integer): integer;
    Var i,S: integer;
Begin S:=0; for i:=1 to N do S:=S+A[i]; Summa:=S End;
Begin {Ввод массивов X и Y}
    Sum:=Summa(X,100); Sum:=Sum+Summa(Y,200); Writeln(Sum)
End.
```

Однако для того, чтобы так же хорошо писать на Ассемблере, нам понадобятся другие соглашения о связях между процедурой и основной программой. Вспомним, что хорошо написанная процедура в языке Паскаль получает все свои аргументы как фактические параметры и не использует имён глобальных переменных. При программировании процедур на языке Ассемблера мы будем использовать так называемые стандартные соглашения о связях.

7.9.1. Стандартные соглашения о связях

Сначала поймём необходимость существования некоторых *стандартных* соглашений о связях между процедурой и основной программой. Действительно, иногда программист просто не сможет "договориться", например, с процедурой, как она должна принимать свои параметры. В качестве первого примера можно привести так называемые библиотеки стандартных процедур. В этих библиотеках собраны готовые процедуры, реализующие алгоритмы для некоторой предметной области (например, для работы с матрицами). Такие библиотеки обычно поставляется в виде набора так называемых *объектных модулей*, что исключает возможность вносить изменения в исходный текст этих процедур (с объектными модулями мы познакомимся далее в нашем курсе).

¹ В языке Турбо-Паскаль для этой цели можно использовать оператор присваивания `bx := @X[1]`

Другим примером является написание частей программы на нескольких языках программирования, при этом чаще всего основная программа пишется на некотором языке высокого уровня (Фортране, Паскале, С и т.д.), а процедура – на Ассемблере. Вспомним, что когда мы говорили об областях применения Ассемблера, то одной из таких областей и было написание процедур, которые вызываются из программ на языках высокого уровня. Например, для языка Турбо-Паскаль такая, как говорят, *внешняя*, функция может быть описана следующим образом:

```
Function Summa(Var A: Mas, N: integer): integer;
External;
```

Служебное слово **External** является указанием на то, что эта функция описана не в данной программе и Паскаль-машина должна вызвать эту *внешнюю* функцию, как-то передать ей параметры и получить результат работы функции. Если программист пишет эту функцию на Ассемблере, то он конечно никак не может "договориться" с Паскаль-машиной, как он хочет получать параметры и возвращать результат работы своей функции.

Именно для таких случаев и разработаны **стандартные соглашения о связях**. При этом если процедура или функция, написанная на Ассемблере, соблюдает эти стандартные соглашения о связях, то это гарантирует, что эту процедуру или функцию можно будет вызывать из программы, написанной на другом языке программирования, если в нём тоже соблюдаются такие же стандартные соглашения о связях.

Рассмотрим типичные стандартные соглашения о связях, обычно они включают следующие пункты.

- Фактические параметры перед вызовом процедуры или функции записываются в стек.¹ При передаче параметра *по значению* в стек записывается это значение, а в случае передачи параметра *по ссылке* в стек записывается адрес начала фактического параметра.² Порядок записи фактических параметров в стек может быть *прямым* (сначала записывается первый параметр, потом второй и т.д.) или *обратным* (когда, наоборот, сначала записывается последний параметр, потом предпоследний и т.д.). В разных языках программирования этот порядок различный. Так, в языке С это *обратный* порядок, а в большинстве других языков программирования высокого уровня – *прямой*.³
- Если в процедуре или функции необходимы *локальные* переменные, то место им отводится в стеке. Обычно это делается путём увеличения размера стека, для чего, как мы уже знаем, надо *уменьшить* значение регистра SP на число байт, которые занимают эти локальные переменные.
- Функция возвращает своё значение в регистрах al, ax или в паре регистров <dx, ax>, в зависимости от величины этого значения. Для возврата значений, превышающих двойное слово, устанавливаются специальные соглашения.
- Если в процедуре или функции изменяются регистры, то в начале работы необходимо запомнить значения этих регистров в локальных переменных, а перед возвратом – восстановить эти значения (для функции, естественно, не запоминаются и не восстанавливаются регистр(ы), на котором(ых) возвращается результат её работы). Обычно также не запоминаются и не восстанавливаются регистры для работы с вещественными числами.
- Перед возвратом из процедуры и функции стек очищается от всех локальных переменных, в том числе и от фактических параметров (вспомним, что в языке Паскаль *формальные* параметры, в которые передаются соответствующие им *фактические* параметры, тоже являются *локальными* переменными процедур и функций!).

¹ Практически все современные компьютеры имеют аппаратно реализованный стек, если же это не так, то в стандартных соглашениях о связях для таких ЭВМ устанавливаются какие-нибудь другие способы передачи параметров, этот случай мы рассматривать не будем.

² Является ли адрес близким (т.е. *смещением* в сегменте данных) или же дальним (значение сегментного регистра и смещение) обычно специфицируется при написании процедуры на языке высокого уровня. Ясно, что, скорее всего во внешнюю процедуру будут передаваться дальние адреса.

³ Обратный порядок позволяет более легко реализовывать процедуры и функции с *переменным* числом параметров, но он менее эффективен, чем прямой, так как труднее удалить из стека фактические параметры после окончания процедуры. В связи с этим, например, в языке С при описании внешней процедуры программист может явно указывать порядок записи фактических параметров в стек.

Участок стека, в котором процедура или функция размещает свои локальные переменные (в частности, фактические параметры) называется *стековым кадром* (stack frame). Стековый кадр начинает строить основная программа перед вызовом процедуры или функции, помещая туда фактические параметры. Затем команда передачи управления с возвратом **call** помещает в стек адрес возврата (это одно слово для близкой процедуры и два – для дальней). Далее уже сама процедура или функция продолжает построение стекового кадра, размещая в нём свои локальные переменные.

Заметим, что если построением стекового кадра занимаются как основная программа, так и процедура (функция), то полностью разрушить стековый кадр должна процедура (функция), так что при возврате в основную программу стековый кадр будет уже уничтожен.¹

Перепишем теперь нашу последнюю программу с использованием стандартного соглашения о связях. Будем предполагать, что передаваемый по ссылке адрес фактического параметра-массива занимает одно слово (т.е. является смещением в сегменте данных). Для хранения стекового кадра (локальных переменных функции) зарезервируем в стеке 32 слова. Ниже показано возможное решение этой задачи.

```
include io.asm
data segment
X    dw    100 dup(?)
Y    dw    200 dup(?)
Sum  dw    ?
data ends
stack segment stack
     dw    64 dup (?); для системных нужд
     dw    32 dup (?); для стекового кадра
stack ends
code segment
     assume cs:code,ds:data,ss:stack
Summa proc near
; стандартные соглашение о связях
     push bp
     mov  bp,sp; база стекового кадра
     push bx
     push ax
     push cx;   запоминание регистров
     sub  sp,2;  порождение локальной переменной
S     equ  word ptr [bp-8]
; имя S будет эквивалентным адресу локальной переменной
     mov  cx,[bp+4]; cx:=длина массива
     mov  bx,[bp+6]; bx:=адрес первого элемента
     mov  S,0; сумма:=0
L:    mov  ax,[bx]; сложение двумя командами,
     add  S,ax;   так как нет формата память-память
     add  bx,2
     loop L
     mov  ax,S; результат функции
     add  sp,2; уничтожение локальной переменной
     pop  cx
     pop  ax
     pop  bx
     pop  bp; восстановление регистров cx, bx и bp
     ret  2*2
; возврат с очисткой стека от фактических параметров
Summa endp
```

¹ При обратном порядке записи фактических параметров в стек (как в языке С) удаление из стека параметров обычно делает не процедура, а основная программа, т.к. это легче реализовать для *переменного* числа параметров. Плохо в этом случае то, что такое удаление приходится выполнять в *каждой* точке возврата из процедуры, что может существенно увеличить размер кода программы.

```

start:mov  ax,data
      mov  ds,ax
; здесь команды для ввода массивов X и Y
      mov  ax, offset X; адрес начала X
      push ax; первый фактический параметр
      mov  ax,100
      push ax; второй фактический параметр
      call Summa
      mov  Sum,ax; сумма массива X
      mov  ax, offset Y; адрес начала Y
      push ax; первый фактический параметр
      mov  ax,200
      push ax; второй фактический параметр
      call Summa
      add  Sum,ax; сумма массивов X и Y
      outint Sum
      newline
      finish
code  ends
      end start

```

Подробно прокомментируем эту программу. Первый параметр функции у нас передаётся по ссылке, а второй – по значению. После выполнения команды вызова процедуры `call Summa` стековый кадр имеет вид, показанный на рис. 7.2. После полного формирования стековый кадр будет иметь вид, показанный на рис. 7.3.

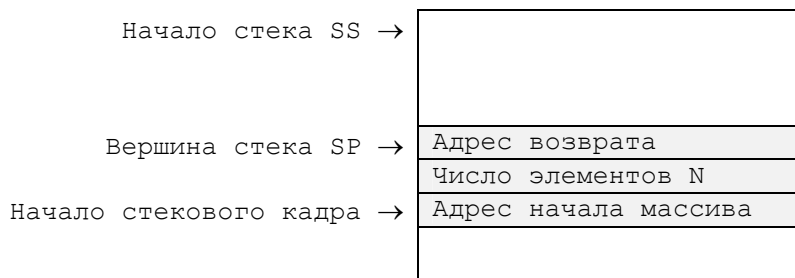


Рис. 7.2. Вид стекового кадра при входе в функцию Summa.

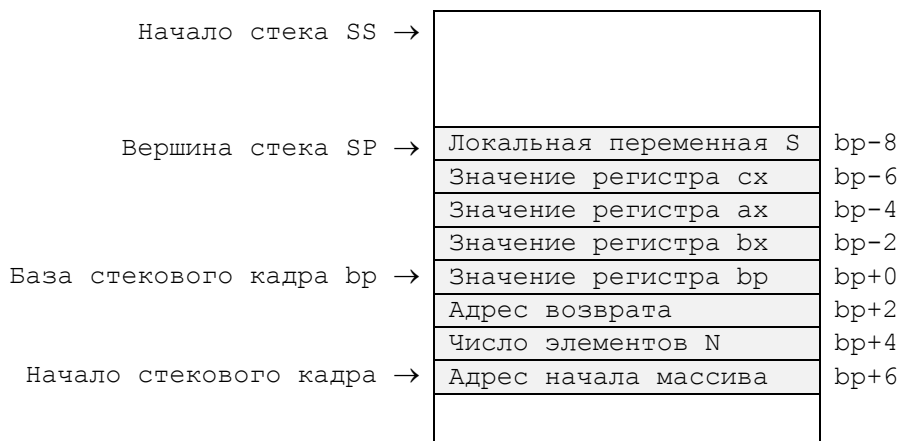


Рис. 7.3. Вид полного стекового кадра (справа показаны смещения слов кадра относительно значения регистра bp).

Сначала отметим особое значение, которое имеет индексный регистр bp при работе со стеком. В архитектуре нашего компьютера это единственный индексный регистр, который предписывает по умолчанию осуществлять запись и чтение данных из *сегмента стека*. Так команда нашей программы

```
mov  cx, [bp+4]; cx:=длина массива
```

читает в регистр cx слово, которое расположено по физическому адресу

$$A_{\text{физ}} = (SS*16 + (4 + \langle \text{bp} \rangle)_{\text{mod } 2^{16}})_{\text{mod } 2^{20}},$$

а не по адресу

$$A_{\text{физ}} = (DS*16 + (4 + \langle \text{bp} \rangle)_{\text{mod } 2^{16}})_{\text{mod } 2^{20}},$$

как происходит при использовании на месте `bp` любого другого индексного регистра, т.е. `bx`, `si` или `di`.

Таким образом, если установить регистр `bp` внутрь стекового кадра, то его легко использовать для доступа к локальным переменным процедуры или функции. Так мы и поступили в нашей программе, поставив регистр `bp` примерно на середину стекового кадра. Теперь, отсчитывая смещения от регистра `bp` вниз, например `[bp+4]`, мы получаем доступ к фактическим параметрам, а, отсчитывая смещение вверх – доступ к сохранённым значениям регистров и локальной переменной, например `[bp-8]` это адрес локальной переменной, которую в программе на Паскале мы назвали именем `S` (см. рис. 7.3).

Обратите внимание, что локальные переменные в стековом кадре не имеют имён, что может быть не совсем удобно. В нашем примере мы присвоили локальной переменной имя `S` при помощи директивы эквивалентности

```
S equ word ptr [bp-8]
```

И теперь всюду вместо имени `S` Ассемблер будет подставлять выражение `word ptr [bp-8]`, которое имеет, как нам и нужно, тип слова. Для порождение этой локальной переменной мы отвели ей место в стеке с помощью команды

```
sub sp,2; порождение локальной переменной
```

т.е. просто уменьшили значение регистра-указателя вершины стека на два байта. Этой же цели можно было бы достичь, например, более короткой (но менее понятной для нашей цели) командой

```
push ax; порождение локальной переменной
```

Перед возвратом из функции мы начали разрушение стекового кадра, как этого требуют стандартные соглашения о связях. Сначала командой

```
add sp,2; уничтожение локальной переменной
```

мы уничтожили локальную переменную, затем восстановили из стека старые значения регистров `cx`, `bx` и `bp` (заметьте, что регистр `bp` нам больше не понадобится в нашей функции). И, наконец, команда возврата

```
ret 2*2; возврат с очисткой стека
```

удаляет из стека адрес возврата и значение двух слов – фактических параметров функции. Уничтожение стекового кадра завершено.

Продолжение изучения стандартных соглашений о связях мы начнём со следующего замечания. В различных системах программирования стандартные соглашения о связях могут несколько различаться. Например, результат значения функции может возвращаться не на регистре `ax`, как в нашем последнем примере, а на *вершине стека*. В этом случае функция должна запоминать и восстанавливать регистр `ax` наравне с другими регистрами, а полное разрушение стекового кадра будет производить основная программа путем чтения результата работы функции из стека.

Важной особенностью использования стандартных соглашений о связях является и то, что они позволяют производить *рекурсивный* вызов процедур и функций, причём рекурсивный и не рекурсивный вызовы "по внешнему виду" не отличаются друг от друга. В качестве примера рассмотрим реализацию функции вычисления факториала от неотрицательного целого числа, при этом будем предполагать, что значение факториала поместится в слово. На языке Турбо-Паскаль эта функция имеет следующий вид:

```
Function Factorial(N: word): word;
Begin
  if N<=1 then Factorial:=1
  else Factorial:=N*Factorial(N-1)
End;
```

Реализуем теперь эту функцию в виде близкой процедуры на Ассемблере:

```
Factorial proc near; стандартные соглашение о связях
```



```

    push bp
    mov  bp,sp; база стекового кадра
    push dx
N     equ  word ptr [bp+4]; фактический параметр N
    mov  ax,1; Factorial(N<=1)
    cmp  N,1
    jbe  Vozv
    mov  ax,N
    dec  ax; N-1
    push ax
    call Factorial; Рекурсия
    mul  N;    Factorial(N-1)*N
Vozv: pop  dx
    pop  bp
    ret  2
Factorial endp

```

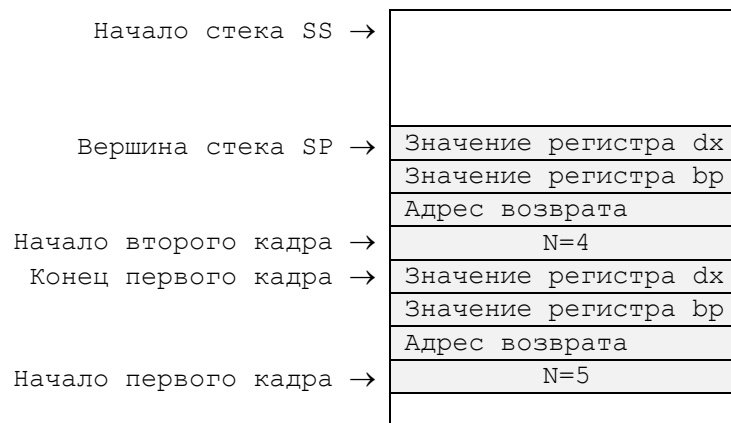


Рис. 7.4. Два стековых кадра функции Factorial.

Рассмотрим вызов этой функции Factorial для вычисления факториала числа 5. Такой вызов можно в основной программе сделать, например, следующими командами:

```

mov  ax,5
push ax
call Factorial
outword ax

```

На рис. 7.4 показан вид стека, когда произведён первый *рекурсивный* вызов функции, в стеке при этом два стековых кадра.

В качестве ещё одного примера рассмотрим реализацию с помощью *процедуры* следующего алгоритма: задана константа N=30000, найти скалярное произведение двух массивов, содержащих по N беззнаковых целых чисел.

$$\text{Scal} := \sum_{i=1}^N X[i] * Y[i]$$

На языке Паскаль это можно записать, например, следующим образом: ¹

```

Const N=30000;
Type  Mas = array[1..N] of word;
Var   A,B: Mas; S: word;
Procedure SPR(var X,Y: Mas; N: integer; var Scal: word);
    Var i: integer;
Begin Scal:=0; for i:=1 to N do Scal:=Scal+X[i]*Y[i] end;

```

¹ На языке Турбо-Паскаль так написать нельзя, массивы А и В не поместятся в один сегмент данных, а размещать *статические* переменные в разных сегментах Турбо-Паскаль не умеет.

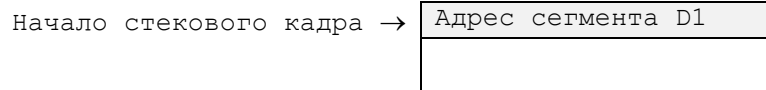


Рис. 7.5. Стековый кадр при входе в процедуру скалярного произведения.

Теперь опишем нашу дальнюю процедуру:

```

SPR  proc far
      push bp; база стекового
      mov   bp,sp; кадра
; сохранение остальных регистров
      push ds
      push es
.186
      pusha ;в стек ax,cx,dx,bx,sp,bp,si,di
      sub   bx,bx; локальная сумма
      mov   cx,[bp+10]; Выбор N
      mov   ds,[bp+18]; Сегмент D1
      mov   si,[bp+16]; Адрес A
      mov   es,[bp+14]; Сегмент D2
      mov   di,[bp+12]; Адрес B
L:    mov   ax,[si]; A[i]
      mul  word ptr es:[di]; A[i]*B[i]
      jc   Err; при переполнении
      add  bx,ax
      jc   Err; при переполнении
      add  di,2
      add  si,2
      loop L
Vozv: mov  ds,[bp+8]; Сегмент D1
      mov  si,[bp+6]; Адрес S
      mov  [si],bx; Результат в S
; восстановление регистров
      popa ;из стека ax,cx,dx,bx,sp,bp,si,di
      pop  es
      pop  ds
      pop  bp
      ret  2*7; Очистка 7 слов из стека
Err:  mov  bx,-1;Код ошибки
      jmp  Vozv
SPR  endp

```

В этом примере для экономии текста программы мы использовали команды `pusha` и `popa` из языка команд старшей модели нашего семейства ЭВМ, о чём предупредили Ассемблер директивой `.186`.

На этом мы закончим изучение процедур в языке Ассемблера.

8. Система прерываний.

Далее мы продолжим изучение переходов. Как мы уже упоминали, некоторые переходы производятся не при выполнении команд, а могут делаться центральным процессором *автоматически* при возникновении определённых условий. Если компьютер обладает такими способностями, то говорят, что в этом компьютере реализована *система прерываний*. Все современные компьютеры имеют систему прерываний, и сейчас мы начнём изучать, что это такое.

Сначала введём понятие *события* (возникшей ситуации). События могут возникать как в центральном процессоре (например, деление на ноль, попытка выполнить машинную команду с несуществующим кодом операции, выполнение некоторых особых команд и т.д.), так и в периферийных устройствах (например, нажата кнопка мыши, на печатающем устройстве кончилась бумага, получен сигнал по линиям связи и др.). Ясно, что при возникновении события продолжать

выполнение программы может быть либо бессмысленно (деление на ноль), либо нежелательно, так как нужно срочно предпринять какие-то действия, для выполнения которых текущая программа просто не предназначена (например, надо отреагировать на нажатие кнопки мыши, на сигнал от встроенного таймера и т.д.).

В архитектуре компьютера предусмотрено, что каждое устройство, в котором произошло событие (центральный процессор, память, устройства ввода/вывода) генерирует *сигнал прерывания* – электрический импульс, который приходит на специальную электронную схему центрального процессора. Сигнал прерывания, связанный с каждым из событий, имеет свой *номер*, чтобы отличить его от сигналов, связанных с другими событиями. По месту возникновения сигналы прерывания бывают внутренними (в центральном процессоре) и внешними (в периферийных устройствах).

Получив такой сигнал, центральный процессор автоматически предпринимает некоторые действия, которые называются *аппаратной реакцией* на сигнал прерывания. Надо сказать, что, хотя такая реакция, конечно, сильно зависит от архитектуры компьютера, всё же можно указать какие-то общие черты, присущие всем ЭВМ. Сейчас мы рассмотрим, что обычно входит в аппаратную реакцию центрального процессора на сигнал прерывания.

Сначала надо сказать, что центральный процессор "смотрит", пришел ли сигнал прерывания, только после выполнения очередной команды, таким образом, этот сигнал ждёт завершения текущей команды.¹ Исключением из этого правила являются команды **halt** и **wait**. Команда **halt** останавливает выборку команд центральным процессором, и только сигнал прерывания может вывести компьютер из этого "ничегонеделания". Команда **wait** в младшей модели нашего семейства ждёт окончания операции с вещественными числами, которые мы не рассматриваем. Кроме того, прерывание не возникает после выполнения команды-префикса программного сегмента, т.к. она существенно влияет на следующую за ней команду.

К описанному выше правилу начала аппаратной реакции на сигнал прерывания необходимо сделать существенное замечание. Дело в том, что большинство современных ЭВМ отступают от принципа фон Неймана *последовательного* выполнения команд. Напоминаем, что согласно этому принципу очередная команда начинала выполняться только после полного завершения текущей команды.

Современные компьютеры могут одновременно выполнять несколько команд программы (а наиболее "продвинутые" из них – даже несколько команд из *разных* программ). Компьютеры, обладающие такими возможностями, называются *конвейерными*, они могут одновременно выполнять до восьми и более команд. Для конвейерных ЭВМ необходимо уточнить, когда начинается аппаратная реакция на сигнал прерывания. Обычно это происходит после полного завершения *любой* из выполняющихся в данный момент команд. Выполнение остальных команд прерывается и в дальнейшем их необходимо повторить *с начала*. Понятно, что конвейерные ЭВМ весьма "болезненно" относятся к сигналам прерывания, так как при этом приходится *повторять заново* несколько последних частично выполненных команд прерванной программы. Несколько более подробно о конвейерных ЭВМ мы поговорим в конце нашей книги.

Итак, после окончания текущей команды центральный процессор анализирует номер сигнала прерывания (для нашего компьютера это целое беззнаковое число формата `i8`). Для некоторых из этих номеров сигнал прерывания *игнорируется*, и центральный процессор переходит к выполнению *следующей* команды программы. Говорят, что прерывания с такими номерами *в данный момент* запрещены или *замаскированы*. Для компьютера нашей архитектуры можно замаскировать некоторые прерывания от внешних устройств (кроме прерывания с номером 2), установив в ноль значение специального *флага прерывания* `IF` в регистре флагов `FLAGS` (это можно выполнить командой `cli`). Для компьютеров некоторых других архитектур можно замаскировать каждое прерывание по отдельности, установив в ноль соответствующий этому прерыванию бит в специальном *регистре маски* прерываний. Говорят, что прерывания с определёнными номерами можно *закрывать* (маскировать) и *открывать* (разрешать, снимать с них маску).

¹ Даже если это деление на ноль, всё равно можно считать, что центральный процессор *закончил* выполнение этой команды, хотя значение частного при этом, конечно, не определено. Для особо любознательных студентов можно также сказать, что некоторые старые ЭВМ могли начать реакцию на прерывание и без завершения текущей команды (например, если это "длинная" команда, выполнение которой занимает много времени, в этом случае выполнение такой команды могло быть продолжено с прерванного места).

В том случае, если прерывание игнорируется (замаскировано), сигнал прерывания, тем не менее, продолжает оставаться на входе соответствующей схемы центрального процессора до тех пор, пока маскирование этого сигнала не будет снято, или же на вход центрального процессора придёт следующий сигнал прерывания. В последнем случае первый сигнал безвозвратно теряется, что может быть очень опасно, так как мы не прореагировали должным образом на некоторое событие, и оно прошло для нас незамеченным. Отсюда следует, что маскировать сигналы прерываний от внешних устройств можно только на некоторое весьма короткое время, после чего необходимо *открыть* возможность реакции на такие прерывания.

Разберёмся теперь, зачем вообще может понадобиться замаскировать прерывания. Дело в том, что, если произошедшее прерывание не замаскировано то, как мы вскоре увидим, центральный процессор прерывает выполнение *текущей* программы и переключается на выполнение некоторой в общем случае *другой* программы. Это может быть нежелательно и даже опасно, если текущая программа занята срочной работой, которую нельзя прерывать даже на короткое время. Например, эта программа может обрабатывать некоторое важное событие с другим номером прерывания, или же управлять каким-либо быстрым процессом во внешнем устройстве (линия связи с космическим аппаратом, химический реактор и т.д.).

С другой стороны, необходимо понять, что должны существовать и специальные *немаскируемые* сигналы прерывания. В качестве примера такого сигнала можно привести сигнал о неисправности в работе самого центрального процессора, ясно, что маскировать его бессмысленно. Другим примером может служить сигнал о том, что выключилось электрическое питание компьютера. Надо сказать, что в этом случае компьютер останавливается не мгновенно, какую-то долю секунды он ещё может работать за счёт энергии конденсаторов в блоке питания. Этого времени хватит на выполнение нескольких десятков или даже сотен тысяч команд и можно принять важные решения: послать сигнал тревоги, спасти ценные данные в энергонезависимой памяти (такая память называется статической), переключится на резервный блок питания и т.д. Кроме того, бессмысленно маскировать сигналы прерываний, которые выдаются при выполнении некоторых специальных команд, т.к. основным назначением этих команд и является выдача сигнала прерывания. Для нашего компьютера, как уже упоминалось, существует только одно немаскируемое прерывание *от внешних устройств* с номером 2.

Продолжим теперь рассмотрение аппаратной реакции на *незамаскированное* прерывание. Сначала центральный процессор автоматически запоминает в некоторой области памяти (обычно в текущем стеке) самую необходимую (минимальную) информацию о прерванной программе. Во многих книгах по архитектуре ЭВМ это называется *малым упрятыванием* информации о считающейся в данный момент программе, что хорошо отражает смысл такого действия. Для нашего компьютера в стек последовательно записываются значения трёх регистров центрального процессора, это регистр флагов (FLAGS), кодовый сегментный регистр (CS) и счётчик адреса (IP). Как видим, эти действия при минимальном упрятывании похожи на действия при выполнении команды перехода с возвратом **call**, да и назначение у них одно – обеспечить возможность возврата в прерванное место текущей программы. Из этого следует, что стек должен быть у любой программы, даже если она сама им и не пользуется.¹

После выполнения минимального упрятывания центральный процессор по определённым правилам находит (вычисляет) адрес оперативной памяти, куда надо передать управление для обработки сигнала прерывания с данным номером. Говорят, что на этом месте оперативной памяти находится программа реакции (процедура обработки прерывания, обработчик) сигнала прерывания с данным номером.

Для компьютера нашей архитектуры определение адреса начала процедуры-обработчика прерывания с номером N производится по следующему правилу. В начале оперативной памяти расположен так называемый *вектор прерываний* – массив из 256 элементов (по числу возможных номеров прерываний от 0 до 255). Каждый элемент этого массива состоит из двух машинных слов (т.е. имеет формат m32) и содержит *дальний* адрес процедуры-обработчика. Таким образом, адрес процедуры-обработчика прерывания с номером N находится в двух словах, расположенных по физическим адресам $4 \cdot N$ и $4 \cdot N + 2$. Можно сказать, что для перехода на процедуру-обработчика необходимо выполнить безусловный переход

¹ Хотя некоторые программы могут выполняться полностью в режиме закрытых прерываний, но не надо забывать, что существуют прерывания, которые нельзя замаскировать.

```
jmp dword ptr [4*N]; IP:=[4*N], CS:=[4*N+2]
```

Заметим, что на самом деле это аналог дальнего перехода с возвратом, так как в стек уже занесена информация о точке возврата в прерванную программу.

Непосредственно перед переходом на процедуру-обработчика центральный процессор закрывает (маскирует) внешние прерывания, так что обработчик начинает своё выполнение в режиме запрета прерываний. Это гарантирует, что, начав свою работу, процедура-обработчик не будет тут же прервана другим сигналом прерывания. Для нашей архитектуры центральный процессор устанавливает в ноль флаги IF и TF регистра флагов. Как мы уже говорили, значение флага IF=0 маскирует все прерывания от внешних устройств, кроме прерывания с номером 2. Флаг TF устанавливается равным нулю потому, что при значении TF=1 центральный процессор всегда посылает *сам себе* сигнал прерывания с номером N=1 после выполнения *каждой* команды. Этот флаг используется для пошагового выполнения (трассировки) программы, Вы будете изучать эту тему в курсе "Системное программное обеспечение".

На этом *аппаратная* реакция на незамаскированное прерывание заканчивается. Заметим, что некоторым аналогом аппаратной реакции ЭВМ на прерывание в живой природе является *безусловный рефлекс*. Безусловный рефлекс позволяет живому существу "автоматически" (а, следовательно, быстро, "не раздумывая") реагировать на произошедшее событие. Например, если человек обжигает пальцы на огне, то сначала он автоматически отдёргивает руку, а лишь потом начинает разбираться, что же произошло. Так и компьютер по сигналу прерывания автоматически "не раздумывая" переключается на процедуру-обработчика этого сигнала.

Итак, после завершения аппаратной реакции на незамаскированный сигнал прерывания начинает работать процедура-обработчик прерывания с данным номером, эта процедура выполняет *программную* реакцию на прерывание. Аналогом программной реакции на прерывание в живой природе можно считать условный рефлекс. Как у живого организма можно выработать условный рефлекс на некоторый внешний раздражитель, так и компьютер можно "научить", как реагировать на то или иное событие, написав процедуру-обработчика сигнала прерывания с номером, соответствующим этому событию.

Рассмотрим теперь схему работы процедуры-обработчика прерывания. Напомним, что эта процедура начинает выполняться в режиме с *закрытыми* прерываниями от внешних устройств. Как мы говорили выше, долго работать в таком режиме очень опасно, и следует как можно скорее разрешить (открыть) прерывания, для нашего компьютера это установка в единицу флаг IF (это можно выполнить командой `sti`)

Действия, выполняемые в режиме с закрытыми прерываниями, обычно называются *минимальной программной реакцией* на прерывание. Как правило, минимальная программная реакция включает в себя следующие действия.

- Для прерванной программы запоминается вся информация, необходимая для возврата в эту программу. Это все адресуемые регистры (в том числе сегментные регистры и регистры для работы с вещественными числами), а также некоторые системные регистры (последнее сильно зависит от архитектуры конкретного компьютера). Вся эта информация запоминается в специальной области памяти, связанной с прерванной программой, обычно это область памяти называется *информационным полем* программы или *контекстов* программы.¹
- Выполняются самые необходимые действия, связанные с произошедшим событием. Например, если нажата или отпущена клавиша на клавиатуре, то это надо где-то зафиксировать (например, запомнить в очереди введённых с клавиатуры символов). Если этого не сделать на этапе минимальной реакции и открыть прерывания, то процедура-обработчик может быть надолго прервана новым сигналом, который произведёт переключение на какую-то другую процедуру-обработчика, за время работы которой уже может быть нажата другая клавиша, а информация о нажатой ранее клавише таким образом будет потеряна.

После выполнения минимальной программной реакции процедура-обработчик включает (разрешает) прерывания – в нашей архитектуре устанавливает IF=1. Далее производится *полная программная реакция* на прерывания, т.е. процедура-обработчик выполняет *все* необходимые действия, связанные с происшедшим событием. Вообще говоря, допускается, что на этом этапе наша

¹ Точнее *контекстом процесса*, о процессах Вы узнаете в курсе "Системное программное обеспечение".

процедура-обработчик может быть прервана другим сигналом прерывания.¹ В этом случае процедура-обработчик должна при каждом входе в неё резервировать память под свой контекст, где будут запоминаться данные, необходимые для возврата в эту процедуру.

Закончив полную обработку сигнала прерывания, процедура-обработчик должна вернуть управление программе, прерванной последним сигналом прерывания.² Для этого сначала необходимо из контекста прерванной программы восстановить значение всех её регистров (кроме регистров `FLAGS`, `CS` и `IP`). После этого надо произвести возврат на следующую команду прерванной программы, для чего в нашем компьютере можно использовать специальную команду языка машины – команду выхода из прерывания

`iret`

Эта команда без параметров выполняется по схеме:

Изстека (IP) ; Изстека (CS) ; Изстека (FLAGS)

Напомним, что уже восстановлены регистры `SS` и `IP` прерванной программы, т.е. из её стека можно читать.

В Таблице 8.1 изображено начало вектора прерываний для компьютера изучаемой нами архитектуры.

Таблица 8.1. Начало вектора прерываний

Номер	Описание события
N=0	Ошибка в команде деления
N=1	Установлен флаг <code>TF=1</code>
N=2	Немаскируемое внешнее прерывание
N=3	Выполнена команда <code>int</code>
N=4	Выполнена команда <code>into</code> и <code>OF=1</code>
N=5	...
N=6	Команда с плохим кодом операции
...	

Продолжим теперь изучение переходов, вызванных командами прерываний, выполнение каждой такой команды в нашей архитектуре *всегда* вызывает прерывание (исключением является команда `into`, которую мы рассмотрим далее). Каждая команда прерывания реализует *дальний косвенный переход* (понять это!).

Сначала рассмотрим команды, о которых упоминается в Таблице 8.1. Команда `int` является самой короткой (длиной в один байт) командой, которая всегда вызывает прерывание с номером `N=3`. В основном эта команда используется при работе *отладчика* – специальной программы, облегчающей программисту разработку новых программ. Отладчик ставит в программный код отлаживаемой программы так называемые *контрольные точки* – это те места, в которых отлаживаемая программа должна передать управление программе-отладчику. Для такой передачи хорошо подходит команда `int`, если программа-отладчик реализована в виде обработчика прерывания с номером `N=3`. Более подробно с работой отладчика Вы будете знакомиться в курсе следующего семестра "Системное программное обеспечение".

Команда `into` реализует *условное* прерывание: при выполнении этой команды прерывание происходит только в том случае, если флаг `OF=1`, иначе продолжается последовательное выполнение программы. Основное назначение команды `into` – *эффективно* реализовать надёжное

¹ В частности, сигналом с таким же номером `N`, при этом произойдёт *повторный* вход в начало этой же самой процедуры-обработчика. Те программы, которые допускают такой повторный вход в своё начало (не путать это с рекурсивным вызовом!), называются *повторно-входимыми* или *реентерабельными*. Программы обработки прерываний для младших моделей нашего семейства могли и не быть реентерабельными, что порождало известные проблемы, которые мы здесь обсуждать не будем.

² Если прерванных (готовых к продолжению своего счёта) программ больше одной, то часто может понадобиться произвести возврат не в последнюю прерванную программу, а в какую-нибудь другую из этих программ. Поэтому после окончания своей работы процедура-обработчик прерывания передаёт управление специальной системной программе – *диспетчеру*, и уже программа-диспетчер определяет, которая из прерванных программ должна быть продолжена.

программирование при обработке целых *знаковых* чисел. Как мы знаем, при выполнении операций сложения и вычитания со знаковыми числами возможна ошибка, при этом флаг переполнения устанавливается в единицу. Кроме того, вспомним, что флаг переполнения устанавливается в единицу и при операциях умножения, если *значащие* биты результата попадают в *старшую* часть произведения.

При надёжном программировании проверку флага переполнения необходимо ставить после каждой такой команды. Для такой проверки хорошо подходит команда `into`, так как эта самая короткая (однобайтная) команда условного перехода по значению $OF=1$. При этом, правда, обработку аварийной ситуации должна производить процедура-обработчик прерывания с номером $N=4$.

Рассмотрим в качестве примера простейшую реализацию такой процедуры-обработчика прерывания. Для простоты эта процедура-обработчик будет *фрагментом* нашей программы и располагается в сегменте кода. При возникновении ошибки будет просто выдаваться аварийная диагностика, и продолжаться выполнение нашей программы (естественно, с неверным результатом). Заметим, что наш обработчик прерывания *не является* процедурой в смысле языка Ассемблер.

```
include io.asm
data segment
A dw ?
X dw ?
Old dw ?
Dw ?
Diagn db 'Ошибка - большое значение!$'
Data ends
st segment stack
dw 64 dup (?)
st ends
code segment
assume cs:code, ds:data, ss:st
start:mov ax,data
mov ds,ax
; инициализация обработчика into
mov ax,0
mov es,ax; es - на вектор прерываний
My_into equ word ptr es:[4*4]
; сохранения адреса старой процедуры into
mov ax,My_into
mov Old,ax
mov ax,My_into+2
mov Old+2,ax;
; занесение нового адреса процедуры into
cli ; Закрыть прерывания
mov My_into,offset Error; Начало
mov My_into+2,code; процедуры-обработчика
sti Открыть прерывания
; собственно начало программы
mov ax,data
mov ds,ax
inint A
inint X
mov ax,A
add ax,X; Возможно переполнение
into
imul X; Возможны значащие биты в DX
into
mov X,ax; X:=X*(A+X)
outint X
; восстановление старого адреса into
Voz: cli ; Закрыть прерывания
```



```

mov ax,Old
mov My_into,ax
mov ax,Old+2
mov My_into+2,ax
sti Открыть прерывания

```

```

finish

```

```

; Начало нашей процедуры-обработчика
Error:

```

```

; Минимальная программная реакция
push ds; Сохранение регистров
push dx
push ax
sti Открыть прерывания

```

```

; Полная программная реакция
mov ax,data
mov ds,ax
mov dx,offset Diagn
outstr
newline
pop ax; Восстановление регистров
pop dx
pop ds
iret ; Возврат из прерывания
code ends
end start

```

Обратите внимание, что при выполнении некоторых групп команд работу нашей программы нельзя прерывать, иначе может возникнуть ситуация, когда в вектор прерывания не занесётся полностью вся необходимая информация, а уже произойдёт переключение на другую программу по пришедшему сигналу прерывания. В программировании такие группы команд называются *критическими секциями* программы. В нашей программе критические секции заключены в рамки, в начале каждой критической секции стоит команда запрета прерывания от внешних устройств `cli`, а в конце секции – команда открытия прерываний `sti`.

После выдачи диагностики об ошибке наша процедура-обработчик может не продолжать выполнение программы, а, например, завершать выполнение программы. Для этого вместо предложения

```

iret ; Возврат из прерывания

```

надо поставить два предложения

```

add SP,3*2; Очистка стека от IP, CS и FLAGS
jmp Voz

```

И, наконец, рассмотрим команду, которая всегда вызывает прерывание с номером N, заданным в качестве её операнда:

```

int op1

```

Здесь `op1` имеет формат `i8`. Заметим, что с помощью этой команды можно вызвать прерывание с любым номером, например прерывание, соответствующее делению на ноль или плохому коду операции. Более того, прерывания с номерами большими 31, в нашей архитектуре можно вызвать, *только* выполняя команду `int` с соответствующим параметром-номером прерывания. Используя эти команды, легко отлаживать процедуры-обработчики прерываний, но основное назначение таких команд состоит в другом.

Дело в том, что в большинстве программ необходимо выполнять некоторые широко распространённые действия (обмен данными с внешними устройствами, выполнение стандартных процедур и многое другое). Обычно процедуры, реализующие эти действия, оформляются в виде библиотеки стандартных процедур и всегда находятся в оперативной памяти компьютера. Так как адреса этих процедур часто меняются, то лучше всего присвоить каждой такой процедуре свой номер N и оформлять такие процедуры в виде обработчиков прерываний с этим номером. В этом случае вызов конкретной процедуры с номером N следует производить командой `int N`.

Исходя из описанного выше, такие команды прерывания (а часто и соответствующие им процедуры) обычно называют *системными вызовами* (системными функциями операционной системы), а библиотека стандартных процедур – Базовой системой процедур ввода/вывода (английское сокращение – BIOS). Параметры для таких процедур обычно передаются на регистрах, т.е. для системных вызовов не выполняются стандартные соглашения о связях.

В качестве примера рассмотрим системный вызов `int 21h`, который реализует многие операции ввода/вывода. Так, для вывода строки текста на экран в качестве параметров следует передать номер конкретного действия на регистре `ah` (для вывода строки `ah=9`) и адрес начала выводимой строки на регистрах `ds:dx` (строка должна кончатся символом '\$'). Исходя из этого на место нашей макрокоманды вывода строки текста

```
outstr
```

можно подставить команды

```
mov ah, 9
int 21h
```

В качестве примера опишем на Ассемблере процедуру, использующую системный вызов. Эта процедура при её вызове выдаёт звуковой сигнал:

```
Beep proc
push ax
push dx
mov al, 7; символ-звуковой сигнал
mov ah, 02h; номер функции вывода символа
int 21h; системный вызов
pop dx
pop ax
ret
Beep endp
```

Можно заметить, что наша процедура `Beep` при своём вызове выполняет те же действия, что и макрокоманда `outch 7`.

В дальнейшем мы познакомимся ещё с одним важным назначением системных вызовов при изучении мультипрограммного режима работы ЭВМ.

Процедуры обработки прерываний реализуют особый стиль программирования, их иногда называют процедурами обратного вызова (*call back procedure*) или процедурами-демонами. Такая процедура при своей *инициализации* (размещении в памяти) оставляет в определённом месте адрес своего начала. Далее вызов этой процедуры производится при возникновении соответствующих условий путём (дальнего) косвенного перехода на эту процедуру.

В качестве примера рассмотрим расчёт платы за междугородний телефонный разговор, при котором за каждую новую минуту разговора к общей сумме прибавляется некоторая величина – тариф за минуту разговора с данным городом. При наступлении льготного времени (обычно ночью и в выходные дни) срабатывает будильник (специальная системная программа-обработчик прерываний от встроенного в ЭВМ таймера), который вызывает процедуру-демона пересчёта всех тарифов. Заметим, что в некоторые языки высокого уровня включены аналогичные возможности, например, в языке С можно писать так называемые функции-реакции на сигналы, о чём Вы узнаете в следующем семестре в курсе "Системное программное обеспечение".

В заключении нашего по необходимости краткого рассмотрения прерываний заметим, что появление в компьютерах системы прерываний было, несомненно, одним из важнейших событий в развитии архитектуры вычислительных машин. Недаром появившиеся компьютеры с системой прерываний стали относить к следующему, третьему поколению ЭВМ. Подробнее об этом можно прочитать в книге [3].

9. Дополнительные возможности Ассемблера.

9.1. Строковые команды.

Сейчас мы рассмотрим последний полезный для понимания архитектуры нашего компьютера формат команд память-память (формат *SS*). До сих пор для работы с переменными в оперативной памяти мы использовали формат команд регистр-память (или память-регистр), при этом один из аргументов находился на регистре центрального процессора. Это не всегда удобно, если мы не предполагаем больше никаких операций с выбранным на регистр операндом, тогда его пересылка из памяти на регистр оказывается лишним действием.¹ Именно для таких случаев и предназначены для таких случаев и предназначены команды формата память-память. В архитектуре нашего компьютера такие команды относятся к так называемым строковым командам (мы скоро поймём, почему они так называются).

Строковые команды хорошо использовать для обработки элементов массивов. Массивы коротких беззнаковых целых чисел могут трактоваться как строки (цепочки) символов, отсюда и название – *строковые* или *цепочечные* команды. При выполнении строковой команда в цикле получается удобный способ обработки массивов, элементами которых являются короткие или длинные целые числа.

Знакомство со строковыми командами начнём с команд пересылки байта (**movsb**) или слова (**movsw**) с одного места памяти в другое. Эти команды различаются только битом размера операнда *w* в коде операции. С битом *w* мы познакомились при изучении форматов команд регистр-регистр и регистр-память, *w=0* для операндов-байтов и *w=1* для операндов-слов. Команды **movsb** и **movsw** не имеют явных операндов, их оба операнда *op1* и *op2* формата *m8* (для *w=0*) и *m16* (для *w=1*) заданы неявно (по умолчанию).

Выполнение команд **movsb** и **movsw** существенно зависит от так называемого флага направления *DF* из регистра флагов *FLAGS*. Для смены значения этого флага можно использовать команды **cld** (для операции *DF:=0*), и **std** (для операции *DF:=1*). Для более компактного описания правил выполнения команд **movsb** и **movsw** введём следующие условные обозначения:

$$\delta = (w+1) * (-1)^{DF}; \quad \varphi(r16) = \{ r16 := (r16 + \delta) \bmod 2^{16} \}$$

Как видим, δ может принимать значения ± 1 и ± 2 , а оператор φ меняет величину регистра на значение δ . В этих обозначениях команды **movsb** и **movsw** выполняются по правилу:

$$\langle es, di \rangle := \langle ds, si \rangle; \quad \varphi(di); \quad \varphi(si)$$

Таким образом, неявный операнд *op1* находится в памяти по адресу, заданному регистровой парой $\langle es, di \rangle$, а операнд *op2* – по адресу $\langle ds, si \rangle$, т.е. операнды находятся в *разных* сегментах памяти.

Для того чтобы лучше понять логику работы описанных выше команд, рассмотрим задачу пересылки массива целых чисел с одного места оперативной памяти в другое. На языке Паскаль такая задача решается просто, например:

```
Var A,B: array[1..10000] of integer;
```

```
  . . .
  B := A;
```

Для языка Турбо-Паскаль, правда, это только частный случай задачи пересылки массива, так как массивы *A* и *B* будут располагаться в *одном* сегменте данных. Более общий случай пересылки массива на Паскале можно реализовать, например для массивов символов, в таком виде:

```
Const N = 50000;
Type Mas = Array[1..N] of char;
Var A,B: ↑Mas;
  . . .
  New(A); New(B);
  . . .
```

¹ Точнее, лишним будет только использование *адресуемого* регистра центрального процессора (*ax*, *bx* и т.д.). Как мы знаем, двухадресные команды формата память-память **КОП op1, op2** обязательно требует для своего выполнения использования служебного регистра центрального процессора, (мы обозначали этот регистр *R1*) и выполняются по схеме:

$$R1 := op1; \quad R1 := R1 \text{ КОП } op2; \quad op1 := R1.$$

$B \uparrow := A \uparrow;$

В этом примере динамические переменные (массивы символов, для Ассемблера, как мы знаем, это массивы коротких беззнаковых целых чисел) обязательно будут располагаться в *разных* сегментах памяти (понять это!). А теперь рассмотрим реализацию похожей задачи пересылки массива из одного места памяти в другое на языке Ассемблер (наши два массива будут не динамическими, а статическими, но так же располагаться в *разных* сегментах памяти). Сначала опишем два сегмента данных, в которых будут располагаться наши массивы А и В:

```

N      equ    50000
D1     segment
      . . .
A      db     N dup (?)
      . . .
D1     ends
D2     segment
      . . .
B      db     N dup (?)
      . . .
D2     ends

```

На начало сегмента D1 установим сегментный регистр ds, а на начало сегмента D2 – сегментный регистр es, тогда фрагмент программы для реализации оператора присваивания $B := A$ может, например, иметь такой вид:

```

Code segment
assume cs:Code, ds:D1, es:D2, ss:Stack
Start: mov ax, D1
      mov ds, ax
      mov ax, D2
      mov es, ax
      . . .
      mov si, offset A
      mov di, offset B
      mov cx, N

```

<pre> jcxz L1 L: mov al, [si] mov es:[di], al inc si inc di loop L L1: . . . </pre>
--

Оценим сложность нашего алгоритма пересылки массива. За единицу измерения примем обмен данными или командами между центральным процессором и оперативной памятью. В нашем случае сложность алгоритма пересылки массива равна $7 * N$, где N – это длина массива (N чтений элементов массива, N записей в память, $5 * N$ раз считать в цикле команды из памяти в устройство управления).

Как видим, для пересылки целого числа из одного места памяти в другое нам понадобились две команды

```

mov al, [si]
mov es:[di], al

```

так как написать одну команду

```

mov byte ptr [si], es:[di]

```

нельзя – она требует несуществующего формата команды пересылки `mov m8, m8`. Здесь, однако, хорошо подходит наша новая команда пересылки короткого целого числа `movsb`, с её помощью заключённый в рамку фрагмент программы можно записать в виде:

<pre> jcxz L1 cld </pre>

```
L:  movsb
    loop L
```

Теперь в нашем цикле пересылки массива осталось всего две команды, следовательно сложность нашего алгоритма снизилась до $4*N$ операций обмена. Для дальнейшего ускорения выполнения таких циклов в язык машины была включена специальная команда цикла **rep**, которая называется *префиксом повторения*. Она похожа на команду **loop**, но не имеет явного операнда – метки перехода на начало цикла. Эта метка не нужна, так как в теле цикла **rep** может находиться только *одна*, непосредственно следующая за ней команда, другими словами, пара команд

```
rep <строковая команда>
```

выполняется по схеме

```
while cx<>0 do begin
    dec (cx); <строковая команда>
end;
```

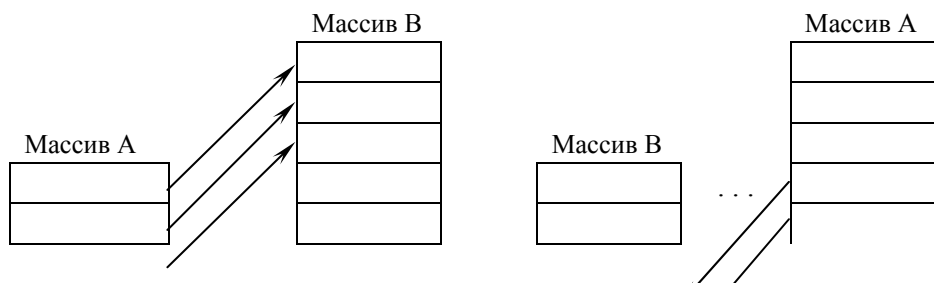
С использованием этой новой команды цикла заключенный в рамку фрагмент нашей программы пересылки массива можно записать уже совсем кратко в виде:

```
cld
rep movsb
```

Заметим, что хотя **rep** и является служебным словом (кодом операции), но его часто пишут на месте метки (в качестве первого поля предложения Ассемблера), так как служебное слово нельзя спутать с именем, заданным пользователем. Пара команд **rep movsb** является тесно связанной, они *вместе* выбираются на регистр команд центрального процессора, так что в цикле пересылки массива нет необходимости обращаться в память *за командами*. Теперь сложность нашего алгоритма снизилась до теоретического минимума в $2*N$ операций, т.е. это позволило значительно поднять эффективность пересылки массива.¹

Разберёмся теперь с назначением флага направления DF. Отметим сначала, что этот флаг позволяет производить пересылку массива в *прямом* направлении (от первого элемента к последнему) при значении DF=0 и в *обратном* направлении (от последнего элемента массива к его первому элементу) при DF=1, отсюда и название флага – флаг направления пересылки массива.

Пересылка массива в обратном направлении – не прихоть программиста, а часто единственный способ *правильного* присвоения значений массивов. Правда следует сразу сказать, что флаг направления влияет на правильное присваивание одному массиву значения другого массива только в том случае, если эти массивы *перекрываются* в памяти (т.е. один массив полностью или частично занимает то же место в памяти, что и второй массив). В качестве примера на рис. 9.1 показано два случая перекрытия массивов А и В в памяти. Из этого рисунка видно, что для случая 9.1 а) необходима пересылка в *прямом* направлении с флагом DF=0, а для случая 9.1 б) правильный результат присваивания массивов получается при *обратном* направлении пересылки элементов массива с флагом DF=1.



¹ Такое значительное уменьшение сложности алгоритма верно только для младших моделей нашего семейства. В старших моделях появилась специальная быстродействующая область памяти – кэш, в которую *автоматически* помещаются, в частности, последние выполняемые команды. Таким образом, весь наш первоначальный цикл пересылки из 5 команд будет находиться в этой кэш-памяти, скорость чтения из которой примерно на порядок больше скорости чтения из оперативной памяти. Другими словами, обращения в оперативную память *за командами* при выполнении цикла пересылки массива в старших моделях тоже производиться не будет, и сложность алгоритма также приближается к $2*N$ обменов. Более подробно с памятью типа кэш Вы познакомитесь в курсе "Системное программное обеспечение".

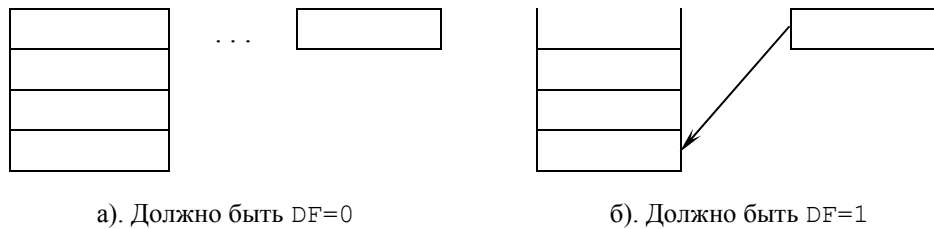


Рис. 9.1. Два случая перекрытия массивов в памяти при пересылке.

Перекрытие массивов при пересылке часто встречается в практике программирования. Типичным примером является работа текстового редактора, когда при операции вставки и удаления фрагментов текста приходится раздвигать или сжимать редактируемый текст.

Упражнение. Определите, какие значения должен иметь флаг направления DF при операции вставки и при операции удаления участка редактируемого текста.

Продолжим изучение строковых команд. Команды сравнения двух операндов **cmpsb** и **cmpsw** имеют тот же формат память-память, что и команды **movsb** и **movsw**. Команды **cmpsb** и **cmpsw** выполняются по схеме:

```
cmp <ds,si>,<es,di>; φ(di); φ(si)
```

т.е. производится сравнение между собой двух коротких или длинных целых чисел и соответствующим образом устанавливаются флаги (так же, как при выполнении команды сравнения).

Как мы знаем, команды сравнения необходимы для работы команд условного перехода. С командами сравнения **cmpsb** и **cmpsw** удобно использовать команды-префиксы повторения **repe/repz** и **repne/repnz**. Эти команды похожи на команду **rep**, но обеспечивают возможность досрочного выхода из цикла по значению флага ZF=0 (для команд **repe/repz**) и ZF=1 (для команд **repne/repnz**).

В качестве примера рассмотрим следующую задачу. Как мы знаем, строки текста можно сравнивать между собой не только на равенство и неравенство, но и с помощью других операций отношения (больше, меньше и т.д.). При этом строки считаются упорядоченными в так называемом *лексикографическом* порядке (а по-простому – как в словаре). С помощью строковых команд и префиксов повторения операцию сравнения двух строк можно так реализовать на Ассемблере (правда, здесь строки одинаковой длины, сравнение строк разной длины реализуется более сложно):

```
N      equ    20000
Data   segment
X      db     N dup (?)
Y      db     N dup (?)
      .      .      .
Data   ends
Code   segment
      assume cs:Code,ds:Data,es:Data,ss:Stack
      .      .      .
      mov    cx,N
      cld
      lea   si,X
      lea   di,Y
repe   cmpsb
      je    EQ; Строки X и Y равны
      jb   LT; Строка X меньше Y
      ja   GT; Строка X больше Y
```

В нашем примере сравниваемые строки для простоты расположены в одном сегменте (сегменте данных). Как видим, основная часть работы – последовательное сравнение в цикле символов двух строк до первых несовпадающих символов или до конца строк – выполняется двумя тесно связанными командами **repe** **cmpsb**.

Остальные строковые команды имеют формат регистр-память, но они тоже ориентированы на задачи обработки строк (массивов) коротких и длинных целых чисел. Команды *сканирование строки* являются командами *сравнения* и, при использовании в цикле, хорошо подходят для поиска в строке

заданного короткого (**scasb**) или длинного (**scasw**) целого значения. Эти команды отличаются только битом размера аргументов w и имеет два *неявных* операнда $op1$ и $op2$, где $op1=al$ для $w=0$, и $op1=ax$ для $w=1$, а второй неявный операнд $op2=<es, di>$ является соответственно байтом или словом. Если обозначить буквой r соответственно регистры al или ax , то схему выполнения команд сканирования строки можно записать так:

```
cmp r,<es,di>; φ(di)
```

Для иллюстрации использования команды сканирования напомним фрагмент программы для реализации следующей задачи: в массиве длинных целых чисел найти номер *последнего* нулевого элемента (пусть элементы нумеруются с единицы). Если в массиве нет нулевых элементов, то будем в качестве ответа выдавать номер ноль.

```
N      equ    20000
D      segment
      . . .
X      dw     N dup (?)
      . . .
D      ends
C      segment
      assume cs:C,ds:D,es:D,ss:Stack
Start:mov    ax,D
      mov    ds,ax
      mov    es,ax
      . . .
      mov    cx,N; Число элементов
      sub    ax,ax; Образец для поиска=0
      lea   si,X+2*N-2; Последний элемент
      std   ; Обратный просмотр
repne scasw
      jnz   NoZero; Нет нулевых элементов
      inc  cx; Номер последнего нулевого
NoZero:outword cx
```

Заметим, что выход из нашего цикла возможен при попадании на нулевой элемент массива, при исчерпании счётчика цикла, а также при совпадении обоих этих условий. Следовательно, после команд **repne** **scasw** необходимо проверить, имел ли место случай просто выхода из цикла без нахождения нулевого элемента, что мы и сделали командой условного перехода **jnz NoZero**.

Следующими рассмотрим команды *загрузки* элемента строки, которые являются командами *пересылки* и, при использовании в цикле, хорошо подходят для эффективной последовательной загрузки на регистр коротких (**lods b**) или длинных (**lods w**) элементов целочисленного массива. Эти команды отличаются только битом размера аргументов w , и имеют два *неявных* операнда $op1$ и $op2$, где $op1=al$ для $w=0$, и $op1=ax$ для $w=1$, а второй неявный операнд $op2=<ds, si>$ является соответственно байтом или словом. Если обозначить буквой r регистры al или ax , то схему выполнения этих команд можно записать так:

```
mov r,<ds,si>; φ(si)
```

В качестве примера использования команды загрузки напомним фрагмент программы для реализации следующей задачи: найти сумму тех элементов *беззнакового* массива коротких целых чисел, значения которых больше 100. Если в массиве нет таких элементов, то будем в качестве ответа выдавать число ноль.

```
N      equ    10000
D      segment
      . . .
X      db     N dup (?)
      . . .
D      ends
C      segment
      assume cs:C,ds:D,ss:Stack
```

```

Start:mov  ax,D
      mov  ds,ax
      . . .
      mov  cx,N; Число элементов
      sub  dx,dx; Сумма=0
      lea  si,X; Адрес первого элемента
      cld  ; Прямой просмотр
L:    lodsb
      cmp  al,100
      jbe  NoSum
      add  dl,al; Суммирование
      adc  dh,0; Прибавление CF
NoSum:loop L
      outword dx

```

При суммировании коротких целых чисел мы получаем в качестве результата длинное целое число на регистре dx.

Последними в этом формате SS рассмотрим команды *сохранения* элемента строки, которые являются командами *пересылки* и, при использовании в цикле, хорошо подходят для эффективного присваивания всем элементам массива заданного короткого (**stosb**) или длинного (**stosw**) целого значения. Эти команды отличаются только битом размера аргументов *w*, и имеют два *неявных* операнда *op1* и *op2*, где *второй* неявный операнд *op2=al* для *w=0*, и *op2=ax* для *w=1*, а *первый* неявный операнд *op1=<es,di>* является соответственно байтом или словом. Если обозначить буквой *r* регистры *al* или *ax*, то схему выполнения команд можно записать так:

```
mov <es,di>,r; φ(di)
```

В качестве примера использования команды сохранения напомним фрагмент программы для присваивания всем элементам массива длинных целых чисел значения единица.

```

N      equ  30000
D      segment
      . . .
X      dw   N dup (?)
      . . .
D      ends
C      segment
      assume cs:C,ds:D,es:D,ss:Stack
Start:mov  ax,D
      mov  ds,ax
      mov  es,ax
      . . .
      mov  cx,N; Число элементов
      mov  ax,1; Присваиваемое значение
      lea  di,X; Адрес первого элемента
      cld  ; Прямой просмотр
rep    stosw

```

Рассмотрим ещё один пример. Напишем фрагмент программы для решения задачи присваивания всем элементам знакового массива целых чисел *Y* абсолютных значений соответствующих им элементов массива *X*, т.е. $Y := \text{abs}(X)$ (учтите, что этот оператор присваивания – это только иллюстрация, так в Паскале написать нельзя).

```

N      equ  5000
D      segment
X      dw   N dup (?)
Y      dw   N dup (?)
Diagn  db   'Большое значение в X!$'
      . . .
D      ends

```



```

C      segment
      assume cs:C,ds:D,es:D,ss:Stack
Start:mov  ax,D
      mov  ds,ax
      mov  es,ax
      . . .
      mov  cx,N
      cld  ; Прямой просмотр
      lea  si,X
      lea  di,Y
L:     lodsw
      cmp  ax,0
      jge  L1
      neg  ax
      jno  L1
      lea  dx,Diagn
      outstr
      finish
L1:    stosw
      loop L

```

В приведённом примере массивы X и Y находятся в одном сегменте, поэтому регистры ds и es имеют одинаковые значения. Так как не у каждого отрицательного числа есть соответствующее ему абсолютное значение, то при обнаружении такой ситуации выдаётся аварийная диагностика, и выполнение программы прекращается.

На этом мы закончим наше краткое изучение строковых команд и перейдём к следующему классу команд – логическим командам.

9.2. Логические команды.

Все логические команды рассматривают свои операнды как упорядоченные наборы битовых значений. В таком наборе может содержаться 8 бит или 16 бит, в зависимости от размера операнда.¹ Бит со значением 1 может трактоваться как логическое значение **True**, а нулевой бит – как логическое значение **False**.

Сначала рассмотрим двухадресные логические команды, имеющие такой общий вид

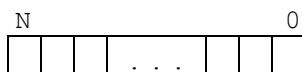
КОП op1,op2

Ниже приведены допустимые операнды таких команд:

op1	op2
r8	r8, m8, i8
m8	r8, i8
r16	r16, m16, i16
m16	r16, i16

Как видно, у этих команд допускаются такие же операнды, как и у команд сложения, вычитания и сравнения.

Будем нумеровать биты в операндах логических команд так же, как и в регистрах, от нуля до некоторого числа N. Число N равно 7 для коротких (байтовых) операндов и равно 15 для длинных (размером в слово):



Таким образом, операнды логической команды можно рассматривать как массивы логических элементов, проиндексированных от 0 до N. На языке Паскаль такие массивы можно, например, описать в следующем виде

```
Var op1,op2: array[0..N] of Boolean;
```

Схему выполнения команды *логического умножения*

¹ Для двух команд сдвига в этом наборе будет 9 или 17 бит, это мы далее отметим особо.

and op1,op2

на языке Паскаль можно записать в виде

```
for i:=0 to N do op1[i]:=op1[i] and op2[i]
```

Схему выполнения команды *логического сложения*

or op1,op2

на языке Паскаль можно записать в виде

```
for i:=0 to N do op1[i]:=op1[i] or op2[i]
```

Схему выполнения команды *неэквивалентности* (её часто называют также командой сложения по модулю 2)

xor op1,op2

на языке Паскаль можно записать в виде

```
for i:=0 to N do op1[i]:=op1[i] <> op2[i]
```

Команда *логического тестирования*

test op1,op2

выполняется точно так же, как и команда логического умножения, но без записи результата на место первого операнда, т.е. как и у команды сравнения с кодом **cmp** её единственным результатом является установка флагов.

Все перечисленные выше логические команды устанавливают флаги так же, как команда вычитания, например, флаги CF и OF будут равны нулю (никакого переноса и переполнения, конечно, нет), во флаг знака SF переносится левый бит результата и т.д. Но, как мы вскоре увидим, интерес для программиста представляет здесь только флаг нулевого результата ZF, который, как обычно, равен единице (поднят) для полностью нулевого результата, и равен нулю (опущен), если в результате есть хотя бы один ненулевой бит.

Следует заметить, что мы использовали цикл языка Паскаль только для более строгого описания работы логических команд. Не следует понимать это слишком буквально: на самом деле логические команды на современных ЭВМ выполняют операции над битами своих операндов одновременно (чаще всего за один такт работы центрального процессора), а не в последовательном цикле.¹

9.3. Команды сдвига.

Команды сдвига предназначены для сдвига (изменения индексов) битов в своём операнде. Операнд при этом можно рассматривать как битовый вектор, элементы которого пронумерованы от 0 до N так же, как и для рассмотренных выше логических команд. Команды сдвига имеют формат

коп op1,op2

где op1 может быть r8, m8, r16 или m16, а op2 – иметь значение единицы или быть коротким регистром c1.² Мы рассмотрим сначала команды сдвига вида **коп op1,1**, а затем обобщим их на случай команд вида **коп op1,c1**.

Команда сдвига операнда на один бит *влево* имеет показанный ниже вид

shl op1,1

и её выполнение можно так описать в виде цикла на Паскале:

```
CF:=op1[N]; for i:=N downto 1 do op1[i]:=op1[i-1]; op1[0]:=0
```

Как видим, при сдвиге битового вектора на одну позицию влево самый левый бит не теряется, а попадает во флаг переноса CF, а на освободившееся справа место записывается нулевой бит. Все команды сдвига устанавливают флаги по значению результата по тем же правилам, что и команды сложения и вычитания. Например, флагу – признаку отрицательного результата SF приваивается самый левый бит результата, т.е. SF:=op1[N]. Однако среди флагов, изменяемых командами сдвигов, в практическом программировании имеет смысл рассматривать только флаг переноса CF и

¹ Для любознательных отметим, что для описания работы логических команд более подходит оператор цикла параллельного Фортрана, например: **for all i do** op1[i]:=op1[i]<>op2[i], который предписывает выполнить тело цикла *одновременно (параллельно)* для всех значений параметра цикла i.

² В старших моделях нашего семейства у команд сдвига допускается также второй операнд формата i8. Мы в своих примерах этим форматом пользоваться не будем.

флаг нуля ZF (он устанавливается в единицу, если, как обычно, получается полностью нулевой вектор-результат).

Команда сдвига влево часто называется *логическим* сдвигом влево, у кода операции этой команды есть синоним **sal**, который называется *арифметическим* сдвигом влево. Логический и арифметический сдвиги выполняются одинаково, а различие в их названиях будет понятно из дальнейшего изложения.

В том случае, если мы будем трактовать операнд команды сдвига влево на один бит как *целое число*, то результатом сдвига является умножение этого числа на два. При этом результат умножения получается правильным, если во флаг переноса CF попадает *незначащий* бит результата. Таким образом, для *беззнакового* числа при правильном умножении на 2 должно быть CF=0, а для *знакового* операнда результат получается правильным только тогда, когда значение флага переноса совпадает со знаковым (крайним слева) битом результата, т.е. после выполнения команды сдвига справедливо равенство CF=op1[N].

Рассмотрим теперь команды сдвига на один разряд *вправо*. По аналогии со сдвигом на один разряд влево, сдвиг на один разряд вправо можно трактовать как *деление* целого числа на два. Однако, так как деление на два должно выполняться по разным правилам для знаковых и беззнаковых целых чисел (вспомним *различные* команды **div** и **idiv**), то существуют две *разные* команды сдвига операнда на один бит вправо. Команда *логического* сдвига на один разряд вправо

```
shr op1,1
```

выполняется по правилу:

```
CF:=op1[0]; for i:=0 to N-1 do op1[i]:=op1[i+1]; op1[N]:=0
```

Эта команда эквивалентна делению на два *беззнакового* целого числа, результат при этом всегда получается правильным. Делению на два *знакового* целого числа эквивалентна команда *арифметического* сдвига операнда на один бит вправо

```
sar op1,1
```

она выполняется по правилу:

```
CF:=op1[0]; for i:=0 to N-1 do op1[i]:=op1[i+1]
```

Как видим, крайний левый бит аргумента при арифметическом сдвиге вправо *не меняется*. Особым здесь является случай, когда операнд равен минус единице, тогда операция деления этого операнда на два *не эквивалентна* операции арифметического сдвига вправо на один бит. Легко проверить, что $(-1) \text{ div } 2 = 0$, а результат арифметического сдвига вправо операнда со значением -1 снова даёт -1 (т.е. **sar** $-1, 1 = -1$).

Заметим далее, что, как и "настоящие" команды деления, сдвиг вправо даёт *два* результата: частное на месте своего операнда и остаток от деления на 2 во флаге CF. Действительно, легко видеть, что

```
CF:=op1 mod 2      для беззнакового операнда и
```

```
CF:=abs(op1 mod 2) для знакового операнда.
```

Таким образом, для проверки того, является ли целое число X *нечётным*, можно использовать следующие две команды

```
shl X,1
```

```
jc ODD; Нечётное X
```

Программисты, однако, не любят этот способ проверки на нечётность, так как при этом портится операнд X. Лучше проверять целое число X на нечётность двумя командами

```
test X,1
```

```
jne ODD; Нечётное X
```

Следующая группа команд сдвига – так называемые *циклические* сдвиги. Эти команды рассматривают свой операнд как замкнутый в кольцо: после бита с номером N располагается бит с номером 0, а перед битом с номером 0 – бит с номером N. Ясно, что при циклическом сдвиге операнд сохраняет все свои биты, меняются только номера этих битов. Команда циклического сдвига *влево*

```
rol op1,1
```

выполняется по правилу

```
shl op1,1; op1[0]:=CF
```

Команда циклического сдвига *вправо*

```
ror op1,1
```

выполняется по правилу

```
shr op1,1; op1[N]:=CF
```

Команда циклического сдвига *через флаг переноса* включают в кольцо сдвигаемых битов дополнительный бит – флаг переноса CF, который включается между битами с номерами 0 и N. Таким образом, в сдвиге участвуют N+1 бит. Команда циклического сдвига *влево через флаг переноса*

```
rcl op1,1
```

выполняется по правилу

```
t:=CF; rol op1,1; op1[0]:=t
```

Здесь t – некоторая вспомогательная (временная) переменная.

Команда циклического сдвига *вправо через флаг переноса*

```
rcr op1,1
```

выполняется по правилу

```
t:=CF; ror op1,1; op1[N]:=t
```

Команды циклического сдвига в практике программирования используются редко – когда надо проанализировать биты операнда и в операнде можно изменять порядок этих битов.

Теперь нам осталось описать команды сдвига, вторым операндом которых служит регистр c1. Каждая такая команда (КОП – любой из кодов операций сдвигов)

```
КОП op1,c1
```

Выполняется по правилу

```
for c1:=c1 downto 1 do КОП op1,1; c1:=0
```

Таким образом, значение регистра c1 задаёт число разрядов, на которые *в цикле* происходит сдвиг операнда, после цикла, как обычно, счётчик цикла (в данном случае регистр c1) обнуляется. Ясно, что задавать сдвиги более чем на N разрядов не имеет большого смысла.

Главное назначение логических команд – обрабатывать отдельные биты и группы битов в байтах и словах. Разберём несколько примеров использования логических команд в программировании на Ассемблере. Сначала составим фрагмент программы, в котором подсчитывается и выводится число битов со значением "1" во внутреннем машинном представлении переменной X размером в слово:

```
X      dw      ?
      . . .
      inint X
      mov   ax,X
      sub   cx,cx; число "1"
L:     cmp   ax,0
      jz    Pech
      shl   ax,1
      adc   cx,0; cx:=cx+CF
      jmp   L
Pech: outint cx
```

Заметим, что операция подсчёта числа битов машинного слова со значением "1" является весьма важной в архитектуре некоторых ЭВМ. В качестве примера рассмотрим отечественную ЭВМ третьего поколения БЭСМ-6, которая производилась в 70-х годах прошлого века [3]. В этой ЭВМ сигналы прерывания устанавливали в "1" биты в специальном 48-разрядном *регистре прерываний* (каждый бит соответствовал своему типу прерывания). В этой архитектуре существовала специальная машинная команда для подсчёта количества "1" в своём аргументе, что позволяло быстро определить число ещё необработанных сигналов прерывания.

Рассмотрим теперь использование логических команд при обработке *упакованных* структур данных. Пусть, например, на языке Паскаль дано описание упакованного массива с элементами ограниченного целого типа:

```
Const N=10000;
Var   A: packed array[0..N] of 0..15;
```

X: byte;

Напомним, что служебное слово **packed** есть *рекомендация* Паскаль-машине по возможности более компактно хранить данные, даже если это повлечёт за собой увеличения времени доступа к этим данным по чтению и записи. Многие Паскаль-машины могут и "не прислушаться" к этим рекомендация (игнорировать их).

Рассмотрим фрагмент программы на Ассемблере, который работает с описанным выше упакованным массивом A. Реализуем, например, оператор присваивания $X:=A[i]$. Каждый элемент массива требует для своего хранения 4 бита памяти, так что будем в одном байте хранить два последовательных элемента массива A:

```
N      equ    10000
Data  segment
A      db    N/2 dup (?); N/2 ≡ N div 2
X      db    ?
      . . .
Data  ends
      . . .
      mov    bx,i
      xor    ax,ax; ax:=0
      shr    bx,1
      mov    al,A[bx]; в al два элемента
      jc    L1; i-ый элемент - правый
      mov    cl,4; число сдвигов
      shr    al,cl
L1:    and    al,1111b; выделение A[i]
      mov    X,al
```

Сначала мы разделили индекс i на 2 и определили тот байт массива A, в котором находится пара элементов, один из которых нам нужен. На положение нужного нам элемента в байте указывает остаток от деления индекса i на 2: если остаток равен нулю (т.е. i чётное), то элемент расположен в *левых* четырёх битах байта, иначе – в *правых*. Для выделения нужного элемента, который занимает в байте только 4 бита из 8, мы использовали команду логического умножения `and al,1111b`, где второй операнд задан для удобства понимания смысла команды в виде двоичного числа, на что указывает буква b в конце этого числа.

Использование команды логического умножения для выделения нужной нам части байта или слова, и обнуление остальной части является типичной для программирования на Ассемблере. При этом константа, используемая для выделения нужной части (у нас это `00001111b`), называется *маской выделения* или просто маской. Таким образом, мы поместили элемент массива X, который занимает 4 бита, в регистр `al` и дополнили этот элемент до 8 бит нулями слева. Заметим, что это не изменило величины нашего элемента, так как он беззнаковый (0..15).

Наш простой пример показывает, что при работе с упакованными данными скорость доступа к ним уменьшается в несколько раз и программист должен решить, стоит ли это достигнутой нами экономии памяти (в нашем примере мы сэкономили 5000 байт оперативной памяти). Обычно это типичная ситуация в программировании: выигрывая в скорости обработки данных, мы проигрываем в объёме памяти для хранения этих данных, и наоборот. Иногда, по аналогии с физикой, это называют "законом рычага", который гласит, что, выигрывая в силе, мы проигрываем в длине перемещения конца рычага, и наоборот.

Упражнение. Реализуйте для нашего примера оператор присваивания $A[i]:=X$.

В качестве последнего примера использования логических команд рассмотрим реализацию на Ассемблере некоторых операций языка Паскаль над *множествами*. Пусть на Паскале есть описания двух символьных множеств X и Y, а также символьной переменной Sym:

```
Var X,Y: set of char; Sym: char;
```

Напишем на Ассемблере фрагмент программы для реализации операции объединения двух множеств:

```
X := X + Y;
```

Каждое такое множество будем хранить в памяти в виде 256 последовательных битов, т.е. 32 байт или 16 слов. Бит, расположенный в позиции I принимает значение 1, если символ с номером I присутствует во множестве, иначе этот бит имеет значение 0. Множества X и Y можно так описать на Ассемблере:

```
Data Segment
. . .
X dw 16 dup (?)
Y dw 16 dup (?)
. . .
Data ends
```

Тогда операцию объединения двух множеств можно реализовать, например, таким фрагментом на Ассемблере:

```
mov cx,16
xor bx,bx
L: mov ax,Y[bx]
or X[bx],ax
add bx,2
loop L
```

В заключение рассмотрим условный оператор языка Паскаля, включающий операцию отношения **in** (символ Sym входит во множество X):

```
if Sym in X then goto L;
```

На Ассемблере этот оператор можно, например, реализовать в виде такого фрагмента программы:

```
X db 32 dup (?); 32*8=256 битов
Sym db ?
. . .
mov al,Sym
mov cl,3
shr al,cl
xor bx,bx
mov bl,al; Индекс нужного байта в X
mov al,X[bx]; Байт с символом Sym
mov cl,Sym
and cl,111b; Позиция символа Sym в байте
shl al,cl; наш бит в al - крайний слева
shl al,1; Нужный бит в CF
jc L
```

Сначала, используя команду сдвига на 3 (это, как мы знаем, аналогично делению на 8), на регистр bx заносим индекс того байта в X , в котором находится бит, соответствующий символу Sym в множестве X . Затем этот байт выбирается на регистр al , а на регистр cl помещаем три последних бита номера символа Sym в алфавите – это и будет номер нужного нам бита в выбранном байте (правда, биты при этом нумеруем *слева направо*, начиная с нуля). После этого первой командой сдвига перемещаем нужный нам бит в крайнюю левую позицию в байте, а следующей командой сдвига – во флаг переноса. Теперь осталось только сделать условный переход по значению этого флага.

На этом мы закончим рассмотрение логических команд.

10. Модульное программирование

Сейчас мы переходим к новой теме – модульному программированию. Модульное программирование предполагает особый способ разработки программы, которая при этом строится

из нескольких относительно независимых друг от друга частей – *модулей*. Модули могут писаться как на одном языке программирования, например, на Ассемблере, так и на разных языках, в этом случае говорят, что используется *многоязыковая система программирования*. Что такое система программирования мы строго определим несколько позже, а пока изучим общее понятие модульного программирования и *программного модуля*.

Мы уже знаем одно из полезных свойств такой программы, отдельные части (модули) которой написаны на разных языках программирования – это позволяет нам из программ на языках высокого уровня вызывать процедуры на Ассемблере. Познакомимся теперь со свойствами модульной программы, написанной на одном языке программирования (в нашем случае на Ассемблере).

Перечислим сначала те преимущества, которые предоставляет модульное программирование. Во-первых, как мы уже отмечали, это возможность писать модули на *разных* языках программирования. Во-вторых, модуль является естественной единицей локализации имён: как мы говорили, внутри модуля на Ассемблере все имена должны быть различны (уникальны),¹ что не очень удобно, особенно когда модуль большой по объёму или совместно пишется разными программистами. Как и в блоке программы на языке Паскаль, имена *локализованы* в модуле на Ассемблере и не видны из другого модуля, если только это не указано явно с помощью специальных директив.

Следующим преимуществом модульного программирования является локализация места ошибки: обычно исправление ошибки внутри одного модуля не влечёт за собой исправление других модулей (разумеется, это свойство будет выполняться только при *хорошем* разбиении программы на модули, с малым числом *связей* между модулями, о чём мы будем говорить далее). Это преимущество особенно сильно сказывается во время отладки программы. Например при внесении изменений только в один из нескольких десятков модулей программы, только он и должен быть заново проверен программой Ассемблером и переведён на язык машины.² Обычно говорят о *малом времени перекомпиляции* всей программы при исправлении ошибки в одном модуле, что сильно ускоряет процесс отладки всей программы.

Разумеется, за всё надо платить, у модульного программирования есть и свои слабые стороны. Во-первых, модули не являются совсем независимыми друг от друга: между ними существуют *связи*, то есть один модуль иногда может использовать переменные и программный код другого модуля. Необходимость связей между модулями естественно вытекает из того факта, что модули *совместно* решают одну общую задачу, при этом каждый модуль выполняет свою часть задачи. Связи между модулями на Ассемблере должны быть явно заданы при описании этих модулей.

Во-вторых, теперь перед счётом программы необходим особый этап *сборки* программы из составляющих её модулей. Этот процесс достаточно сложен, так как кроме собственно сборки программы из модулей, необходимо проконтролировать и установить все связи между модулями.³ Сборки программы из модулей производится специальной системной программой, которая называется *редактором внешних связей* между модулями.

В-третьих, так как теперь наш Ассемблер никогда не видит *всей* исходной программы одновременно, то, следовательно, и не может получить полностью готовую к счёту программу на машинном языке. Более того, в каждый момент времени он видит только *один модуль*, и не может проконтролировать, правильно ли установлены связи между модулями. Ошибка в связях теперь выявляется на этапе сборки программы из модулей, а иногда только на этапе счёта, если используется так называемое *динамическое* связывание модулей, обо всём этом мы будем говорить далее. Позднее обнаружение ошибок связи между модулями может существенно замедлить процесс отладки программы.

Несмотря на отмеченные недостатки, преимущества модульного программирования так велики, что сейчас это главный способ разработки программного обеспечения. Теперь мы начнём знакомиться с особенностями написания модульной программы на языке Ассемблера.

¹ Из этого правила совсем немного исключений, с большинством из них мы познакомимся при изучении *макросредств* языка Ассемблера.

² Точнее на *объектный* язык, о чём мы будем говорить далее.

³ Установление связей может быть отложено на этап счёта программы, о чём мы будем говорить позже при изучении динамического загрузчика.

10.1. Модульное программирование на Ассемблере.

Как мы уже говорили, программа на Ассемблере может состоять из нескольких модулей. Исходным (или входным) программным модулем на Ассемблере называется текстовый файл, состоящий из предложений языка Ассемблер и заканчивающийся специальной директивой с именем **end** – признаком конца модуля.

Среди всех модулей, составляющих программу, должен быть один и только один модуль, который называется *головным* модулем программы. Признаком головного модуля является параметр-метка у директивы **end** конца модуля, в учебниках такую метку часто называют именем *Start*, хотя это, как мы отмечали, несущественно и можно выбрать любое подходящее имя. Эта метка должна быть меткой *команды*, которая находится в одном из сегментов головного модуля. Именно этот сегмент по определению будет *кодowym* сегментом и содержать первую выполняемую команду всей программы. Перед началом счёта программы загрузчик установит на начало этого кодового сегмента регистр *CS*, а в счётчик адреса *IP* запишет смещение указанной метки начальной команды в сегменте кода.

Как уже отмечалось, модули не могут быть абсолютно независимыми друг от друга, так как решают разные части одной общей задачи, и, следовательно, хотя бы время от времени должны обмениваться между собой информацией. Таким образом, между модулями существуют *связи*. Говорят, что между модулями существуют связи *по управлению*, если один модуль может передавать управление (с возвратом или без возврата) на программный код в другом модуле. В архитектуре нашего компьютера для такой передачи можно использовать одну из команд перехода.

Кроме связей по управлению, между модулями могут существовать и связи по *данным*. Связи по данным предполагают, что один модуль может иметь доступ к областям памяти (переменным) в другом модуле. Частным случаем связи по данным является и использование одним модулем целочисленной *константы*, определённой в другом модуле (в Ассемблере такая константа может объявляться, например, директивой эквивалентности **equ**).

Связи между модулями реализуются в виде адресов, для нашей архитектуры это одно число (близкий адрес) или два числа (дальний адрес – значение сегментного регистра и смещения в сегменте).¹ Действительно, чтобы выполнить команду из другого модуля, а также считать или записать значение в переменную, нужно знать месторасположение (адрес) этой команды или переменной. Заметим, что численные значения связей между модулями (значения адресов) невозможно установить на этапе компиляции модуля, так как будущее расположение модулей в памяти во время счёта неизвестно на этапе компиляции.

Связи между модулями будем называть *статическими*, если численные значения этих связей (т.е. адреса) известны *до начала* счёта программы (до выполнения её первой команды). В отличие от статических, значения *динамических* связей между модулями становятся известны только во время счёта программы. Вскоре мы приведём примеры статических и динамических связей, как по данным, так и по управлению.

На языке Ассемблера связи между модулями задаются с помощью специальных директив. Директива

```
public <список имён модуля>
```

объявляет перечисленные в директиве имена *общедоступными*, т.е. разрешает использование этих имён в других модулях. В некоторых модульных системах программирования про такие имена говорится, что они *экспортируются* в другие модули.² В Ассемблере вместе с каждым именем экспортируется и его тип. Как мы уже знаем, для имён, использованных в директивах резервирования памяти, тип имени определяет длину области памяти, а для меток их тип равен -1 для близкой метки и -2 для дальней. Остальные имена (имена сегментов, имена констант в директивах эквивалентности и другие) имеют тип ноль. Тип имени в принципе позволяет проводить контроль использования этого имени в другом модуле. Все остальные имена модуля, кроме имён, перечисленных в директивах **public**, являются *локальными* и не видны извне (из других модулей).

¹ Это верно и для случая, когда один модуль использует константу, определённую в другом модуле, так как константа – это тоже целое число (непосредственное значение в поле адреса *i8* или *i16*).

² В некоторых языках имена по умолчанию считаются доступными из других модулей, если они описаны или объявлены в определённой части модуля. Например, в модуле на языке *C* общедоступны все имена, описанные *вне* функций, если про них явно не сказано, что это локальные имена модуля.

Экспортируемые имена одного модуля не становятся *автоматически* доступными в других модулях. Для получения доступа к таким именам этот другой модуль должен, с помощью специальной директивы, явно объявить о своём желании использовать общедоступные имена первого модуля. Это делается с помощью директивы

```
extrn <имя:тип>, . . . , <имя:тип>
```

В этой директиве перечисляются *внешние* имена, которые используются в этом модуле, но *не описаны* в нём. Внешние имена должны быть описаны и объявлены общедоступными в других модулях. Вместе с каждым внешним именем объявляется и тип, который должно иметь это имя в другом модуле. Проверка того, что это имя в другом модуле *на самом деле* имеет такой тип, может проводиться только на этапе сборки из модулей готовой программы, о чём мы будем говорить далее.

Таким образом, для установления связи между двумя модулями первый модуль должен *разрешить* использовать некоторые из своих имён в других модулях, а второй модуль – явно *объявить*, что он хочет использовать внутри себя такие имена. В языке Ассемблера общедоступные имена называются *входными точками* модуля, что хорошо отражает суть дела, так как только в эти точки возможен доступ к модулю извне (из других модулей). Внешние имена модуля называются *внешними адресами*, так как это адреса областей памяти и команд, а также значения констант в других модулях.

Все программы, которые мы писали до сих пор, на самом деле состояли из двух модулей, но один из них с именем **ioproc** мы не писали сами, он поставлялся нам в готовом виде. Этот второй модуль содержит процедуры ввода/вывода, к которым мы обращаемся с помощью наших макрокоманд (**inint**, **outint** и других). Теперь настало время написать программу, которая будет содержать два наших собственных модуля, и модуль **ioproc** (так как без ввода/вывода нам, скорее всего, не обойтись).

В качестве примера напишем программу, которая вводит массив *A* *знаковых* целых чисел и выводит сумму всех элементов этого массива. Ввод массива и вывод результатов будет выполнять головной модуль нашей программы, а подсчёт суммы элементов массива будет выполнять процедура, расположенная во втором модуле программы. Для иллюстрации использования связей между модулями мы не будем делать процедуру суммирования полностью со стандартными соглашениями о связях, она будет использовать *глобальные* имена для получения своих параметров, выдачи результата работы и диагностики об ошибке.

Текстовый файл, содержащий первый (головной) модуль нашей программы, мы, не долго думая, назовём `p1.asm`, а файл второго модуля с процедурой суммирования – `p2.asm`. Ниже приведён текст первого модуля.

```
; p1.asm
; Ввод массива, вызов внешней процедуры
include io.asm
St segment stack
    dw 64 dup (?)
St ends
N equ 1000
Data segment public
A dw N dup (?)
    public A,N; Входные точки
    extrn Summa:word; Внешняя переменная
Diagn db 'Переполнение!',13,10,'$'
Data ends
Code segment public
    assume cs:Code,ds:Data,ss:St
Start:mov ax,Data
    mov ds,ax
    mov cx,N
    sub bx,bx; индекс массива
L: inint A[bx]; Ввод массива A
    add bx,type A
    loop L
    extrn Sum:far; Внешнее имя
```

```

    call Sum; Процедура суммирования
    outint Summa
    newline
; А теперь вызов с ошибкой
    mov    A,7FFFh; Maxint
    mov    A+2,1; Для переполнения
    call   Sum
    outint Summa; Сюда возврата не будет
    newline
    finish ; Вообще-то не нужен
    public Error; Входная точка
Error:lea    dx,T
    outstr
    finish
Code ends
    end    Start; головной модуль

```

В нашем головном модуле три входные точки с именами A, N и Error и два внешних имени: Sum, которое имеет тип дальней метки, и Summa, которое имеет тип слова. Работу программы подробно рассмотрим после написания текста второго модуля с именем p2.asm.

```

Comment * модуль p2.asm
    Суммирование массива, контроль ошибок
    include io.asm не нужен - нет ввода/вывода
    Используется стек головного модуля
    В конечном end не нужна метка Start
*
Data segment public
Summa dw    ?
    public Summa; Входная точка
    extrn N:abs; Внешняя константа
    extrn A:word; Внешний адрес
Data ends
Code segment public
    assume cs:Code,ds>Data
    public Sum; Входная точка
Sum proc far
    push ax
    push cx
    push bx; сохранение регистров
    xor ax,ax; ax:=0
    mov cx,N
    xor bx,bx; индекс 1-го элемента
L: add ax,A[bx]
    jno L1
; Обнаружена ошибка
    pop bx
    pop cx
    pop ax
    extrn Error:near
    jmp Error
L1: add bx,type A
    loop L
    mov Summa,ax
    pop bx
    pop cx
    pop ax; восстановление регистров
    ret
Code ends
    end

```

Наш второй модуль не является головным, поэтому в его директиве **end** нет метки первой команды программы. Модуль `p2.asm` имеет три внешних имени `A`, `N` и `Error` и две входные точки с именами `Sum` и `Summa`. Так как второй модуль не производит никаких операций ввода/вывода, то он не подключает к себе файл `io.asm`. Оба наших модуля используют общий стек объемом 64 слова, что, наверное, достаточно, так как стековый кадр процедуры `Sum` невелик.

Разберём работу нашей программы. После ввода массива `A` головной модуль вызывает внешнюю процедуру `Sum`. Это *статическая связь модулей по управлению*, дальний адрес процедуры `Sum` будет известен головному модулю до начала счёта. Этот адрес будет расположен в формате `i32` на месте операнда `Sum` команды `call Sum`.

Между основной программой и процедурой установлены следующие (нестандартные) соглашения о связях. Суммируемый массив *знаковых* чисел расположен в головном модуле и имеет общедоступное имя `A`. Длина массива является общедоступной константой с именем `N`, описанной в головном модуле. Вычисленная сумма массива помещается в общедоступную переменную с именем `Summa`, описанную во втором модуле. Всё это примеры *статических* связей по данным. Наша программа не содержит динамических связей по данным, в качестве примера такой связи можно привести передачу параметра *по ссылке* в процедуру другого модуля. Действительно, адрес переменной становится известным процедуре только во время счёта программы, когда он передан ей вызывающей программой (обычно в стеке).

В том случае, если при суммировании массива обнаружена ошибка (переполнение), второй модуль передаёт управление на общедоступную метку с именем `Error`, описанную в головном модуле. Остальные имена являются локальными в модулях, например, обратите внимание, что в обоих модулях используется одинаковая метка с именем `L`.

Здесь необходимо отметить важную особенность использования внешних адресов. Рассмотрим, например, команду

```
L:    add    ax, A[bx]
```

во втором модуле. При получении из этого предложения языка Ассемблера машинной команды необходимо знать, по какому сегментному регистру базируется наш внешний адрес `A`. На это во втором модуле (а только его и видит во время перевода программа Ассемблера, первый модуль недоступен!) указывает *местоположение* директивы

```
extrn A:word; Внешний адрес
```

внутри второго модуля. Эта директива располагается в сегменте с именем `Data`, а директива

```
assume cs:Code, ds:Data
```

определяет, что во время счёта на этот сегмент будет установлен регистр `ds`. Следовательно, адрес `A` соответствует области памяти в том сегменте, на который указывает регистр `ds`.¹ Как видим, директива **assume** нам снова пригодилась.

Продолжим рассмотрение работы нашей программы. Получив управление, процедура `Sum` сохраняет в стеке используемые регистры (эта часть соглашения о связях выполняется), и накапливает сумму всех элементов массива `A` в регистре `ax`. При ошибке переполнения процедура восстанавливает значения регистров и передаёт управление на метку `Error` в головном модуле. В нашем примере второй вызов процедуры `Sum` специально сделан так, чтобы вызвать ошибку переполнения. Заметим, что переход на внешнюю метку `Error` – это тоже статическая связь по управлению, так как адрес метки известен до начала счёта. В то же время возврат из внешней процедуры по команде `ret` является *динамической* связью по управлению, так как конкретный адрес возврата в другой модуль будет помещён в стек только во время счёта программы.

Программа Ассемблера не в состоянии перевести каждый исходный модуль в готовый к счёту фрагмент программы на машинном языке, так как, во-первых, не может определить внешние адреса модуля, а, во-вторых, не знает будущего расположения сегментов модуля в памяти. Говорят, что

¹ Иногда говорят, что имя `A` *объявлено* в сегменте `Data` (правда термин "объявить имя" используется в основном в языках высокого уровня). Объявление имени в некотором сегменте, в отличие от *описания* этого имени, не требует выделения для переменной `A` памяти в сегменте. Для Ассемблера это является указанием (директивой) о том, что во время счёта программы данная переменная будет находиться в памяти именно этого сегмента.

Ассемблер переводит исходный модуль на специальный промежуточный язык, который называется *объектным языком*. Следовательно, программа Ассемблер преобразует входной модуль в объектный модуль. Полученный объектный модуль оформляется в виде файла, имя этого файла обычно совпадает с именем исходного файла на языке Ассемблер, но имеет другое расширение. Так, наши исходные файлы `p1.asm` и `p2.asm` будут переводиться (или, как чаще говорят, *компилироваться* или *транслироваться*) в объектные файлы с именами `p1.obj` и `p2.obj`.

Рассмотрим теперь, чего не хватает в объектном модуле, чтобы быть готовым к счёту фрагментом программы на машинном языке. Например, самая первая команда всех наших программ

```
mov ax, Data
```

должна переводиться в машинную команду пересылки формата `mov ax, i16`, однако значение константы `i16`, которая равна физическому адресу начала сегмента `Data` в памяти, делённому на 16, неизвестна программе Ассемблера и поле операнда `i16` в команде пересылки остаётся *незаполненным*. Таким образом, в объектном модуле некоторые адреса остаются неизвестными (неопределёнными). До начала счёта программы, однако, все такие адреса обязательно должны получить конкретные значения.

Объектный модуль, получаемый программой Ассемблера, состоит из двух частей: *тела* модуля и *паспорта* (или заголовка) модуля. Тело модуля состоит из сегментов, в которых находятся команды и переменные нашего модуля, а паспорт содержит описание структуры объектного модуля. В этом описании содержатся следующие данные об объектном модуле.

- Сведения обо всех сегментах модуля (длина сегмента, его спецификация).
- Сведения обо всех общедоступных (экспортируемых) именах модуля, с каждым таким именем связан его тип (**abs**, **byte**, **word**, **near** и т.д.) и адрес (входная точка) внутри какого-либо сегмента модуля (для константы типа **abs** это просто целое число).
- Сведения о местоположении и типе всех внешних адресов модуля.
- Сведения о местоположении всех остальных незаполненных адресов в модуле, для каждого такого адреса содержится информация о способе его заполнения перед началом счёта.
- Другая информация, необходимая для сборки программы из модулей.

На рис. 10.1 показано схематическое изображение объектных модулей `p1.obj` и `p2.obj`, полученных программой Ассемблера, для каждого модуля изображены его сегменты, входные точки и внешние адреса. Вся эта информация содержится в паспортах объектных модулей.¹

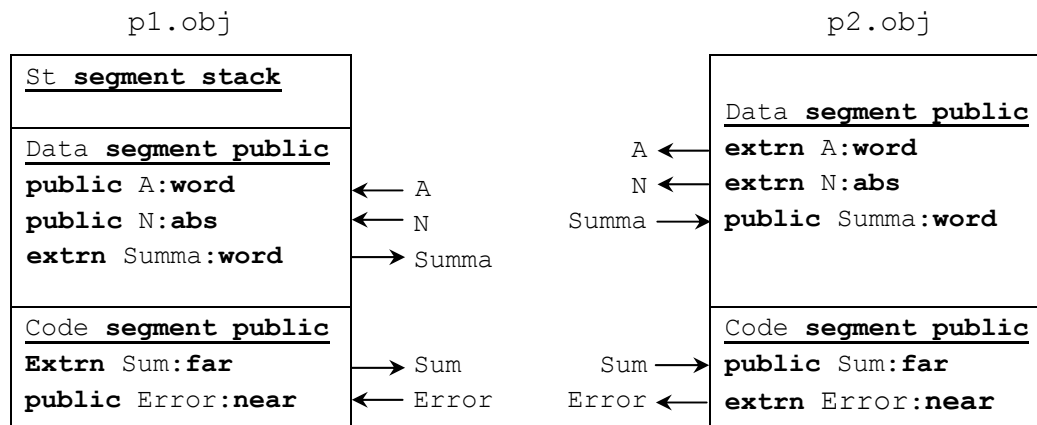


Рис. 10.1. Схематический вид объектных модулей с внешними адресами и входными точками.

Обратимся теперь к проблеме сборки программы из модулей. Как мы уже упоминали, эту работу выполняет специальная системная программа, которая называется *редактором внешних связей*. Из этого названия хорошо видно одно из назначений этой программы – устанавливать связи между внешними адресами и входными точками модулей. Рассмотрим схему работы редактора внешних связей на нашем примере.

¹ Вообще говоря, все имена, кроме внешних имён и имён входных точек, могут заменяться в объектном модуле номерами. Например, уже на важно, какие именно имена давал программист сегментам в своей программе, эта информация больше никому не понадобится.

10.2. Схема работы редактора внешних связей.

Целью работы редактора внешних связей является построение из объектных модулей почти готового к счёту программного модуля, который называется *загрузочным модулем*. Загрузочный модуль всё ещё не является полностью готовой к счёту программой на машинном языке, в этом модуле остаются незаполненными некоторые поля. Например, наша команда

```
mov ax, Data
```

всё ещё будет иметь незаполненное поле `Data` формата `i16` на месте второго операнда, так как конкретное значение этого поля будет известно только перед самым началом счёта программы, когда все её сегменты будут размещены в памяти компьютера.

При вызове редактора внешних связей ему в качестве параметров передаются имена всех объектных модулей, а также имя загрузочного модуля, который необходимо построить. Для нашего примера вызов редактора внешних связей (его имя **link**) будет выглядеть, например, так

```
link p1+p2+ioproc,p
```

Здесь `p1`, `p2` и `ioproc` – имена объектных модулей (не забывайте о третьем объектном модуле с именем `ioproc`), а `p` – имя загрузочного модуля, который надо построить.¹ Первый из перечисленных объектных модулей считается *головным* модулем, с него начинается процесс сборки загрузочного модуля. Работа редактора внешних связей включает в себя два этапа. На первом этапе происходит обработка сегментов, а на втором – собственно редактирование внешних связей и построение загрузочного модуля (загрузочные модули для нашего компьютера имеют расширение `.exe`). Разберёмся сначала с первым этапом.

В нашем примере (если не принимать во внимание объектный модуль `ioproc.obj`) имеется пять сегментов: три сегмента с именами `St`, `Data` и `Code` в модуле `p1.obj` и два сегмента с именами `Data` и `Code` в модуле `p2.obj`. Спрашивается, сколько сегментов будет в загрузочном модуле `p.exe`? Здесь логически возможны три случая.

- Все сегменты переходят в загрузочный модуль. В этом случае в нашем модуле `p.exe` должно было бы быть 5 сегментов: один стековый, два кодовых и два сегмента данных.
- Некоторые из сегментов *склеиваются*, то есть один сегмент присоединяется в конец другого сегмента.
- Некоторые из сегментов *накладываются* друг на друга (если сегменты имеют разную длину, то, конечно, более длинный сегмент будет "торчать" из-под более короткого сегмента). Разумеется, почти всегда накладывать друг на друга имеет смысл только сегменты данных, в этом случае у нескольких модулей будут *общие* сегменты данных (или, как иногда говорят, *общие области* данных).

Как именно будут обрабатываться сегменты при сборке загрузочного модуля из объектных модулей, определяет программист, задавая определённые *параметры* в директивах **segment**. Существуют следующие параметры, управляющие обработкой сегментов.

Параметр **public** у *одноимённых* сегментов означает их склеивание.² Так как сборка начинается с головного модуля, то из двух одноимённых сегментов с параметром **public** сегмент из головного модуля будет первым, в его конец будут добавляться соответствующие сегменты из других объектных модулей. В том случае, если одноимённые сегменты с параметром **public** встречаются не в головном модуле, то их порядок при склейке определяется конкретным редактором внешних связей (надо читать документацию к нему).³

Для нашего примера сегмент данных с именем `Data` объектного модуля `p2.obj` будет добавлен в конец одноимённого сегмента данных головного модуля `p1.obj`. Такая же операция будет проведена и для сегментов кода этих двух модулей. Таким образом, в загрузочном модуле останутся

¹ Если объектных модулей очень много, то существует более компактный способ задать их, не перечисляя их все в строке вызова редактора внешних связей.

² Вообще говоря, склеиваемые сегменты должны ещё принадлежать к одному и тому же *классу* сегментов. В наших примерах это выполняется, для более полного изучения понятия класса сегмента необходимо обратиться, например, к учебнику [5].

³ Заметим, что одноимённые сегменты с параметром **public** могут встречаться и внутри *одного* модуля. В этом случае такие сегменты тоже склеиваются, но эту работу, как правило, проводит сама программа Ассемблера.

только три сегмента: сегмент стека `St`, сегмент данных `Data` и кодовый сегмент `Code`. При склейке кодовых сегментов редактору внешних связей придётся изменить некоторые адреса в командах перехода внутри *добавляемого* модуля. Правда, как легко понять, меняются адреса только в командах *абсолютного* перехода и не меняются *относительные* переходы (это ещё одно достоинство команд перехода, которые реализуют *относительный* переход).

Для склеиваемых сегментов данных могут измениться начальные значения переменных во втором сегменте, например, пусть в сегменте данных второго модуля находится такое предложение резервирования памяти

```
Z      dw      Z
```

Здесь начальным значением переменной `Z` служит её собственный адрес (смещение от начала сегмента данных). При склейке сегментов это значение увеличится на длину сегмента данных первого модуля.

Отметим также, что параметр директивы сегмента **stack**, кроме того, что определяет сегмент стека, даёт такое же указание о склейке одноимённых сегментов одного класса, как и параметр **public**. Другими словами, одноимённые сегменты стека тоже склеиваются, это позволяет каждому модулю увеличивать размер стека на нужное этому модулю число байт. Таким образом, головной модуль (что вполне естественно) может не знать, какой дополнительный размер стека необходим для правильной работы остальных модулей.

Для указания *наложения одноимённых* сегментов одного класса друг на друга при сборке программы из объектных модулей предназначен параметр **common** директивы **segment**. В качестве примера использования параметра **common** рассмотрим другое решение предыдущей задачи, при этом сегменты данных двух наших модулей будут *накладываться* друг на друга. Итак, новые варианты модулей `p1.asm` и `p2.asm` приведены ниже.

```
; p1.asm
; Ввод массива, вызов внешней процедуры
include io.asm
St      segment common
        dw      64 dup (?)
St      ends
N       equ     1000
Data    segment public
A       dw      N dup (?)
S       dw      ?
Diagn   db      'Переполнение!',13,10,'$'
Data    ends
Code    segment public
        assume cs:Code,ds:Data,ss:St
Start:  mov     ax,Data
        mov     ds,ax
        mov     cx,N
        sub     bx,bx; индекс массива
L:      inint  A[bx]; Ввод массива A
        add     bx,type A
        loop   L
        extrn  Sum:far; Внешнее имя
        call   Sum; Процедура суммирования
        outint S; синоним имени Summa
        newline
        finish
        public Error; Входная точка
Error:  lea     dx,T
        outstr
        finish
Code    ends
        end    Start; головной модуль
```

```

Comment * модуль p2.asm
    Суммирование массива, контроль ошибок
    include io.asm не нужен – нет ввода/вывода
    Стек головного модуля не увеличивается
    В конечном end не нужна метка Start

*
M    equ    1000
Data segment common
B    dw    M dup (?)
Summa dw    ?
Data ends
Code segment public
    assume cs:Code,ds:Data
    public Sum; Входная точка
Sum  proc far
    push ax
    push cx
    push bx; сохранение регистров
    xor  ax,ax
    mov  cx,M
    xor  bx,bx; индекс 1-го элемента
L:   add  ax,B[bx]
    jno  L1
; Обнаружена ошибка
    pop  bx
    pop  cx
    pop  ax
    extrn Error:near
    jmp  Error
L1:  add  bx,type B
    loop L
    mov  Summa,ax
    pop  bx
    pop  cx
    pop  ax; восстановление регистров
    ret
Code ends
    end

```

Теперь сегменты данных будут накладываться друг на друга (в головном модуле сегмент данных немного длиннее, так что длина итогового сегмента данных будет равна максимальной длине накладываемых сегментов). Как видим, почти все имена в модулях теперь являются *локальными*, однако из-за наложения сегментов данных друг на друга получается, что имя A является *синонимом* имени B (это имена одной и той же области памяти – нашего массива). Аналогично имена S и Summa также будут обозначать одну и ту же переменную в сегменте данных.

Можно сказать, что при наложении друг на друга сегментов разных модулей получаются *неявные* статические связи по данным (очевидно, что накладывать друг на друга кодовые сегменты почти всегда бессмысленно). Вследствие этого можно (как в нашем примере) резко сократить число *явных* связей по данным (то есть входных точек и внешних адресов). Надо, однако, заметить, что такой стиль модульного программирования является весьма опасным: часто достаточно ошибиться в расположении хотя бы одной переменной в накладываемых сегментах, чтобы программа стала работать неправильно.¹ Например, рассмотрите, что будет, если поменять в одном из накладываемых сегментов места массив и переменную для хранения суммы этого массива (никакой диагностики об ошибке при этом, естественно, не будет).

¹ Стиль программирования с общими (накладываемыми друг на друга) сегментами данных широко используется, например, в языке Фортран, при этом часто случаются ошибки, вызванные плохим семантическим согласованием общих сегментов данных.

Заметим, что во всех предыдущих примерах нам было всё равно, в каких именно конкретных областях памяти будут располагаться сегменты нашей программы во время счёта. Более того, считается хорошим стилем так писать программы, чтобы их сегменты на этапе счёта могли располагаться в *любых* свободных областях оперативной памяти компьютера. Однако очень редко может понадобиться расположить определённый сегмент с явно заданного программистом адреса оперативной памяти. Для обеспечения такой возможности на языке Ассемблер служит параметр **at** <адрес сегмента> директивы **segment**. Здесь <адрес сегмента> является адресом начала сегмента в оперативной памяти, делённым на 16. В качестве примера рассмотрим такое описание сегмента с именем `Interrupt_Vector`.

```
Interrupt_Vector Segment at 0
Divide_by_Zero dd ?
Trace_Program dd ?
Fatal_Interrupt dd ?
Int_Command dd ?
Into_Command dd Code:Error
Interrupt_Vector ends
```

Этот сегмент во время счёта программы будет накладываться на начало вектора прерываний, а переменные этого сегмента будут обозначать конкретные адреса процедур обработки прерываний. Так заданный сегмент данных может облегчить написание собственных процедур-обработчиков прерываний.

Рассмотрим теперь второй этап работы редактора внешних связей – настройку всех внешних имён на соответствующие им входные точки в других модулях. На этом этапе редактор внешних связей начинает просматривать паспорта всех модулей и читать оттуда их внешние имена. Эта работа начинается с головного модуля, для всех его внешних имён ведётся поиск соответствующих им входных точек в других модулях. Если такой поиск оказывается безуспешным, то редактор внешних связей фиксирует ошибку: неразрешённое (в смысле не найденное) внешнее имя.

Для некоторого внешнего имени могут существовать и несколько входных точек в разных модулях. При этом многие редакторы внешних связей такую ошибку не фиксируют и берут первое встреченное внешнее имя, так что программисту надо быть осторожным и обеспечить уникальность входных имён у всех модулей. К большому сожалению, некоторые редакторы внешних связей (в том числе и в Ассемблере MASM-4.0) не проверяют *соответствие типов* у внешнего имени и входной точки. Таким образом, например, внешнее имя-переменная размером в слово может быть связано с входной точкой – переменной размером в байт или вообще с меткой. При невнимательном программировании это может привести к серьёзным ошибкам, которые трудно найти при отладке программы.

Когда для некоторого внешнего имени найдена соответствующая входная точка, то устанавливается связь: адрес входной точки записывается в соответствующее поле внешнего имени. Например, для команды

```
call Sum; Формат i32=seg:off= call seg:off
```

на место поля `off` запишется смещение начала процедуры суммирования в объединённом после склеивания сегменте кода, а поле `seg` пока останется незаполненным, его значение (адрес начала сегмента кода, делённый на 16) будет известно только после размещения программы в оперативной памяти перед началом счёта. Аналогично, на место команды

```
mov cx, N
```

запишется команда

```
mov cx, 1000
```

Итак, если для каждого внешнего имени найдена входная точка в другом объектном модуле, то редактор внешних связей нормально заканчивает свою работу, выдавая в качестве результата загрузочный модуль. Загрузочный модуль, как и объектный, состоит из тела модуля и паспорта. Тело загрузочного модуля содержит все его сегменты,¹ а в паспорте собраны необходимые для дальнейшей работы данные:

¹ В загрузочном модуле могут не храниться "пустые" сегменты данных, которые состоят только из директив резервирования памяти *без начальных значений*. Для сегментов данных, в которых есть области памяти, как с начальными значениями, так и без начальных значений, лучше сначала описывать области памяти с начальными значениями. Это даёт возможность помещать такой сегмент данных в загрузочный модуль в

- информация обо всех сегментах (длина и класс сегмента), в частности, данные о сегменте стека;
- информация обо всех ещё неопределённых полях в сегментах модуля;
- информация о расположении входной точки программы (в нашем примере – метки `Start`);
- другая необходимая информация.

На рис. 10.2 показан схематический вид загрузочного модуля, полученного для первого варианта нашего примера (со склеиваемыми сегментами). Внутри сегмента кода показаны незаполненные поля (они подчёркнуты). Метку `Start` можно рассматривать как единственную *входную точку* загрузочного модуля.

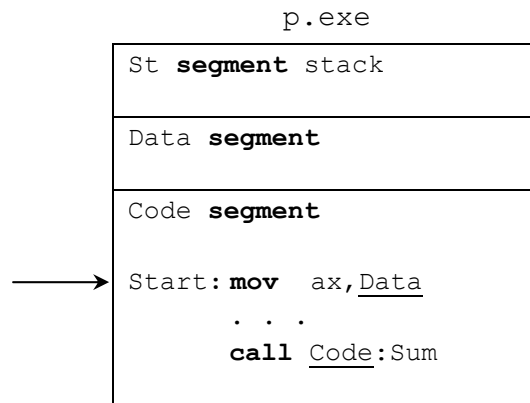


Рис. 10.2. Схематический вид загрузочного модуля, незаполненные поля подчёркнуты.

Вот теперь всё готово для запуска программы на счёт. Осталось только поместить нашу программу в оперативную память и передать управление на её начало (в нашем примере – на метку `Start`). Эту работу делает служебная программа, которая называется статическим загрузчиком (далее мы познакомимся и с другим видом загрузчика – *динамическим* загрузчиком). Сейчас мы рассмотрим схему работы статического загрузчика.

10.3. Схема работы статического загрузчика.

При своём вызове статический загрузчик (в дальнейшем – просто загрузчик) получает в качестве параметра имя файла загрузочного модуля. Работа начинается с чтения паспорта загрузочного модуля и определения объёма памяти, необходимого для счёта программы. Обычно это сумма длин всех сегментов программы, однако иногда в процессе счёта программа может запрашивать у операционной системы дополнительные сегменты (или, как говорят, блоки) памяти. Эти блоки чаще всего используются для размещения динамических переменных, так как заранее неизвестно, сколько памяти потребуется для хранения этих динамических переменных. Обычно в паспорте можно указать минимальный объём дополнительной памяти, без наличия которого нельзя запускать программу на счёт, и максимальный объём такой памяти, который может запросить программа.¹

Итак, если на компьютере нет необходимого объёма свободной памяти, то загрузчик выдаёт аварийную диагностику и не запускает программу на счёт. В противном случае из загрузочного модуля читаются и размещаются в памяти все сегменты этого модуля,² для каждого сегмента, таким образом, определяется адрес его начала в оперативной памяти.

"урезанном" виде, все области данных без начальных значений в файл не записываются, что позволяет уменьшить размер файла загрузочного модуля. Разумеется, в паспорте загрузочного модуля хранится полная информация обо всех его сегментах (в частности, об их длине).

¹ Максимальный объём дополнительной памяти указывается из соображений безопасности, чтобы из-за ошибок в программе она не стала запрашивать в цикле всё новые и новые блоки памяти, что может сильно помешать счёту других программ.

² Как мы уже упоминали, "пустые" сегменты, содержащие только переменные без начальных значений, в загрузочном модуле обычно не хранятся, такие сегменты просто размещаются на свободных местах памяти и, естественно, области памяти в них не будут иметь конкретных начальных значений.

Затем загрузчик просматривает паспорт загрузочного модуля и заполняет в сегментах все поля, которые ещё не имели необходимых значений. В нашем примере для загрузочного модуля `p.exe` это поля с именами `Data` и `Code` в сегменте команд (см. рис. 10.2). В эти поля загрузчик записывает соответственно адреса начал сегментов данных и кода, делённые на 16. На этом настройка программы на конкретное месторасположение в оперативной памяти заканчивается.

Далее загрузчик, анализируя паспорт, определяет тот сегмент, который будет начальным сегментом стека программы (как мы знаем, этот сегмент имеет параметр **Stack** в директиве начала сегмента). Адрес начала этого сегмента, делённый на 16, записывается в сегментный регистр `SS`, а длина этого сегмента – в регистр вершины стека `SP`. Таким образом, стек программы готов к работе.¹

И, наконец, последним действием загрузчик производит дальний абсолютный переход с возвратом на начала загруженной программы, например, по команде

```
call Code:Start
```

Здесь `Code` – адрес начала головного кодового сегмента, а `Start` – входная точка программы, с которой начинается её выполнение (эта информация, как уже говорилось, содержится в паспорте загрузочного модуля). Далее начинается собственно выполнение загруженной программы.

Как можно догадаться из описания схемы работы загрузчика, макрокоманда **finish** должна, в конечном счете, вернуть управление загрузчику, чтобы он освободил занимаемую программой память и подготовился к загрузке следующей программы. Такой же возврат к загрузчику должен производиться и при аварийном завершении программы, например, при делении на ноль. Как Вы можете догадаться, это должна делать процедура-обработчик соответствующего прерывания. В нашем курсе мы не будем более подробно рассматривать весь этот механизм.

В заключение рассмотрения схемы работы загрузчика отметим, что иногда, хотя и редко, требуется в одной служебной программе объединить функции редактора внешних связей и загрузчика. Например, это может понадобиться в том случае, когда некоторая программа получает новый объектный модуль или изменяет один или несколько существующих объектных модулей и тут же хочет загрузить и выполнить изменённую программу. Системная программа, которая объединяет в себе функции редактора внешних связей и загрузчика, называется обычно *связывающим загрузчиком*. В нашем курсе мы не будем изучать подробности работы связывающего загрузчика.

Итак, мы изучили *схему* выполнения модульной программы, эта схема включает в себя два этапа: редактирование внешних связей и загрузки программы в оперативную память с настройкой по месту её расположения в этой памяти. Такая схема счёта носит название "статическая загрузка и статическое связывание модулей". Главное достоинство этой схемы выполнения модульной программы состоит в том, что после того, как программа загружена в память и начала выполняться, для её работы не требуется вмешательство системных программ при использовании внешних связей, все статические внешние связи уже установлены, а соответствующие внешние адреса известны и записаны в сегментах.

Поймём теперь, что эта схема выполнения модульной программы со статической загрузкой и статическим связыванием модулей имеет очень серьёзный недостаток. Предположим, что наша достаточно сложная программа состоит из 100 модулей (для простоты будем считать, что в каждом модуле содержится одна из *процедур* программы). Тогда перед началом работы все эти 100 процедур должны быть размещены в оперативной памяти и связаны между собой и с основной программой (т.е. в нужных местах проставлены адреса этих процедур).

В то же время чаще всего бывает так, что при каждом конкретном счёте программы, в зависимости от введённых данных, **на самом деле** понадобится вызвать только относительно небольшое количество этих процедур (скажем, 10 из 100). Тогда получается, что для каждого запуска программы необходимы лишь 10 процедур из 100, а остальные только зря занимают место в памяти, обращений к ним не будет. Конечно, для следующего запуска программы (с другими входными данными) могут понадобиться **другие** 10 процедур из 100, но в целом картина не меняется: каждый раз около 90% памяти не используется!

Исходя из вышесказанного понятно, что на первых ЭВМ, когда оперативной памяти было мало, схема счёта со статическим связыванием и статической загрузкой модулей применялась редко.

¹ Так как эти действия требуют больше одной команды, то их надо проводить в режиме с закрытыми прерываниями, чтобы не использовать не полностью подготовленный к работе стек при возникновении прерывания (т.е. эта часть загрузчика является уже знакомой нам критической секцией).

Первые программисты не могли себе позволить так нерационально использовать дорогую оперативную память, поэтому при счёте модульных программ применялась схема с *динамическим связыванием* и *динамической загрузкой* модулей в оперативную память. Однако в дальнейшем, при увеличении объёмов оперативной памяти, и особенно после появления так называемой виртуальной памяти,¹ стала в основном использоваться схема счёта модульных программ со статической загрузкой и связыванием.

Далее отметим, что в настоящее время счёт модульных программ (а подавляющее большинство программ только такие теперь и есть), уже снова чаще всего выполняется с динамическим связыванием и динамической загрузкой модулей. Причина здесь состоит в том, что, несмотря на сильно возросший объём памяти в современных ЭВМ, сложность задач и число реализующих их модулей растёт *значительно быстрее*, чем объём памяти.²

Перейдём теперь непосредственно к изучению схемы счёта модульной программы с использованием динамического связывания и динамической загрузки модулей. Обычно та системная программа, которая занимается динамическим связыванием и динамической загрузкой модулей, называется *динамическим загрузчиком*.

10.4. Схема работы динамического загрузчика.

Суть работы динамического загрузчика состоит в следующем. Сначала он размещает в памяти не всю программу целиком, как статический загрузчик, а только её основную часть (головной модуль программы). Все остальные модули загружаются в оперативную память по мере необходимости, когда к ним будет реальное обращение из головного модуля или других (уже загруженных) модулей программы. Иногда это называется *загрузкой по требованию*. Отметим здесь важную особенность такой загрузки по требованию. После размещения головного модуля в оперативной памяти динамический загрузчик *не проверяет*, что все другие модули, которые он может вызывать в процессе своей работы, *на самом деле существуют* (на это остаётся только надеяться из всех сил ☺).³ Естественно, что, если какой-нибудь модуль не будет найден, когда понадобится его вызвать, то будет зафиксирована ошибка времени выполнения программы.

Надо отметить, что динамические загрузчики современных ЭВМ достаточно сложны, поэтому мы рассмотрим только упрощённую схему работы такого загрузчика. Эта схема будет очень похожа на схему работы динамических загрузчиков в ЭВМ второго поколения середины прошлого века.

Сначала разберёмся с редактированием внешних связей при динамической загрузке модулей (это и называется *динамическим связыванием* модулей). Работу динамического загрузчика будем рассматривать на примере программы, головной модуль которой вызывает три внешних процедуры с именами A, Beta и C12. Ниже приведён фрагмент сегмента кода этого головного модуля на Ассемблере:

```
Code segment
      assume cs:Code, ds:Data, ss:Stack
Start:mov   ax, Data
      mov   ds, ax
      . . .
      extrn A: far
      call  A
      . . .
      extrn Beta: far
```

¹ Виртуальная память позволяет использовать в программе диапазон адресов памяти, превышающий физический объём памяти компьютера. Например, при физической памяти 2^{20} байт (как в нашей младшей модели) можно использовать адресное пространство в 2^{24} байт. С виртуальной памятью Вы познакомитесь в следующем семестре в курсе "Системное программное обеспечение".

² Есть и другая причина широкого распространения схемы счёта с динамическим связыванием и динамической загрузкой модулей, мы рассмотрим эту причину при изучении работы ЭВМ в так называемом мультипрограммном режиме.

³ В современных технологиях программирования проверка того, что существуют **все** модули, вызов которых *возможен* при счёте программы, может оказаться весьма трудоёмкой операцией. Дело в том, что эти модули могут, в принципе, находиться где угодно, например, в сетевой (удалённой) библиотеке объектных модулей на другом конце Земли.

```

    call Beta
    . . .
    extrn C12:far
    call C12
    . . .
    finish
Code ends
end Start; головной модуль

```

Пусть внешние процедуры с именами *A*, *Beta* и *C12* расположены каждая в своём отдельном модуле, где эти имена, естественно, объявлены общедоступными (**public**). При своём вызове динамический загрузчик получает в качестве параметра имя головного *объектного* модуля, по существу это первый параметр редактора внешних связей (загрузочного модуля у нас нет, и не будет). Сначала динамический загрузчик размещает в оперативной памяти все сегменты головного модуля и начинает настройку его внешних адресов. Для этих целей он строит в оперативной памяти две вспомогательные таблицы: *таблицу внешних имён* (ТВИ) и *таблицу внешних адресов* (ТВА), каждая из этих таблиц располагается в своём сегменте памяти.

В таблицу внешних имён заносятся все внешние имена программы (в начале работы это имена внешних процедур головного модуля *A*, *Beta* и *C12*). Каждое имя будем представлять в виде текстовой строки, заканчивающейся, как это часто делается, символом с номером ноль в алфавите (будем обозначать этот символ `\0`, как это принято в языке C). Ссылка на имя – это смещение начала этого имени от начала ТВИ. В заголовке (в первых двух байтах) ТВИ хранится ссылка на начало свободного места в этой таблице (номер первого свободного байта). На рис. 10.3 приведён вид ТВИ после обработки головного модуля (в каждой строке таблицы, кроме заголовка, мы разместили по четыре символа).

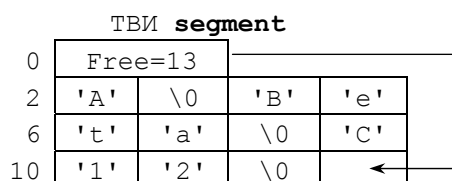
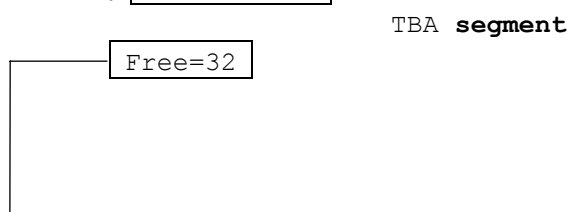


Рис. 10.3. Вид таблицы внешних имён после загрузки головного модуля.

Другая таблица динамического загрузчика, таблица внешних адресов, состоит из строк, каждая строка содержит четыре поля. Первое поле имеет длину четыре байта, в нём находится команда близкого абсолютного перехода `jmp LoadGo`. Это переход на начало некоторой *служебной* процедуры динамического загрузчика. Мы назвали эту служебную процедуру именем *LoadGo*, что будет хорошо отражать её назначение – загрузить внешнюю процедуру и перейти на её выполнение. Процедура *LoadGo* загружается вместе с головным модулем и *статически* связана с ним.

Во втором поле *Offset* длиной 2 байта находится адрес (смещение) внешней процедуры на специальном рабочем поле, о котором мы расскажем немного ниже. До первого обращения к внешней процедуре в это поле динамический загрузчик записывает константу `0FFFFh`, что является признаком *отсутствия* данной процедуры на рабочем поле. В третьем поле длиной в два байта расположена ссылка на имя этой внешней процедуры в таблице внешних имён. И, наконец, четвёртое поле, тоже длиной в 2 байта, содержит различную служебную информацию (флаги режимов работы) для динамического загрузчика, о чём мы также немного поговорим далее. Таким образом, каждая строка таблицы внешних имён описывает одну внешнюю процедуру и имеет длину 10 байт. В заголовке (первых двух байтах) ТВА содержится ссылка на начало свободного места в этой таблице. Таким образом, перед началом счёта ТВА будет иметь вид, показанный на рис. 10.4.

Каждая команда вызова внешней процедуры в головном модуле заменяется динамическим загрузчиком на команду перехода с возвратом на соответствующую строку ТВА. Например, команда `call Beta` заменяется на команду `call TBA:12`, а команда в головном модуле `call C12` заменяется на команду `call TBA:22`.



2	<code>jmp LoadGo</code>	0FFFFh	2 ('A')	flags
12	<code>jmp LoadGo</code>	0FFFFh	4 ('Beta')	flags
22	<code>jmp LoadGo</code>	0FFFFh	9 ('C12')	flags
→ 32				

Рис. 10.4. Вид таблицы внешних адресов после загрузки головного модуля.

Проследим работу нашей программы. Пусть головная программа в начале своего выполнения вызывает внешнюю процедуру с именем *Beta*. Естественно, что при первой попытке основной программы вызвать процедуру *Beta*, управление получает служебная процедура *LoadGo* динамического загрузчика. Получив управление, процедура *LoadGo* последовательно выполняет следующие действия.

1. Сначала вычисляется величина `TBA_proc`, равная адресу строки вызываемой процедуры *Beta* в таблице TBA. Для случая вызова процедуры *Beta* величина `TBA_proc`=12.
2. Затем анализируется поле `Offset` в строке `TBA_proc`. Если `Offset`=-1, то это означает, что нужной внешней процедуры с именем *Beta* в оперативной памяти ещё нет. В этом случае процедура *LoadGo* производит поиск объектного модуля, содержащего требуемую процедуру (в паспорте этого модуля указана входная точка с именем *Beta* с типом дальней метки *far*). Если такой объектный модуль не найден, то фиксируется фатальная ошибка времени выполнения и наша программа завершается, иначе требуемая внешняя процедура *Beta* загружается служебной процедурой *LoadGo* в оперативную память и динамически связывается с основной программой. Для загрузки процедур в памяти выделяется специальная область, она часто называется *рабочим полем процедур*. Мы отведём под рабочее поле сегмент с именем *Work*, занимающий, например, 50000 байт:

```
Work    segment
        db    50000 dup (?)
Work    ends
```

Рабочее поле размещается в оперативной памяти одновременно с сегментами головного модуля, TBI и TBA. После загрузки *Beta* поле `Offset` в строке `TBA_proc` принимает значение адреса начала процедуры на рабочем поле. В нашем случае процедура *Beta* загружается с начала рабочего поля, так как оно пока не содержит других внешних процедур, так что поле `Offset` принимает значение 00000h.

3. Анализируется адрес дальнего возврата, расположенный на вершине стека (по этому адресу процедура *Beta* должна возвратиться после окончания своей работы). Целью такого анализа является определение того, производится ли вызов процедуры *Beta* из *головного* модуля программы (как в нашем примере), или же из некоторой *внешней* процедуры, уже расположенной на рабочем поле (что, конечно, тоже возможно). Ясно, что такой анализ легко провести по значению сегмента в адресе возврата (обязательно поймите, как это сделать).
 - Если вызов внешней процедуры производится из головного модуля, то наша служебная процедура *LoadGo* производит дальний абсолютный переход на начало требуемой внешней процедуры, расположенной на рабочем поле, по команде вида `jmp Work:Offset`. Ясно, что в этом случае возврат из внешней процедуры будет производиться в головной модуль нашей программы (адрес возврата, как обычно, на вершине стека).
 - Если вызов производится из процедуры, расположенной на рабочем поле, то процедура *LoadGo* производит следующие действия, при выполнении которых используется ещё один служебный сегмент динамического загрузчика, описанный, например, так:

```
My_Stack segment
Free      dw    0
          dw 15000 dup (?)
My_Stack ends
```

Этот сегмент используется как вспомогательный программный (не аппаратный) стек динамического загрузчика. Наш стек *My_Stack* в отличие от машинного стека, будет "расти" сверху-вниз, при этом переменная *Free* выполняет роль указателя вершины программного

стека – регистра SP. Сначала LoadGo извлекает из аппаратного стека (на него, как обычно, указывает регистровая пара <SS, SP>) адрес возврата (два слова) и записывает этот адрес в программный стек My_Stack. Затем туда же записывается значение TBA_proc, таким образом запоминается, из какой процедуры произошёл вызов. И, наконец, LoadGo производит вызов необходимой внешней процедуры, расположенной на рабочем поле, командой вида `call Work:Offset`. Очевидно, что возврат из внешней процедуры в этом случае будет производиться в нашу служебную процедуру LoadGo.

4. После возврата из внешней процедуры в LoadGo, она производит следующие действия (напомним, что уже известно о том, что вызов внешней процедуры был осуществлён из некоторой процедуры, расположенной на рабочем поле).
 - Сначала из вспомогательного стека My_Stack извлекается значение TBA_proc той процедуры, в которую необходимо вернуться (это значение на вершине нашего программного стека по адресу My_Stack[Free]).
 - Затем анализируется значение поля Offset в строке TBA_proc. Если величина Offset=-1, то это означает, что наша процедура была удалена с рабочего поля. В этом случае производится повторная загрузка процедуры на рабочее поле (вообще говоря, начиная с другого свободного места этого поля). Адрес нового положения процедуры на рабочем поле записывается в поле Offset в строке TBA_proc.
 - Из вспомогательного стека My_Stack извлекается адрес дальнего возврата, в котором слово, содержащее значение сегмента возврата, заменяется величиной Offset из строки TBA_proc. И, наконец, производится дальний безусловный переход по так скорректированному адресу возврата.

На этом LoadGo завершает обработку вызова внешней процедуры. Как видим, эта процедура играет роль своеобразного буфера, располагаясь между вызывающей программой и вызываемой внешней процедурой. Как следует из описания алгоритма работы LoadGo, она может контролировать наличие требуемой процедуры на рабочем поле и, при необходимости, загружать вызываемую процедуру на свободное место рабочего поля. Обратите внимание, что контроль наличия процедуры на рабочем поле производится как при вызове этой процедуры, так и при возврате в неё.

Упражнение. Объясните, каким образом служебная процедура LoadGo, получив управление по команде `jmp LoadGo`, вычислит величину TBA_proc, то есть определит, что надо загружать именно процедуру с именем Beta, а не какую-нибудь другую внешнюю процедуру.

Продолжим анализ работы динамического загрузчика на нашем примере. Пусть загруженная процедура Beta имеет длину, например, 30000 байт и, в свою очередь, содержит вызов некоторой внешней процедуры с именем Delta:

```
Beta proc far
    . . .
    extrn Delta:far
    call Delta
    . . .
    ret
Beta endp
```

Теперь, после динамической загрузки процедуры Beta на рабочее поле и связывание внешних адресов с помощью TBA, вызов процедуры Beta будет производиться с помощью служебной процедуры LoadGo. Правда, необходимо заметить, что вызов стал длиннее, чем при статическом связывании, за счёт дополнительных команд, выполняемых процедурой LoadGo. Кроме того, как мы вскоре выясним, внешние процедуры могут неоднократно загружаться на рабочее поле и удаляться с него, что, конечно, может вызвать существенное замедление выполнения программы пользователя. Это, однако, неизбежная плата за преимущества динамической загрузки модулей. По существу, здесь опять работает уже упоминавшееся нами правило рычага: выигрывая в объёме памяти, необходимом для счёта модульной программы, мы неизбежно сколько-то проигрываем в скорости работы нашей программы. Важно чтобы выигрыш, с точки зрения конкретного пользователя, был больше проигрыша.

На рис. 10.5 показан вид ТВИ, TBA и рабочего поля после загрузки процедуры Beta.

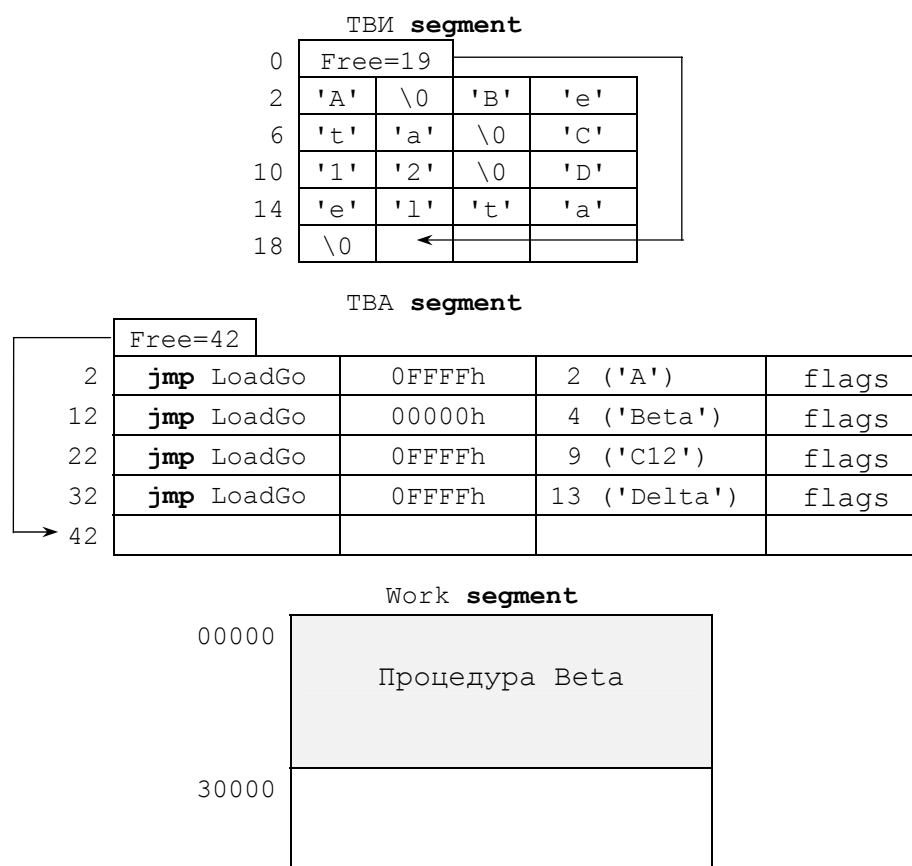
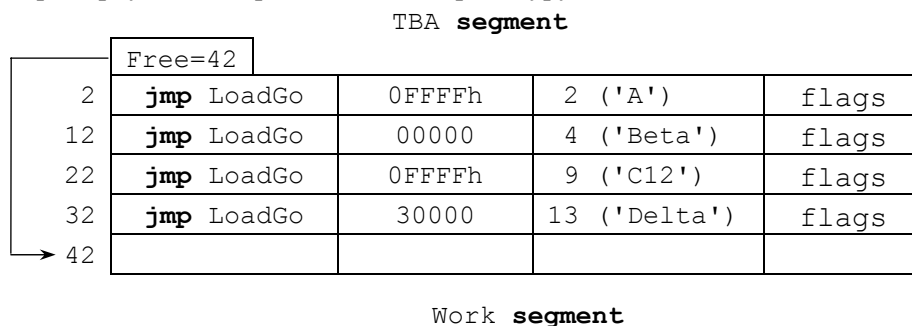


Рис. 10.5. Вид TBI, TBA и рабочего поля после загрузки процедуры Beta.

Продолжим изучение выполнения нашей модульной программы. Предположим далее, что, проработав некоторое время, процедура Beta вызовет внешнюю процедуру с именем Delta, которая имеет длину 15000 байт. Так как команда `call Delta` в процедуре Beta при загрузке этой процедуры на рабочее поле заменена динамическим загрузчиком на команду `call TBA:32`, то управление опять получает служебная процедура LoadGo. Она находит процедуру Delta¹ и размещает её на свободном месте рабочего поля (в нашем примере с адреса 30000), затем настраивает внешние адреса в этой процедуре (если они есть) и соответствующие строки в TBA.

На рис. 10.6 показан вид TBA и рабочего поля после загрузки и связывания процедуры Delta.

Продолжим наше исследование работы динамического загрузчика. Предположим теперь, что произошёл возврат из процедур Delta и Beta в основную программу, которая после этого вызвала процедуру A длиной в 25000 байт. Процедуры A нет на рабочем поле, поэтому её надо загрузить, однако вызванная процедура LoadGo определяет, что на рабочем поле нет достаточно места для размещения процедуры A. Выход здесь только один – удалить с рабочего поля одну или несколько процедур, чтобы освободить достаточно место для загрузки процедуры A. В нашем случае достаточно, например, удалить с рабочего поля процедуру Beta.



¹ Точнее, как мы уже говорили, ищется объектный модуль, в котором расположена эта процедура.

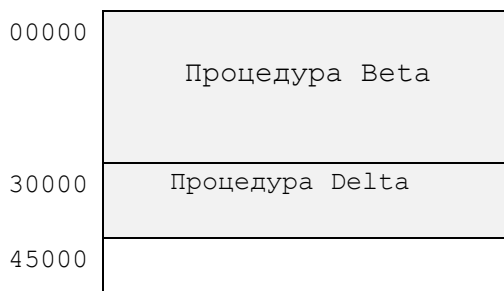


Рис. 10.6. Вид ТВА и рабочего поля после загрузки процедуры Delta.

Итак, служебная процедура LoadGo удаляет с рабочего поля процедуру Beta, загружает на освободившееся место процедуру A и корректирует соответствующим образом строки ТВА. На рис. 10.7 показан вид ТВА и рабочего поля после загрузки процедуры A.

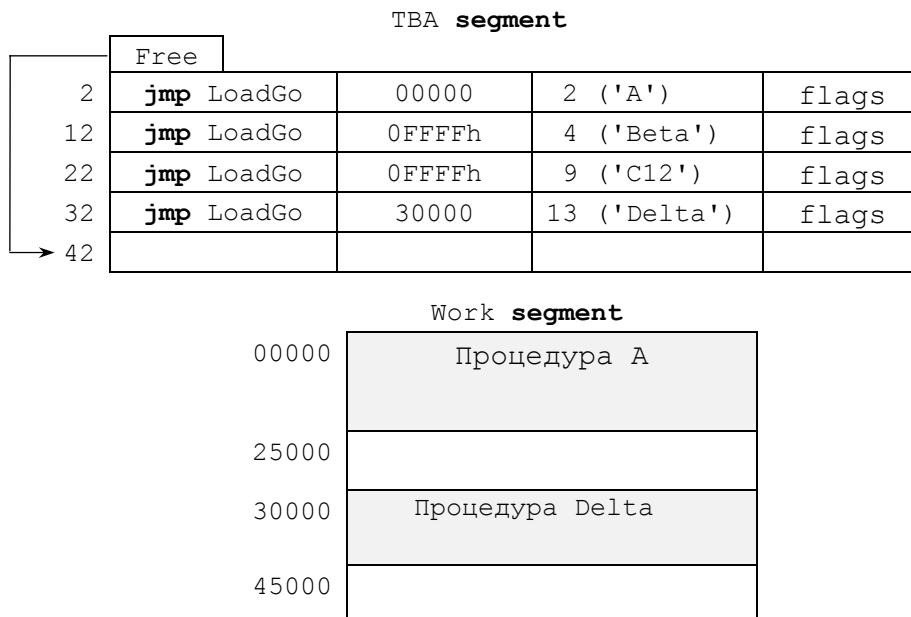


Рис. 10.7. Вид ТВА и рабочего поля после загрузки процедуры A.

Как следует из описания работы динамического загрузчика, на рабочем поле всегда находятся последние из выполняемых процедур, а программа пользователя не должна ни о чём заботиться и работает, как и при статической загрузке модулей, просто обычным образом вызывая необходимые ей внешние процедуры. Часто говорят, что действия динамического загрузчика *прозрачны* (т.е. невидимы) для программы пользователя.

Иногда, однако, программа пользователя нуждается в некотором *управлении* динамической загрузкой модулей на рабочее поле. Например, пусть программист знает, что некоторая процедура X будет вызываться часто (в цикле). В этом случае программист может потребовать у динамического загрузчика, чтобы эта процедура X по возможности не удалялась с рабочего поля. Другими словами, динамический загрузчик при нехватке памяти на рабочем поле должен сначала стараться удалить с него другие процедуры, а лишь в последнюю очередь процедуру X. Говорят, что процедура X *фиксируется* на рабочем поле.

Для фиксации процедуры на рабочем поле в состав динамического загрузчика входит служебная процедура с именем Lock. Программа пользователя должна вызвать эту процедуру с параметром – именем фиксируемой процедуры. На Ассемблере необходимо определить способ передачи этого строкового параметра в служебную процедуру, а на языке Паскаль это можно записать, например, так

```
Lock ('X');
```

Процедура Lock находит в ТВА строку, соответствующую указанной процедуре, и ставит в этой строке признак о том, что она зафиксирована на рабочем поле. Когда необходимость в фиксации процедуры X на рабочем поле отпадёт, программист может *расфиксировать* эту процедуру, вызвав служебную процедуру динамического загрузчика с именем UnLock. На Паскале это, например, можно сделать так:


```
UnLock ('X');
```

Разумеется, в строке TBA в поле флагов теперь надо предусмотреть битовый признак Lock/UnLock. Обратите также внимание, что служебные процедуры LoadGo, Lock и UnLock *статически* связаны с программой пользователя, т.е. расположены в её сегменте кода. Об этом должен позаботиться динамический загрузчик при размещении в оперативной памяти головного модуля программы.

Рассмотрим теперь главные недостатки схемы счёта с динамической загрузкой и связыванием модулей. Во-первых, следует отметить дополнительные вычислительные затраты на выполнение служебных процедур (LoadGo, Lock, UnLock и других) во время счёта программы пользователя. Во-вторых, может достаточно существенно замедлиться выполнения всей программы, так как теперь во время счёта может понадобиться периодически загружать модули на рабочее поле, т.е. использовать относительно медленную внешнюю память. В том случае, если такие затраты допустимы, то схеме счёта с динамической загрузкой следует отдать предпочтение.¹

В современных ЭВМ наборы динамически загружаемых модулей одной тематики обычно объединяют в один файл – библиотеку динамически загружаемых модулей (по-английски Dynamic Link Library – DLL).

На этом мы завершим наше краткое знакомство со схемами выполнения модульных программ.

11. Понятие о системе программирования.

Как мы уже упоминали в начале нашего курса, все программы, которые выполняются на компьютере, можно разделить на два класса – *прикладные* и *системные*. Вообще говоря, компьютеры существуют для того, чтобы выполнять прикладные программы, однако понятно, что нас то в первую очередь будут интересовать именно системные программы. ☺

Все системные программы можно, в свою очередь, разделить на два класса. В один класс входят программы, предназначенные для управления оборудованием ЭВМ (и, вообще говоря, для обеспечения эффективной эксплуатации этого оборудования), а также программы, управляющие на компьютере выполнением *других* программ. Программы этого класса входят в комплекс системных программ, который называется *операционная система* ЭВМ. Подробно операционные системы Вы будете изучать в курсе следующего семестра "Системное программное обеспечение".

В другой класс входят системные программы, предназначенные для автоматизации процесса разработки и реализации *новых* программ. Программы этого класса входят в *системы программирования*. Не надо, однако, думать, что система программирования состоит только из таких системных программ, которые помогают писать новые программы. Ниже перечислены все компоненты, входящие в систему программирования.

11.1. Компоненты системы программирования.

- Языки системы программирования. Сюда относятся как языки программирования, предназначенные для записи алгоритмов (Паскаль, Фортран, С, Ассемблер и т.д.), так и другие языки, которые служат для управления самой системой программирования (например, так называемый язык командных файлов), или предназначены для автоматизации разработки больших программ (например, язык спецификации программ).
- Служебные программы системы программирования. Со многими из этих программ мы уже познакомились в нашем курсе, например, сюда входят такие программы.
 - Текстовые редакторы, предназначенные для набора и исправления текстов программ на языках программирования.
 - Трансляторы (компиляторы) с одного языка на другой (например, программа Ассемблера транслирует с языка Ассемблер на язык объектных модулей).
 - Редакторы внешних связей, собирающие программы из модулей.
 - Загрузчики.

¹ Эта схема может быть неприемлема, например, для так называемых программ реального времени, которые предназначены для управления быстрыми внешними устройствами (ракетой, химическим или ядерным реактором и т.д.).

- Отладчики, помогающие пользователям искать и исправлять ошибки в программах в диалоговом режиме.
 - Оптимизаторы, позволяющие автоматически улучшать программу, написанную на определённом языке.
 - Профилировщики, которые определяют, какой процент времени выполняется та или иная часть программы. Это позволяет выявить наиболее интенсивно используемые фрагменты программы и оптимизировать их (например, переписав на языке Ассемблера).
 - Библиотекари, которые позволяют создавать и изменять файлы-библиотеки процедур (например, библиотеки динамически загружаемых процедур DLL), файлы-библиотеки макроопределений и т.д.
 - Интерпретаторы, которые могут выполнять программы без перевода их на другие языки (точнее, с *построчным* переводом на машинный язык и последующим выполнением каждого такого переведённого фрагмента).
- Информационное обеспечение системы программирования. Сюда относятся различные структурированные описания языков, служебных программ, библиотек модулей и т.п. Без хорошего информационного обеспечения современные системы программирования работать не могут. Каждый пользователь неоднократно работал с этой компонентой системы программирования, нажимая функциональную клавишу F1 или выбирая из меню пункт Help (Помощь).

На рис. 11.1 показана общая схема прохождения программы пользователя через систему программирования. Программные модули пользователя на этом рисунке заключены в прямоугольники, а системные программы – в прямоугольники с закруглёнными углами.

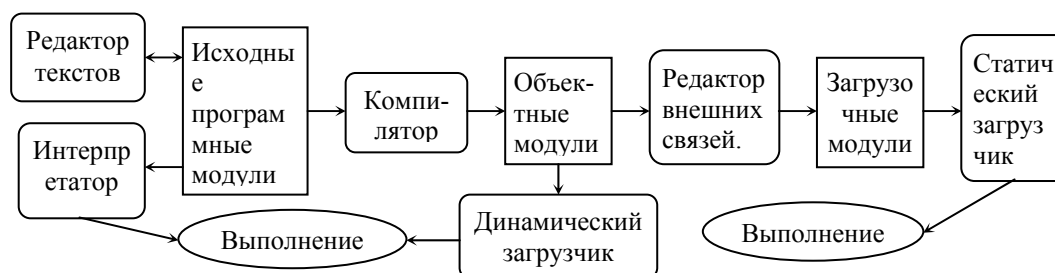


Рис. 11.1. Общая схема прохождения программы через систему программирования.

На этом мы закончим описание состава системы программирования и перейдём к описанию характеристик исполняемых модулей.

11.2. Характеристики исполняемых модулей.

Модульная программа состоит из отдельных исполняемых модулей, которые могут обладать некоторыми специфическими характеристиками, к рассмотрению которых мы сейчас и перейдём.

11.2.1. Перемещаемые модули.

Программные модули, обладающие свойством перемещаемости, могут быть во время счёта программы перенесены в другое место оперативной памяти так, что это не повлияет на правильную работу этих модулей.

Рассмотрим пример, когда это свойство исполняемых модулей может оказаться полезным. На рис. 10.7 показан вид рабочего поля, на котором находятся процедуры с именами A и Delta. Предположим, что динамическому загрузчику необходимо разместить на этом поле новый модуль, скажем процедуру с именем C12, которая имеет длину 8000 байт. Видно, что загрузчику не удастся это сделать, не удалив с рабочего поля какую-нибудь процедуру, так как, несмотря на то, что 10000 байт рабочего поля свободны, но это свободное пространство разбито на две части, ни в одну из которых не войдёт процедура C12.

В том случае, если модуль Delta является перемещаемым, его можно сдвинуть так, чтобы объединить свободные участки рабочего поля и разместить на нём новую процедуру C12. Сдвиг модуля в оперативной памяти является по существу операцией пересылки массива и это заведомо

более быстрая операция, чем удаление модуля с рабочего поля (ведь его потом, скорее всего, придётся вернуть обратно).

Рассмотрим, какими свойствами должен обладать модуль на Ассемблере, чтобы быть перемещаемым. При перемещении сегментов модуля на другое место памяти должны поменяться адреса начал этих сегментов в сегментных регистрах. Отсюда вытекает следующее ограничение на перемещаемый модуль: он не должен загружать значения сегментных регистров (как следствие он не может использовать более 4 сегментов). Следовательно, в нашей архитектуре модуль не может выполнять следующие команды

- Команды пересылки вида `mov SR, op2`, то есть загружать в сегментный регистр значение длинного регистра `r16` или слова из памяти `m16`. Напомним, что параметр `SR` может принимать значения сегментных регистров `DS`, `ES` и `SS`.
- Команды чтения из стека в сегментный регистр `pop SR`.
- Команды *дальнего* возврата `ret` в свой *собственный* сегмент кода, т.е. выполнять дальний возврат внутри самого модуля.

Кроме того, такой модуль *нельзя* перемещать в другое место памяти во время выполнения системного вызова по команде `int i8`, так как возврат из процедуры-обработчика прерывания производится по команде `iret`, которая, как мы знаем, тоже загружает кодовый сегментный регистр.

Из этих свойств видно, что хорошим кандидатом на перемещаемый модуль является процедура на Ассемблере (понять это). На первых ЭВМ перемещаемость была очень полезным свойством выполняемого модуля, так как позволяла уменьшить операции обмена с медленной внешней памятью. На современных ЭВМ, однако, появился механизм *виртуальной памяти*, который позволяет использовать большое логическое адресное пространство, и перемещаемость перестала быть важной характеристикой исполняемых модулей. Виртуальную память Вы будете изучать в курсе "Системное программное обеспечение".

11.2.2. Повторно-выполняемые модули.

Повторное выполнение модуля предполагает, что, будучи один раз загруженным в оперативную память, он допускает своё многократное исполнение (т.е. вход в начало этого модуля после его завершения). Естественно, что процедуры и функции по определению являются повторно используемыми, однако для основной (головной) программы дело обстоит сложнее. Для головной программы на Ассемблере должно выполняться следующее условие: программа не должна менять переменные с начальными значениями или, в крайнем случае, восстанавливать эти значения перед окончанием программы. Например, если в программе на Ассемблере имеются предложения

```
X      dw      1
      . . .
      mov     X, 2
```

то программа будет повторно используемой только тогда, когда она восстанавливает первоначальное значение переменной `X` перед выходом из программы. В настоящее время это свойство программы не имеет большого значения, потому что появилось более сильное свойство модуля – быть *повторно-входимым* (реентерабельным).

11.2.3. Повторно-входимые (реентерабельные) модули.

Свойство исполняемого модуля быть реентерабельным (иногда говорят – параллельно используемым) является очень важным, особенно при написании системных программ. Модуль называется реентерабельным, если он допускает повторный вход в своё начало *до* выхода из этого модуля (для модулей на Ассемблере, как мы знаем, выход производится по команде возврата `ret` для процедур, по команде `iret` для обработчиков прерываний или по макрокоманде `finish` для основной программы).

Особо подчеркнём, что повторный вход в такой модуль производит *не сам* этот модуль, используя прямую или косвенную рекурсию, а *другие программы*, обычно при обработке сигналов прерываний. Таким образом, внутри реентерабельного модуля могут располагаться *несколько* текущих точек выполнения программы. Мы уже сталкивались с такой ситуацией при изучении системы прерываний, когда выполнение процедуры-обработчика прерывания с некоторым номером

могло быть прервано новым сигналом прерывания с таким же номером, так что производился *повторный вход* в начало *этой же* процедуры до окончания обработки текущего прерывания.

Каждая текущая точка выполнения реентерабельной программы имеет, как мы уже упоминали, своё *поле сохранения* (иногда его называют *контекстом* процесса). При прерывании выполнения программы на этом поле сохраняются, в частности, все регистры, определяющие текущую точку выполнения (как сегментные регистры и регистр флагов, так и регистры общего назначения, и регистры для работы с вещественными числами).

Главное отличие реентерабельных программ от обычных *рекурсивных* процедур заключается именно в том, что при *каждом* входе в реентерабельную программу порождается новая текущая точка её выполнения и новое поле сохранения. Это позволяет продолжить выполнение реентерабельной программы с *любой* из этих нескольких текущих точек выполнения программы, восстановив значения всех регистров из поля сохранения этой точки программы.

Ниже перечислены основные свойства, которыми должен обладать модуль на Ассемблере, чтобы быть реентерабельным.

- Модуль не меняет сегменты кода.
- Модуль либо совсем не имеет собственных сегментов данных (т.е. использует сегмент данных другого модуля, на этот сегмент данных, как обычно, указывает значение регистра DS), либо при каждом входе получает новые *копии* своих сегментов данных.
- При каждом входе в модуль он получает новый сегмент стека, пустой для основной программы и с копиями фактических параметров и адресом возврата для процедуры. Вообще говоря, этот сегмент стека является возможным местом и для расположения *области сохранения* модуля, хотя на современных ЭВМ область сохранения обычно размещается в так называемом пространстве ядра операционной системы, это место является *защищённым* от изменения со стороны программ обычных пользователей.

Реентерабельность является особенно важной при написании системных программ. Это следует из того, что если некоторая программа (например, компилятор с Ассемблера) является реентерабельной, то в оперативной памяти достаточно иметь только одну копию этой программы, которая может одновременно использоваться при компиляции любого числа программ на Ассемблере (отсюда второе название таких модулей – параллельно используемые).¹

В современных ЭВМ большинство системных программ являются реентерабельными.

12. Макросредства языка Ассемблер.

Сейчас мы переходим к изучению очень важной и сложной темы – *макросредств* в языках программирования.² С этим понятием мы будем знакомиться постепенно, используя примеры из макросредств нашего языка Ассемблера. Здесь следует подчеркнуть, что для полного изучения макросредств Ассемблера обязательно требуется изучение учебника [5].

Заметим, что мы не случайно будем изучать именно макросредства языка Ассемблера, а не макросредства языков программирования высокого уровня (Паскаля, С и т.д.). Всё дело в том, что макросредства в языках высокого уровня чаще всего примитивны (не развиты) и не обеспечивают тех возможностей, которые мы должны изучить в макросредствах. Именно поэтому для уровня университетского образования приходится изучать достаточно развитые макросредства языка Ассемблер.

Макросредства по существу являются *алгоритмическим языком*, встраиваемым в некоторый другой язык программирования, который в этом случае называется *макроязыком*. Например, изучаемые нами макросредства встроены в язык Ассемблера, который называется Макроассемблером. Таким образом, программа на Макроассемблере содержит в себе запись *двух* алгоритмов: один на макроязыке, а второй – собственно на языке Ассемблера.

¹ Как мы узнаем позже, на однопроцессорной ЭВМ в каждый момент времени компилируется только одна программа, но по сигналам прерывания от внутренних часов компьютера может производиться быстрое переключение с одной программы на другую, так что создаётся впечатление, что компилируются (хотя и более медленно) сразу несколько программ.

² Макросредства могут использоваться также и в формальных языках, не являющихся языками программирования, но этот вопрос далеко выходит за рамки нашего курса.

Как мы знаем из курса первого семестра, у каждого алгоритма должен быть свой *исполнитель*. Исполнитель алгоритма на макроязыке называется *макропроцессором*, а исполнителем алгоритма на Ассемблере является, в конечном счете, компьютер. (Не надо путать макропроцессор с процессором компьютера: макропроцессор – это программа, а не часть аппаратуры ЭВМ). Результатом работы макропроцессора (этот исполнитель работает *первым*) является программный модуль на "чистом" языке Ассемблера, *без* макросредств. Иногда говорят, что макропроцессор *генерирует* модуль на Ассемблере. На рис. 12.1 показана схема работы этих двух исполнителей (их часто называют общим именем – Макроассемблер). Программные модули пользователя на этом рисунке заключены в прямоугольники, а системные программы – в прямоугольники с закруглёнными углами.

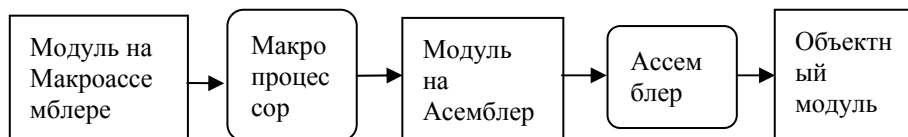


Рис. 12.1. Схема работы Макроассемблера.

Работа макропроцессора по обработке макросредств программного модуля называется *макропроцессированием*. Изучение макросредств языка Ассемблера мы начнём с уже знакомых нам *макрокоманд*. До сих пор мы говорили, что при обработке программы на Ассемблере на место макрокоманды по определённым правилам подставляется некоторый набор предложений языка Ассемблер. Теперь пришло время подробно изучить, как это делается.

Каждое предложение Ассемблера, являющееся макрокомандой, имеет, как мы знаем, обязательное поле – код операции, который является *именем* макрокоманды. Именно по коду операции макропроцессор будет определять, что это именно макрокоманда, а не какое-нибудь другое предложение языка Ассемблер. Коды операций макрокоманд являются *именами пользователя*, а все остальные коды операций – *служебными именами*.¹ Кроме того, у макрокоманды есть (возможно, пустой) список фактических параметров:²

<имя макрокоманды> [<список фактических параметров>]

Итак, каждая макрокоманда имеет имя. Обработка макрокоманды макропроцессором начинается с того, что он, просматривая текст модуля от данной макрокоманды *вверх*, ищет специальную конструкцию Макроассемблера, которая называется *макроопределением* (на жаргоне программистов – *макросом*). Каждое макроопределение имеет имя, и поиск заканчивается, когда макропроцессор находит макроопределение с тем же именем, что и у макрокоманды. Здесь надо сказать, что макропроцессор не считает ошибкой, если в программе будут несколько одноимённых макроопределений, он выбирает из них первое, встреченное при просмотре программы *вверх* от макрокоманды. Говорят, что новое макроопределение *переопределяет* одноимённое макроопределение, описанное ранее.

Макроопределение в нашем Макроассемблере имеет следующий синтаксис:

<имя> **macro** [<список формальных параметров>]

Тело макро-
определения

endm

Первая строка является директивой – *заголовком* макроопределения, она определяет его имя и, возможно, список *формальных параметров*. Список формальных параметров – это (возможно пустая) последовательность *имён*, разделённых запятыми. Тело макроопределения – это набор (возможно пустой) предложений языка Ассемблера (среди них могут быть и предложения, относящиеся к

¹ В наших программах мы выделяли имена макрокоманд жирным шрифтом (**finish**, **inint** и т.д.), хотя это, конечно, не совсем правильно, так как это не служебные слова Макроассемблера.

² Если у макрокоманды есть метка, то считается, что эта метка задаёт отдельное предложение Ассемблера, состоящее только из данной метки. Другими словами, макрокоманда с меткой:

<метка>[:]<имя макрокоманды> [<список фактических параметров>]

эквивалентна двум таким предложениям Ассемблера:

<метка>[:]

<имя макрокоманды> [<список фактических параметров>]

макросредствам языка). Заканчивается макроопределение директивой **endm** (обратите внимание, что у этой директивы нет метки, как, скажем, у директивы конца описания процедуры).

Макроопределение может находиться в любом месте программы **до** первой макрокоманды с таким же именем, но хорошим стилем программирования считается описание всех макроопределений в начале программного модуля. Итак, каждой макрокоманде должно быть поставлено в соответствие макроопределение с таким же именем, иначе в программе фиксируется синтаксическая ошибка. Макроассемблер допускает *вложенность* одного макроопределения внутри другого (в отличие от процедур, вложенность которых на Ассемблере, как мы уже знаем, не допускается), однако это редко используется в практике программирования.

Далее, как мы знаем, в макрокоманде на месте поля операндов может задаваться список *фактических параметров*. Как видим, здесь просматривается большое сходство с механизмом процедур в языке Паскаль, где в *описании процедуры* мог задаваться список формальных параметров, а в *операторе процедуры* – список фактических параметров. Однако на этом сходство между Паскалем и Макроассемблером заканчивается, и начинаются различия.

Каждый фактический параметр макрокоманды является *строкой символов* (возможно пустой). Хорошим аналогом являются строки типа **String** в Турбо-Паскале, однако, фактические параметры *не заключаются в апострофы*. Фактические параметры, если их более одного, разделяются запятыми или *пробелами*. Если фактический параметр расположен не в конце списка параметров и является пустой строкой, то его позиция выделяется запятой, например:

```
Мутасго X, , Y; Три параметра, второй пустой
Мутасго , A B; Три параметра, первый пустой
```

Как видим, в отличие от Паскаля, все параметры макроопределения одного типа – это строки символов, другими словами, всегда есть *соответствие по типу* между фактическими и формальными параметрами. Далее, в Макроассемблере не должно соблюдаться соответствие в числе параметров: формальных параметров может быть как меньше, так и больше, чем фактических. Если число фактических и формальных параметров не совпадает, то макропроцессор выходит из этого положения совсем просто. Если фактических параметров больше, чем формальных, то лишние (последние) фактические параметры отбрасываются, а если фактических параметров не хватает, по недостающие (последние) фактические параметры считаются *пустыми* строками символов.

Рассмотрим теперь, как макропроцессор обрабатывает (*выполняет*) макрокоманду. Сначала, как мы уже говорили, он ищет соответствующее макроопределение, затем начинает передавать фактические параметры (строки символов, возможно пустые) на место формальных параметров (имён). В Паскале, как мы знаем, существуют два способа передачи параметров – по значению и по ссылке. В Макроассемблере реализован другой способ передачи фактических параметров макрокоманды в макроопределение, его нет в Паскале. Этот способ называется передачей *по написанию* (иногда – передачей по имени). При таком способе передачи параметров все *имена* формальных параметров в теле макроопределения заменяются соответствующими им фактическими параметрами (строками символов).¹

Далее начинается просмотр тела макроопределения и поиск в нём предложений, относящихся к макросредствам, например, макрокоманд. Все предложения в макроопределении, относящиеся к макросредствам, обрабатываются макропроцессором так, что в результате получается набор предложений на "чистом" языке Ассемблера, который называется *макрорасширением*. Последним шагом в обработке макрокоманды является подстановка полученного макрорасширения на место макрокоманды, это действие называется *макроподстановкой*. На рис. 12.2 показана схема обработки макрокоманды.

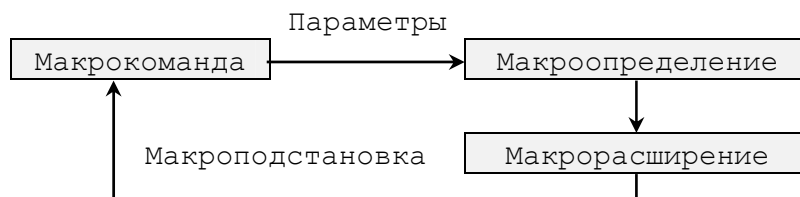


Рис. 12.2. Схема обработки макрокоманды.

¹ Это несколько упрощённое описание действий Макропроцессора при передаче параметров, позже мы сделаем существенные уточнения.

Из рассмотренного механизма обработки макрокоманд вытекает главное применение этого макросредства при программировании на Ассемблере. Как можно заметить, если нам необходимо выполнить в программе некоторое достаточно сложное действие, можно идти двумя путями. Во-первых, можно написать *процедуру* и вызывать её, передавая ей фактические параметры. Во-вторых, можно написать *макроопределение*, в теле которого реализовать нужное нам действие, и обращаться к этому макроопределению по соответствующей макрокоманде, также передавая необходимые параметры.

В дальнейшем мы сравним эти два метода, а пока отметим, что написание макроопределений – это хороший способ *повысить уровень* языка программирования. Действительно, макрокоманда по синтаксису практически ничем не отличается от команд Ассемблера, но может задавать весьма сложное действие. Вспомним, например, макрокоманду **inint** для ввода целого значения. Соответствующее ей макроопределение по своим функциям похоже на процедуру Read языка Паскаль и реализует достаточно сложный алгоритм по преобразованию вводимых символов в значение целого числа. С точки же зрения программиста в языке Ассемблера как бы появляется **новая** машинная команда, предназначенная для ввода целых чисел. Говорят, что при помощи макросредств можно *расширить* язык Ассемблера, как бы вводя в него *новые* команды, необходимые программисту.

Теперь пришло время написать наше собственное простое макроопределение и на его основе продолжить изучение работы макропроцессора. Предположим, что в программе на Ассемблере приходится неоднократно выполнять оператор присваивания вида $z:=x+y$, где x, y и z – целочисленные операнды размером в слово. В общем случае для реализации этого оператора присваивания необходимы три команды Ассемблера, например:

```
mov ax, X
add ax, Y
mov Z, ax
```

Естественно, что программисту было бы более удобно, если бы в языке Ассемблера существовала *трёхадресная* команда, которая реализовывала бы такой оператор присваивания, например, команда с именем Sum:

```
Sum Z, X, Y; Z:=X+Y
```

Потребуем, чтобы первый операнд этой команды мог иметь форматы r16 и m16, а второй и третий – форматы i16, m16 и r16. Такой команды, как мы знаем, в нашем компьютере нет, но можно создать новую *макрокоманду*, которая работала бы так, как нам надо. Для этого можно написать, например, такое макроопределение:

```
Sum macro Z, X, Y
    mov ax, X
    add ax, Y
    mov Z, ax
endm
```

Вот теперь, если в нашей программе есть, например, описания переменных

```
A    dw    ?
B    dw    ?
C    dw    ?
```

и надо выполнить присваивание $C:=A+B$, то программист может записать это в виде одного предложения Ассемблера – макрокоманды

```
Sum C, A, B
```

Увидев такую макрокоманду, макропроцессор (а он работает *раньше* Ассемблера),¹ найдёт соответствующее макроопределение с именем Sum и построит следующее макрорасширение:

```
mov ax, A
add ax, B
```

¹ Вообще говоря, на самом деле Макропроцессор и Ассемблер обрабатывают программу одновременно, предложение за предложением. Как мы вскоре узнаем, первыми каждое предложение обрабатывают специальные программы Ассемблера, которые называются лексическим и синтаксическим анализаторами, а затем, если нужно, это предложение обрабатывает Макропроцессор. Однако конечный этап компиляции – генерация объектного модуля – выполняется Ассемблером уже *после* полного завершения работы Макропроцессора.

```
mov C, ax
```

Это макрорасширение и будет подставлено в текст нашей программы вместо макрокоманды

```
Sum C, A, B
```

(произойдет *макроподстановка* макрорасширения на место макрокоманды).

Программист доволен: теперь текст его программы значительно сократился, и программа стала более понятной. Таким образом, можно приблизить уровень языка Ассемблер (как мы говорили, это язык *низкого* уровня) к языку высокого уровня (например, Паскалю). В этом, как мы уже говорили, и состоит одно из назначений механизма макроопределений и макрокоманд – поднять уровень языка, в котором они используются.

Далее, однако, программист может заметить, что некоторые макрокоманды работают не совсем хорошо. Например, на место макрокоманды с *допустимым* форматом параметров

```
Sum C, ax, B
```

будет подставлено макрорасширение

```
mov ax, ax
```

```
add ax, B
```

```
mov C, ax
```

Первая команда в этом макрорасширении, хотя и не влияет на правильность алгоритма, но явно лишняя и портит всю картину. Естественно, что нам хотелось бы убрать из макрорасширения первую команду, если второй операнд макрокоманды является регистром *ax*. Другими словами, мы бы хотели делать *условную макрогенерацию* и давать макропроцессору указания вида "если выполняется такое-то условие, то вставляй в макрорасширение вот эти предложения Ассемблера, а иначе – не вставляй". На языке Паскаль такие указания мы записывали в виде *условных операторов*. Ясно, что и в макроязыке (а, как мы говорили, это тоже алгоритмический язык) тоже должны допускаться аналогичные *условные макрооператоры*.

В Макроассемблере условные макрооператоры принадлежат к средствам так называемой *условной компиляции*. Легко понять смысл этого названия, если вспомнить, что при выполнении таких макрооператоров меняется вид компилируемой программы на Ассемблере, в ней появляются те или иные группы предложений. Мы изучим только самые употребительные макрооператоры, которые будем использовать в наших примерах, для полного изучения этой темы необходимо обратиться к учебнику [5].

Итак, мы хотим вставить в наше макроопределение условный макрооператор с таким смыслом:

"Если второй параметр *X* не идентичен (не совпадает) с именем регистра *ax*, то тогда необходимо вставить в макрорасширение предложение `mov ax, X`".

На Макроассемблере наше макроопределение с именем *Sum* в этом случае будет иметь такой вид:

```
Sum macro Z, X, Y
  ifdif <X>, <ax>
    mov ax, X
  endif
  add ax, Y
  mov Z, ax
endm
```

Поясним работу этого условного макрооператора. Так как его аргументы – строки символов, т.е. он проверяет совпадение или несовпадение двух строк текста, то надо как-то задать эти строки. В качестве ограничителей строки в Макроассемблере выбраны угловые скобки, так что выражение *<ax>* эквивалентно записи *'ax'* в Паскале. Таким образом, семантику нашего условного макрооператора на языке Паскаль можно записать как

```
if 'X' <> 'ax' then
```

Вставить в макрорасширение `mov ax, X`

Вся тонкость здесь, однако, состоит в том, что на место имени формального параметра *X* внутри кавычек подставляется строка – второй фактический параметр макрокоманды *Sum*.

Изучая дальше наше макроопределение можно заметить, что и на место, например, макрокоманды

```
Sum ax, ax, 13
```

подставится макрорасширение


```

add ax,13
mov ax,ax

```

с лишней последней строкой. Чтобы это исправить, нам придётся снова изменить наше макроопределение, например, так:

```

Sum macro Z,X,Y
ifdif <X>,<ax>
    mov ax,X
endif
    add ax,Y
ifdif <Z>,<ax>
    mov ax,Z
endif
endm

```

Вот теперь на место макрокоманды

```

Sum ax,ax,13

```

будет подставляться ну о-чень хорошее макрорасширение

```

add ax,13

```

Дальнейшее изучение нашего макроопределения, однако, выявит новую неприятность: макрокоманда

```

Sum ax,Y,ax

```

порождает *неправильное* макрорасширение

```

mov ax,Y
add ax,ax

```

Это, конечно, не то же самое, что $ax := ax + Y$. Мы можем справиться с этой новой проблемой, снова усложнив наше макроопределение, например, так:

```

Sum macro Z,X,Y
ifidn <ax>,<Y>
    add ax,X
else
    ifdif <ax>,<X>
        mov ax,X
    endif
    add ax,Y
endif
ifdif <Z>,<ax>
    mov Z,ax
endif
endm

```

В новой версии нашего макроопределения мы использовали *вложенные* условные макрооператоры. Первый из них с именем **ifidn** сравнивает свои аргументы-строки текста и вырабатывает значение **True**, если они идентичны (равны). Как и в условном операторе языка Паскаль, в условном макрооператоре может присутствовать ветвь **else**, которая выполняется, если при сравнении строк получается значение **false**. Обязательно проверьте, что для нашей последней макрокоманды `Sum ax,Y,ax` сейчас тоже получается правильное макрорасширение.

Не нужно, конечно, думать, что теперь мы написали идеальное макроопределение и все проблемы решены. Первая неприятность, которая нас подстерегает, связана с самим механизмом сравнения строк на равенство в условных макрооператорах. Рассмотрим, например, что будет, если записать в нашей программе макрокоманду

```

Sum AX,X,Y

```

На её место будет подставлено макрорасширение

```

mov ax,X
add ax,Y
mov AX,ax

```

с совершенно лишней последней строкой. Причина здесь в том, что макропроцессор, естественно, считает строки `<AX>` и `<ax>` *не идентичными*, со всеми вытекающими отсюда последствиями.

Другая трудность подстерегает нас, если мы попытаемся использовать в нашей программе, например, макрокоманду

```
Sum ax, bx, dl
```

После обработке этой макрокоманды будет построено макрорасширение

```
mov ax, bx
add ax, dl
```

Это макрорасширение макропроцессор "со спокойной совестью" подставит на место нашей макрокоманды. Конечно, позже, на втором этапе, когда Ассемблер будет анализировать правильность программы, для команды `add ax, dl` зафиксирована ошибка – несоответствие типов операндов. Это очень важный момент – ошибка зафиксирована не при обработке макрокоманды `Sum ax, bx, dl`, как происходит при обработке синтаксически неправильных обычных команд Ассемблера, а позже и уже при анализе не макрокоманды, а макрорасширения. В этом отношении наша макрокоманда уступает обычным командам Ассемблера. Нам бы, конечно, хотелось, чтобы диагностика об ошибке (и лучше на русском языке) выдавалась уже на этапе обработки макрокоманды макропроцессором. Немного позже мы научимся, как это делать.

Итак, мы обратили внимание на две трудности, которые, как Вы догадываетесь, можно преодолеть, снова усложнив наше макроопределение. Мы, однако, сделаем это не на примере макрокоманды суммирования двух чисел, а на примерах других, более сложных, задач.

Рассмотрим теперь типичный ход мыслей программиста при разработке нового макроопределения. Как мы знаем, для того, чтобы в Ассемблере выполнить вывод текстовой строки, расположенной в сегменте данных, например,

```
T db 'Строка для вывода$'
```

необходимо загрузить адрес начала этой строки на регистр `dx` и выполнить макрокоманду `outstr`:

```
mov dx, offset T
outstr
```

Ясно, что это не совсем то, что хотелось бы программисту, ему было бы удобнее выводить строку текста, например, так:

```
outtxt 'Строка для вывода'
```

Осознав такую потребность, программист решает написать новое макроопределение, которое позволяет именно так выводить текстовые строки. Проще всего построить новое макроопределение `outtxt` на базе уже существующего макроопределения `outstr`, например, так:

```
outtxt macro X
    local L, T
    jmp L
T db X
db '$'
L: push ds; запоминание ds
    push cs
    pop ds; ds:=cs
    push dx; сохранение dx
    mov dx, offset T
    ostr
    pop dx; восстановление dx
    pop ds; восстановление ds
endm
```

Обратите внимание, что второй фактический параметр (строку символов) – наше макроопределение располагает внутри макрорасширения (т.е. в сегменте кода). А так как макрокоманда `ostr` "думает", что выводит текст из сегмента данных, то мы временно совместили сегменты данных и кода, загрузив в регистр `ds` значение регистра `cs`.

В этом макроопределении мы использовали новое макросредство – директиву

```
local L, T
```

Эта директива объявляет имена `L` и `T` локальными именами макроопределения `outtxt`. Как и локальные имена, например, в языке Паскаль, они не видны *извне* макроопределения, следовательно, в других частях этого программного модуля также могут использоваться эти имена. Директива `lo-`

cal для макропроцессора имеет следующий смысл. При каждом входе в макроопределение локальные имена, перечисленные в этой директиве, получают новые уникальные значения. Обычно макропроцессор выполняет это совсем просто: при первом входе в макроопределение заменяет локальные имена L и T, например, на имена ??0001 и ??0002, при втором входе – на имена ??0003 и ??0004 и т.д. (учтите, что в Ассемблере символ ? относится к *буквам* и может входить в имена).

Назначение директивы **local** становится понятным, когда мы рассмотрим, что будет, если эту директиву убрать из нашего макроопределения. В этом случае у двух макрорасширений макрокоманды **outtxt** будут внутри *одинаковые* метки L и T, что повлечёт за собой ошибку, которая будет зафиксирована на следующем этапе, когда Ассемблер станет переводить программу на объектный язык.

В качестве следующего примера рассмотрим такую проблему. Мы выводим значения знаковых целых чисел, используя макрокоманду **outint**. Эта макрокоманда, однако, позволяет выводить целые значения только форматов r16, m16 и i16. Если программисту необходимо часто выводить целые числа ещё и в форматах r8, m8 и i8, то он, естественно, захочет написать для себя новое макроопределение, которое обеспечивает такие более широкие возможности. Используя макроопределение **outint** как базовое, мы напишем новое макроопределение с именем **oint**. Ниже приведён вид этого макроопределения.

```
oint macro X
    local K
    ifb <X>
        %out Нет аргумента в oint!
        .err
        exitm
    endif
    push ax
    K=0
    irp i, <al, ah, bl, bh, cl, ch, dl, dh,
        AL, AH, BL, BH, CL, CH, DL, DH,
        Al, Ah, Bl, Bh, Cl, Ch, Dl, Dh,
        aL, aH, bL, bH, cL, cH, dL, dH>
        ifidn <i>, <X>
            K=1
        endif
    endm
    if K EQ 1 or type X EQ byte
        push ax
        mov al, X
        cbw
        outint ax
        pop ax
    else
        outint X
    endif
endm
```

В макроопределении **oint** используется много новых макросредств, поэтому мы сейчас подробно прокомментируем его работу. Вслед за заголовком макроопределения находится уже знакомая нам директива с объявлением локального имени K, затем располагается условный макрооператор с именем **ifb**, который вырабатывает значение **true**, если ему задан *пустой* параметр X (пустая строка символов).

Директива Ассемблера **%out** предназначена для вывода во время компиляции диагностики об ошибке, текст диагностики программист располагает сразу вслед за первым пробелом после имени директивы **%out**. Таким образом, программист может задать *свою собственную* диагностику, которая будет выведена при обнаружении ошибки в макроопределении. В нашем примере диагностика "Нет аргумента в oint!" выводится, если программист забыл задать аргумент у макрокоманды **oint**.

Эта диагностика выводится на так называемое устройство стандартного вывода (**stdout**), а её копия – на устройство стандартной диагностики об ошибках (**stderr**).

После того, как макроопределение обнаружит ошибку в своих параметрах, у программиста есть две возможности. Во-первых, можно считать выданную диагностику *предупредительной*, и продолжать компиляцию программы с последующим получением (или, как говорят, *генерацией*) объектного модуля. Во-вторых, можно считать обнаруженную ошибку *фатальной*, и запретить генерацию объектного модуля (Ассемблер, однако, будет продолжать проверку остальной части программы на наличие других ошибок).

В нашем макроопределении мы приняли второе решение и зафиксировали фатальную ошибку, о чём предупредили Ассемблер с помощью директивы **.err**. Получив эту директиву, Ассемблер вставит в протокол своей работы (листинг) диагностику о фатальной ошибке, обнаруженной в программе, эта ошибка носит обобщённое название **forced error** (т.е. ошибка, "навязанная" Ассемблеру Макропроцессором). Копия сообщения о фатальной ошибке посылается и в стандартный вывод **stderr** (обычно он связан с дисплеем).

После выдачи директивы **.err** дальнейшая обработка макроопределения не имеет никакого смысла, и мы прервали эту обработку, выдав макропроцессору директиву **exitm**. Эта директива прекращает процесс построения макрорасширения,¹ и в нём остаются только те строки, которые попали туда до выполнения директивы **exitm**. Например, если вызвать наше макроопределение макрокомандой **oint** без параметра, то будет получено такое макрорасширение:²

```
Нет аргумента в oint!
```

```
.err
```

Именно оно и будет подставлено на место ошибочной макрокоманды без параметра. На этом примере мы показали, как программист может предусмотреть свою собственную реакцию и диагностику на ошибку в параметрах макрокоманды. Обратите внимание, что реакция на ошибку в макрокоманде производится именно на этапе обработки самой макрокоманды, а не позже, когда Ассемблер будет анализировать полученное макрорасширение.

Следующая директива Макроассемблера

```
K=0
```

является макрооператором *присваивания* и показывает использование нового важного понятия из макросредств нашего Ассемблера – так называемых *переменных периода генерации*. Это достаточно сложное понятие, и сейчас мы начнём разбираться, что это такое.

Ещё раз напомним, что макросредства по существу являются алгоритмическим языком, поэтому полезно сравнить эти средства, например, с таким алгоритмическим языком, как Паскаль. Сначала мы познакомились с макроопределениями и макрокомандами, которые являются аналогами соответственно описаний процедур и операторов процедур Паскаля. Затем мы изучили некоторые из условных макрооператоров, являющихся аналогами условных операторов Паскаля, а теперь пришла очередь заняться аналогами *операторов присваивания, переменных и циклов* языка Паскаль в наших макросредствах.

Макропеременные в нашем макроязыке называются *переменными периода генерации*. Такое название призвано подчеркнуть время существования этих переменных: они порождаются только на период обработки исходного программного модуля на Ассемблере и генерации объектного модуля.

Как и переменные в Паскале, переменные периода генерации в Макроассемблере бывают *глобальные* и *локальные*. Глобальные переменные уничтожаются только после построения всего объектного модуля, а локальные, как всегда, после выхода из того макросредства, в котором они порождены. В нашем Макроассемблере различают локальные переменные периода генерации *макроопределения* (они порождаются при входе в макроопределение и уничтожаются после

¹ Точнее, директива **exitm** производится выход вниз за ближайшую директиву **endm**, которая, как мы вскоре узнает, может задавать конец не только макроопределения, но и макроциклов. Таким образом, **exitm** прекращает обработку той части макроопределения, которая ограничена ближайшей вниз директивой **endm**.

² Вообще говоря, во время компиляции стандартный вывод **stdout** должен быть связан с файлом листинга, а стандартный вывод об ошибках **stderr** – с экраном дисплея. Однако некоторые Ассемблеры (и среди них, к сожалению, MASM-4.0) могут во время компиляции связывать стандартный вывод **stdout** тоже с экраном, а вывод в листинг производить путём указания имени конкретного файла. В этом случае диагностика, заданная директивой **%out** в макрорасширение (и в листинг) не попадёт, а будет *дважды* выведена на экран.

построения макрорасширения) и локальные переменные – параметры макроцикла с именем **irp** (они уничтожаются после выхода из этого цикла). В нашем последнем макроопределении локальной является переменная периода генерации с именем **K**, о чём объявлено в директиве **local**, и переменная периода генерации с именем **i**, которая является локальной в макроцикле **irp**.

Переменные периода генерации могут принимать целочисленные или (в особых случаях) строковые значения. В нашем Ассемблере нет специальной директивы (аналога описания переменных **var** в Паскале), при выполнении которой *порождаются* переменные периода генерации (т.е. им отводится место в памяти макропроцессора). У нас переменные периода генерации *порождаются автоматически*, при присваивании им первого значения. Так, в нашем макроопределении локальная переменная периода генерации с именем **K** порождается при выполнении макрооператора присваивания **K=0**, при этом ей, естественно, присваивается нулевое значение.

Следующая важная компонента макросредств Ассемблера – это макроциклы (которые, конечно, должны быть в макросредствах, как в любом "солидном" алгоритмическом языке высокого уровня).¹ В нашем макроопределении мы использовали один из видов макроциклов с именем **irp**. Этот макроцикл называется циклом с параметром и имеет такой синтаксис (параметр цикла мы назвали именем **i**):

```
irp i, <список цикла>
    тело цикла
endm
```

Параметр цикла является локальной в этом цикле переменной периода генерации, которая может принимать строковые значения. Список цикла (он заключается в угловые скобки) является последовательностью (возможно пустой) текстовых строк, разделённых запятыми (напомним, что в макропроцессоре строки не заключаются в апострофы). В нашем последнем макроопределении такой список макроцикла

```
<a1, ah, b1, bh, c1, ch, d1, dh, AL, AH, BL, BH, CL, CH, DL, DH,
    Al, Ah, Bl, Bh, Cl, Ch, Dl, Dh, aL, aH, bL, bH, cL, cH, dL, dH>
```

Этот список содержит 32 двухбуквенные текстовые строки. Вообще говоря, его необходимо записывать в виде одного предложения Макроассемблера (что возможно, так как максимальная длина предложения в Ассемблере около 130 символов), но в нашем примере мы для удобства изобразили его в две строки. При написании текста макроопределения **oint** мы изобразили этот список даже в виде четырёх строк, что, конечно, тоже *неправильно* и сделано только для удобства восприятия нашего примера.

Выполнение макроцикла с именем **irp** производится по следующему правилу. Сначала переменной цикла присваивается первое значение из списка цикла, после чего выполняется тело цикла, при этом все вхождения в это тело параметра цикла заменяются на текущее значение этой переменной периода генерации. После этого параметру цикла присваивается следующее значение из списка цикла и т.д. Так, в нашем примере тело цикла будет выполняться 32 раза, при этом переменная **i** будет последовательно принимать значение строк текста **a1, ah, b1** и т.д.

Как можно заметить, целью выполнения макроцикла в нашем примере является присваивание переменной периода генерации **K** значение единицы, если параметр макрокоманды совпадает по написанию с именем одного из коротких регистров компьютера, причём это имя может задаваться как большими, так и малыми буквами алфавита в любой комбинации. Это позволяет распознать имя короткого регистра, как бы его ни записал пользователь, и присвоить переменной **K** значение единица, в противном случае переменная **K** сохраняет нулевое значение.

Далее в макроопределении расположен условный макрооператор нового для нас вида, который, однако, наиболее похож на условный оператор языка Паскаль:

```
if <логическое выражение>
    ветвь then
else
    ветвь else
endif
```

¹ Заметим, что в языках высокого уровня обычно есть далеко не все из рассматриваемых нами макросредств.

На этом примере мы познакомимся с логическими выражениями макропроцессора. Эти выражения весьма похожи на логические выражения Паскаля, только вместо логических констант **true** и **false** используются соответственно целые числа 1 и 0, а вместо знаков операций отношения – мнемонические двухбуквенные имена, которые перечислены ниже:

EQ вместо =
NE вместо <>
LT вместо <
LE вместо <=
GT вместо >
GE вместо >=

Таким образом, заголовок нашего условного макрооператора

```
if K EQ 1 or type X EQ byte
```

эквивалентен такой записи на Паскале

```
if (K=1) or (type X = byte) then
```

Заметим, что в Паскале нам необходимо использовать круглые скобки, так как операция отношения = имеет *меньший* приоритет, чем операция логического сложения **or**. В Макроассемблере же, наоборот, операции отношения (**EQ**, **GT** и т.д.) имеют более высокий приоритет, чем логические операции (**or**, **and** и **not**), а так как оператор **type** имеет больший приоритет, чем оператор **EQ**, то круглые скобки не нужны. По учебнику [5] Вам необходимо обязательно изучить уровни приоритета всех операторов Ассемблера.

Таким образом, наш условный макрооператор после вычисления логического выражения получает значение **true**, если K=1 (т.е. параметр макрокоманды – это короткий регистр r8) или же для случая, когда **type** X **EQ byte** (т.е. параметр макрокоманды имеет формат m8). В остальных случаях (для параметра форматов r16, m16) логическое выражение имеет значение **false**. Когда это логическое выражение равно **true**, наше макроопределение вычисляет и помещает на регистр ax целочисленное значение, подлежащее выводу. И так как теперь это значение имеет формат r16, то для его вывода можно использовать уже известную нам макрокоманду **outint**, а для значения **false** просто выводить значение параметра X.¹

Необходимо также заметить, что операции отношения **LT**, **GT**, **LE** и **GE**, как правила, рассматривают свои операнды как беззнаковые значения. Исключением является случай, когда макропроцессор "видит", что некоторой переменной периода генерации *явно* присвоено отрицательное значение. Например, рассмотрим следующий фрагмент программы:

```
L:   mov ax,ax; Чтобы была метка, type L=-1
      K=type L; Макропроцессор "видит" беззнаковое K=0FFFFh
      if K LT 0; Верётся K=0FFFFh > 0 => ЛОЖЬ !
      . . .
      K=-1; Макропроцессор "видит" знаковое K=-1
      if K LT 0; Верётся K=-1 < 0 => ИСТИНА !
      . . .
```

Как видим, здесь вопрос весьма запутан, его надо тщательно изучить по учебнику [5].

Не следует, конечно, думать, что мы написали совсем уж универсальное макроопределение для вывода любых целых чисел, которое всегда выдаёт либо правильный результат, либо диагностику об ошибке в своих параметрах. К сожалению, наше макроопределение не будет выводить значения аргументов форматов m8 и m16, если эти аргументы заданы без имён, по которым можно определить их тип, например, вызов `oint [bx]`, будет считаться *ошибочным*. Это связано с тем, что ошибку вызовет оператор **type** [bx].

Кроме того, например, при вызове с помощью макрокоманды

```
oint --8
```

будет получено макрорасширение

¹ Для простоты наше макроопределение никогда не задаёт второй параметр макрокоманды **outint** – ширину поля для вывода целого числа.

```
mov ax, --8
outint ax
```

(т.к. `type --8 = 0`). К сожалению, наш макропроцессор не предоставляет хороших средств, позволяющих выявить синтаксические ошибки такого рода. Показанные выше ошибки будут выявлены уже компилятором с Ассемблера при анализе полученного макрорасширения.

Далее, нам важно понять принципиальное отличие переменных языка Ассемблера и переменных периода генерации. Так, например, переменная Ассемблера с именем `X` может, например, определяться предложением резервирования памяти

```
X dw 13
```

В то время как переменная периода генерации с именем `Y` – макрооператором присваивания

```
Y = 13
```

Главное – это уяснить, что эти переменные имеют разные и непересекающиеся времена существования. Переменные периода генерации существуют только во время компиляции исходного модуля с языка Ассемблер на объектный язык и заведомо уничтожаются до начала счёта, а переменные Ассемблера, наоборот, существуют только во время счёта программы (до выполнения макрокоманды **finish**). Некоторые студенты не понимают этого и пытаются использовать переменную Ассемблера на этапе компиляции, например, пишут такой неправильный условный макрооператор

```
if X EQ 13
```

Это сразу показывает, что они не понимают суть дела, так как на этапе компиляции хотят анализировать *значение* переменной `X`, которая будет существовать только во время *счёта* программы.

В следующем примере мы покажем, как макроопределение может обрабатывать макрокоманды с *переменных* числом фактических параметров. Задачи такого рода часто встают перед программистом. Пусть, например, в программе надо часто вычислять максимальное значение от нескольких знаковых целых величин в формате слова. Для решения этой задачи можно написать макроопределение, у которого будет только *один* формальный параметр, на место которого будет передаваться *список* (возможно пустой) фактических параметров. Такой список в нашем Макроассемблере заключается в угловые скобки. Пусть, например, макроопределение должно вычислить и поместить на регистр `ax` максимальное значение из величин `bx, X, -13, cx`, тогда нужно вызвать это макроопределение с помощью такой макрокоманды (дадим этой макрокоманде имя **maxn**):

```
maxn <bx, X, -13, cx>
```

Здесь *один* фактический параметр, который, однако, является списком, содержащим четыре "внутренних" параметра. Мы будем также допускать, чтобы некоторые параметры из этого списка опускались (т.е. задавались пустыми строками). При поиске максимума такие пустые параметры будем просто отбрасывать. Далее необходимо договориться, что будет делать макроопределение, если список параметров вообще пуст. В этом случае можно, конечно, выдавать диагностику о фатальной ошибке и запрещать генерацию объектного модуля, но мы поступим более "гуманно": будем в качестве результата выдавать самое маленькое знаковое число (это `8000h` в шестнадцатеричной форме записи). Ниже приведён возможный вид макроопределения для решения этой задачи. Наше макроопределение с именем `maxn` будет вызывать *вспомогательное* макроопределение с именем `спрах`. Это вспомогательное макроопределение загружает на регистр `ax` максимальное из двух величин: регистра `ax` и своего единственного параметра `X`.

```
maxn macro X
    mov ax, 8000h; MinInt
irp i, <X>
    ifnb <i>
        cmpax i
    endif
endm

    endm
; Вспомогательное макроопределение
спрах macro X
    local L
```

```

    cmp    ax, X
    jge    L
    mov    ax, X
L:
    endm

```

Поясним работу макроопределения `maxn`, однако сначала, как мы обещали ранее, нам надо существенно уточнить правила передачи фактического параметра (строки символов) на место формального параметра. Дело в том, что некоторые символы, входящие в строку-фактический параметр, являются для макропроцессора *служебными* и обрабатываются по-особому (такие символы называются в нашем макроязыке *макрооператорами*). Ниже приведено описание наиболее интересных макрооператоров, полностью их необходимо изучить по учебнику [5].

- Если фактический параметр заключён в угловые скобки, то они считаются макрооператорами, их обработка заключается в том, что они *отбрасываются* при передаче фактического параметра на место формального.
- Символ восклицательного знака (!) является макрооператором, он *удаляется* из фактического параметра, но при этом блокирует (иногда говорят – *экранирует*) анализ следующего за ним символа на принадлежность к служебным символам (т.е. макрооператорам). Например, фактический параметр `<ab! !+!>` преобразуется в строку `ab!+>`, именно эта строка и передаётся на место формального параметра. Это один из способов, как можно передать в качестве параметров сами служебные символы.
- В том случае, если комментарий начинается с двух символов `;` вместо одного, то это *макрокомментарий*, такой комментарий *не переносится* в макрорасширение.
- Символ `&` является макрооператором, он удаляется макропроцессором из обрабатываемого предложения (заметим, что из двух следующих подряд символов `&` удаляется только один). Данный символ играет роль лексемы – разделителя, он позволяет выделять в тексте имена формальных параметров макроопределения и переменных периода генерации. Например, пусть в программе есть такой макроцикл

```

    K=1
    irp i, <l, h>
        K=K+1
        mov a&i, X&K&i
    endm

```

После обработки этого макроцикла Макропроцессор подставит на это место в текст программы следующие строки:

```

    mov al, X2l
    mov ah, X3h

```

- Символ `%` является макрооператором, он предписывает макропроцессору вычислить следующее за ним *арифметическое выражение* и подставить значение этого выражения вместо знака `%`. Например, после обработки предложений

```

N equ 5
K=1
M equ %(3*K+1)>N

```

Будут получены предложения

```

N equ 5
M equ 4>N

```

Разберём теперь выполнение макроопределения `maxn` для макрокоманды

```

maxn <-13, , bx, Z>

```

На место формального параметра `X` будет подставлена строка символов `-13, , bx, Z`. Таким образом, макроцикл принимает следующий вид

```

irp i, <-13, , bx, Z>
    ifnb <i>
        cmpax i
    endif
endm

```


В теле этого макроцикла располагается условный макрооператор с именем **ifnb**, он проверяет свой параметр и вырабатывает значение **true**, если этот параметр *не является* пустой строкой символов. Таким образом, получается, что в теле макроцикла выполняется условный макрооператор, который только для *непустых* элементов из списка цикла вызывает вспомогательное макроопределение `спрах`. Единственным назначением этого вспомогательного макроопределения является локализация в нём метки `L`, которая таким образом получает уникальное значение для каждого параметра из списка цикла.

Упражнение. Напишите макроопределение `maxn` без использования вспомогательного макроопределения. Надо сразу сказать, что для этого требуется *хорошее* знание языка Ассемблера.

Макроопределения с переменным числом параметров являются достаточно удобным средством при программировании многих задач. Аналогичный механизм (но с совершенно другой реализацией) есть в языке высокого уровня C, который допускает написание функций с переменным числом фактических параметров. Правда, для функций языка C фактических параметров должно быть не менее одного, и значение этого первого параметра должно как-то задавать общее число фактических параметров. Вы будете изучать язык C в следующем семестре.

Теперь мы познакомимся с использованием макросредств для настройки макроопределения на *типы* передаваемых ему фактических параметров. Здесь имеется в виду, что, хотя с точки зрения макропроцессора фактический параметр – это просто строка символов, но с точки зрения Ассемблера у этого параметра может быть тип. Например, в Ассемблере это может быть длинная или короткая целая переменная, константа и т.д. и ясно, что для обработки операндов разных типов требуются и различные команды.

Как мы помним, для языка Паскаль должно соблюдаться строгое соответствие между типами фактических и формальных параметров процедур. А как быть программисту, если, например, ему надо написать функцию для поиска максимального элемента массива, причём необходимо, чтобы в качестве фактического параметра этой функции можно было бы передавать массивы разных типов (с целыми, вещественными, символьными, логическими и т.д. элементами)? На языках высокого уровня хорошо решить эту задачу практически невозможно.¹

Сейчас мы увидим, что макросредства предоставляют программисту простое и элегантное решение, позволяющее *настраивать* макроопределение на тип фактических параметров. Предположим, что нам в программе необходимо суммировать массивы коротких и длинных целых чисел. Для реализации такого суммирования можно написать макроопределение, например, с таким заголовком:

```
SumMas macro X,N
```

В качестве первого параметра этого макроопределения можно задавать массивы коротких или длинных знаковых целых чисел, длина массива указывается во втором параметре. Другими словами, первый операнд макрокоманды `SumMas` может быть формата `m8` или `m16`, а второй – формата `m16` или `i16`. Сумма должна возвращаться на регистре `ax` (т.е. наше макроопределение – в некотором смысле функция). Допускается, чтобы второй параметр был опущен, в этом случае по умолчанию считается, что в массиве 100 элементов. Для простоты наше макроопределение не будет сохранять и восстанавливать используемые регистры, а переполнение при сложении будем игнорировать. Кроме того, не будем проверять допустимость типа второго параметра, например, передачу в качестве второго параметра короткого регистра `r8` или какой-нибудь метки программы. Ниже показан возможный вариант такого макроопределения.

```
SumMas macro X,N
      Local K,L
ifb <X>
```

¹ В стандарте Паскаля можно попытаться сделать элементы массива записями с вариантами, в Турбо-Паскале – использовать так называемые безтиповые массивы. Однако во всех этих случаях, даже если не принимать во внимание отсутствие контроля и понижение надёжности программы, нам придётся передавать в функцию дополнительный параметр, задающий тип элемента массива, а внутри функции будет по-существу находиться оператор **case**, разветвляющий вычисление по разным типам данных. Не спасают здесь положение и объектно-ориентированные языки, всё равно придётся реализовать в программе *несколько* функций с одинаковым именем, компилятор лишь автоматически выберет одну из них, соответствующую типу элементов массива, переданного как фактический параметр.

```

        %out Нет массива!
        .err
        exitm
endif
ifb <N>
        %out Берём длину=100
        mov    cx,100
else
        mov    cx,N
endif
if type X LT 1 or type X GT 2
        %out Плохой тип массива!
        .err
        exitm
endif
        K=word
if type X EQ byte
        K=byte
endif
        lea    bx,X
        xor    dx,dx; Сумма:=0
if K EQ byte
L:      mov    al,[bx]
        cbw
        add    dx,ax
else
L:      add    dx,[bx]
endif
        add    bx,K
        loop  L
        mov    ax,dx
        endm

```

Как видим, наше макроопределение настраивается на тип переданного массива и оставляет в макрорасширении только команды, предназначенные для работы с элементами массива именно этого требуемого типа. Заметьте также, что вся эта работа по настройке на нужный тип параметров производится до начала счёта (на этапе компиляции), то есть на машинном языке получается эффективная программа, не содержащая никаких лишних команд.

Упражнение. Объясните, для чего предназначено показанное ниже макроопределение и как к нему следует обращаться:

```

BegProc macro R
        push  bp
        mov   bp,sp
irp  i,<R>
        push  i
endm
        endm

```

В качестве ещё одного примера рассмотрим следующую задачу. Пусть программисту на Ассемблере необходимо много раз вызывать различные процедуры со стандартным соглашением о связях, передавая им каждый раз досточно большое количество параметров. Естественно, программист хочет автоматизировать процесс вызова процедуры, написав макроопределение, которой позволяет вызывать процедуры почти так же компактно, как и в языках высокого уровня. Например, пусть в Паскале есть описание процедуры с заголовком

```
Procedure P(var X:Mas; N:integer; var Y:integer);
```

Такую процедуру можно, например, вызвать оператором процедуры P(X,400,Y). В Ассемблере для фактических параметров, описанных, например, так:

```
X    dw    400 dup (?)
Y    dw    ?
```

вызов процедуры, как мы знаем, будет, например, производится последовательностью команд:

```
mov  ax,offset X
push ax
mov  ax,400
push ax
mov  ax,offset Y
push ax
call P
```

Для автоматизации вызова процедур напишем такое макроопределение:

```
CallProc macro Name,Param
irp  i,<Param>
    mov  ax,i
    push ax
endm
    call Name
endm
```

Вот теперь вызов нашей процедуры можно производить *одной* макрокомандой

```
CallProc P,<<offset X>,400,<offset Y>>
```

Разумеется, это выглядит не так красиво, как в Паскале, но здесь уж ничего не поделаешь, хотя для любителей макрооператоров Ассемблера можно предложить и альтернативный вызов нашего макроопределения, не перегруженный угловыми скобками:

```
CallProc P,<offset! X,400,offset! Y>
```

Как бы то ни было, главная наша цель достигнута – вызов процедуры стал производится в одно предложение Ассемблера. Заметим, что пустой список фактических параметров можно, как и в Паскале, полностью опускать, например

```
CallProc F
```

Посмотрим теперь, как разработчик нашего макроопределения сможет проконтролировать, что в качестве первого параметра передано *непустое имя* некоторой процедуры (вообще говоря, метки). Проще всего проверить, что переданный параметр не пустой, как мы уже знаем, это можно выполнить, используя условный макрооператор

```
ifb <Name>
    %out Пустое имя процедуры
    .err
    exitm
endif
```

Несколько сложнее обстоит дело, если наш программист захочет проверить, что в качестве первого параметра передано именно имя, а не какая-нибудь недопустимая строка символов. Поставленную задачу можно решить, например, с помощью такого фрагмента на Макроассемблере:

```
Nom=1; Номер символа в имени
irpc i,<Name>
    Err=1; Признак ошибки в очередном символе имени
    if Nom EQ 1
        irpc j,<abcdefghijklmnopqrstuvwxy
            ABCDEFGHIJKLMNOPQRSTUVWXYZ_?@#>
            ifidn <i>,<j>
                Err=0
            endif
        endm
    else
```

```

irpc j,<abcdefghijklmnopqrstuvxyz
      ABCDEFGHIJKLMNOPQRSTUVWXYZ_?@#$0123456789>
      ifidn <i>,<j>
          Err=0
      endif
endm
endif
if Err EQ 1
      exitm; Выход из цикла irpc
endif
endm
if Err EQ 1
      %out Плохой символ i в имени Name
      .err
      exitm; Выход из макроопределения
endif

```

Для анализа символов фактического параметра на принадлежность заданному множеству символов мы использовали макроцикл **irpc**. Выполнение этого макроцикла очень похоже на выполнение макроцикла **irp**, за исключением того, что параметру цикла каждый раз присваивается очередной один символ, входящий в список цикла. В нашем примере в списке цикла мы указали в первом операторе **irpc** все символы, с которых может начинаться имя в Ассемблере (это латинские буквы и некоторые символы, приравнивающиеся к буквам). Во втором операторе **irpc** из нашего примера мы к буквам просто добавили 10 цифр. Заметим, что, как уже отмечалось ранее, каждое предложение надо записывать в одну строку, мы разбили каждый из макроциклов **irpc** на две строки исключительно для удобства чтения нашего примера, что, конечно же, будет классифицировано Ассемблером как синтаксическая ошибка.

Итак, теперь мы убедились, что в качестве первого параметра наше макроопределение получило *некоторое имя*. Но является ли полученное нами имя именно *именем процедуры* (или, в более общем случае, именем команды)? Чтобы выяснить это, в наше макроопределение можно вставить проверку *типа* полученного имени, например, так:

```

if type Name NE -1 and type Name NE -2
      %out Name не имя процедуры!
      .err
      exitm
endif

```

Некоторые характеристики имени можно получить также, применив к этому имени одноместный оператор Ассемблера **.type**. Результатом работы этого оператора является целое значение в формате байта (i8), при этом каждый бит в этом байте, если он установлен в "1", указывает на наличие некоторой *характеристики* имени. Ниже приведены *номера* некоторых битов в байте, которое этот оператор вырабатывает, будучи применённым к своему имени-операнду (напомним, что биты в байте нумеруются справа-налево, начиная с нуля):

$$\text{.type } \langle \text{имя} \rangle = \begin{cases} 0 & \text{- имя команды или процедуры,} \\ 1 & \text{- имя переменной,} \\ 5 & \text{- имя как-то описано,} \\ 7 & \text{- имя описано в } \mathbf{extrn}. \end{cases}$$

Так, например, для имени, описанного в Ассемблере как

```
X      dw      ?
```

оператор **.type** X = 00100001b = 33₁₀. Полностью про этот оператор можно прочитать в учебнике [5], а мы на этом закончим наше краткое знакомство с возможностью макросредств языка Ассемблер.

12.1. Сравнение процедур и макроопределений.

Как мы уже говорили, на Ассемблере один и тот же алгоритм программист, как правило, может реализовать как в виде процедуры, так и в виде макроопределения. Процедура будет вызываться командой **call** с передачей параметров по стандартным или нестандартным соглашениям о связях, а макроопределение – макрокомандой, также с заданием соответствующих параметров. Сравним эти два метода разработки программного обеспечения между собой, оценим достоинства и недостатки каждого из них.

Для изучения этого вопроса рассмотрим пример какого-нибудь простого алгоритма и реализуем его двумя указанными выше способами. Пусть, например, надо реализовать оператор присваивания $ax := \max(X, Y)$, где X и Y – знаковые целые значения размером в слово. Сначала реализуем этот оператор в виде функции со стандартными соглашениями о связях, например, так:

```
Max   proc   near
      push   bp
      mov    bp, sp
      mov    ax, [bp+6]
      cmp    ax, [bp+4]
      jge    L
      mov    ax, [bp+4]
L:    pop    bp
      ret    4
Max   endp
```

Тело нашей функции состоит из 8 команд, а каждый вызов этой функции занимает не менее 3-х команд, например:

```
      ; ax:=Max(A,B)           ; ax:=Max(Z,-13)
      push   A                 push   Z
      push   B                 mov    ax, -13
      call   Max              push   ax
                               call   Max
```

Реализуем теперь нашу функцию в виде макроопределения, например, так (не будем принимать во внимание, что это макроопределение будет неправильно работать для вызовов вида `Max Z, ax`):

```
Max   macro X, Y
      local L
      mov   ax, X
      cmp   ax, Y
      jge   L
      mov   ax, Y
L:
      endm
```

Как видим, каждый вызов нашего макроопределения будет порождать макрорасширение в четыре команды, а каждый вызов процедуры занимает 3-4 команды, да ещё сама процедура имеет длину 8 команд. Таким образом, для коротких алгоритмов выгоднее реализовывать их в виде макроопределений.¹ Всё, конечно, меняется, если длина макрорасширения будет хотя бы 10 команд. В этом случае, если, например, в нашей программе содержится 20 макрокоманд, то в сумме во всех макрорасширениях будет $20 \cdot 10 = 200$ команд. В случае же реализации алгоритма в виде процедуры (пусть её длина тоже 10 команд) и 20-ти вызовов этой процедуры нам потребуется всего $20 \cdot 4 + 10 = 90$ команд. Получается, что для достаточно сложных алгоритмов реализация в виде процедуры более выгодна, что легко понять, если учесть, что процедура присутствует в памяти

¹ Тот факт, что короткие алгоритмы иногда выгоднее реализовывать не в виде процедур и функций, а в виде макроопределений, нашёл отражение и при разработке языков высокого уровня. Так, в некоторых языках высокого уровня, существуют так называемые встраиваемые (**inline**) процедуры и функции, вызов которых во многом производится по тем же правилам, что и вызов макроопределений.

только в одном экземпляре, а каждая макрокоманда требует своего экземпляра макрорасширения, которое будет располагаться в программе на месте этой макрокоманды.

С другой стороны, однако, макроопределения предоставляют программисту такие уникальные возможности, как настройка алгоритма на типы передаваемых параметров, лёгкую реализацию переменного числа параметров, хорошую выдачу диагностик об ошибочных параметрах. Такие возможности весьма трудно и неэффективно реализовываются с помощью процедур.

Исходя из вышеизложенного, наиболее перспективным является *гибридный* метод: реализовать алгоритм в виде макроопределения, в котором производится настройка на типы параметров и выдачу диагностики, а потом, если нужно, вызывается *процедура* для реализации основной части алгоритма. Именно так устроены достаточно сложные макроопределения **inint** и **outint**, которыми Вы часто пользуетесь.

На этом мы закончим наше по необходимости краткое изучение макросредств языка Ассемблера, ещё раз напомним, что необходимо тщательно изучить эту тему в учебнике по языку Ассемблера.

13. Схема работы транслятора с языка Ассемблера.

Сейчас мы рассмотрим, как транслятор преобразует входной модуль на "чистом" языке Ассемблера (уже *без* макросредств) в объектный модуль. Разумеется, мы изучим только общую схему этого, достаточно сложного процесса. Наша цель – рассмотреть основные принципы работы транслятора с языка Ассемблера и ввести необходимую терминологию. Более подробно этот вопрос, который относится к большой теме "Формальные грамматики и методы компиляции", Вы будете изучать в другом курсе.

Итак, как мы знаем, Ассемблер относится к классу системных программ, которые называются компиляторами¹ – переводчиками с одного алгоритмического языка на другой. Наш транслятор переводит модуль с языка Ассемблера на объектный язык. При трансляции выполняются следующие шаги.

- Анализ входного модуля на наличие ошибок.
- Выдача протокола работы транслятора – так называемого листинга, а также выдачу аварийных сообщений об ошибках.² Выдачу листинга при желании, как правило, можно заблокировать.
- Генерация объектного модуля.

Разберём более подробно первый этап – анализ программы на наличие ошибок. Ясно, что найти ошибку можно, только *просмотрев* весь текст модуля, строка за строкой. Каждый просмотр текста программного модуля транслятором называется *проходом*. Наш транслятор с Ассемблера просматривает программу дважды, т.е. совершает два прохода. Такие трансляторы называются двухпроходными. Двухпроходная схема трансляции наиболее простая, можно, однако, усложнив алгоритм, производить трансляцию и за один проход (например, так работает транслятор с языка Турбо-Паскаль).

На первом проходе транслятор с Ассемблера, анализируя текст программы, находит многие (но не все) ошибки, и строит некоторые важные *таблицы*, данные из которых используются как на первом, так и на втором проходе. Вкратце алгоритм первого прохода состоит в следующем. Так как основной синтаксической единицей нашего языка является *предложение* Ассемблера, то рассмотрим, как происходит обработка предложений на первом проходе.

Сначала работает специальная программа, которая называется *лексическим анализатором*. Она разбивает каждое предложение программы на лексемы – логически неделимые единицы языка. О

¹ Термин **транслятор** обозначает программу-переводчика с одного языка на другой. Трансляторы делятся на компиляторы, которые переводят программу целиком, и интерпретаторы, которые выполняют *построчный* перевод и исполнение каждой переведённой строки программы. Хорошим аналогом является письменный (всего текста) и устный (синхронный, по одному предложению) перевод с иностранного языка.

² Как вы узнаете из курса "Системное программное обеспечение" выдача листинга и аварийных диагностик производится на разные устройства – так называемое стандартное устройство вывода (**stdout**) и стандартное устройство для выдачи сообщений об ошибках (**stderr**).

лексемах мы уже должны немного знать из языка Паскаль. Как и в Паскале, в языке Ассемблера существуют следующие классы лексем.

- **Имена.** Как и в Паскале, это ограниченные последовательности букв и цифр, начинающиеся с буквы, при этом большие и маленькие буквы не различаются. Как мы помним, в Паскале к буквам относился также и символ подчёркивания, а в Ассемблере к буквам относятся и такие символы, как вопросительный знак, точка (правда, только в первой позиции и у служебных имён) и некоторые другие. Все имена Ассемблера делятся на *служебные* и имена пользователя.¹
- **Числа.** Язык Ассемблера, как и Турбо-Паскаль, допускает запись чисел в различных системах счисления (десятичной, двоичной, шестнадцатеричной и некоторых других). Не надо забывать и о вещественных числах.
- **Строки символов.** Строки символов в Ассемблере ограничены либо апострофами, либо двойными кавычками. Ещё раз напомним, что в нашем упрощённом изложении алгоритма работы компилятора мы считаем, что макропроцессор, который рассматривал фактические параметры макрокоманд тоже как строки символов, ограниченных запятыми, пробелами или точной с запятой, уже закончил свою работу.
- **Разделители.** Как и в Паскале, лексемы перечисленных выше классов не могут располагаться в тексте программы подряд, между ними обязательно должна находиться хотя бы одна лексема-разделитель. К разделителям относятся знаки арифметических операций, почти все знаки препинания, пробел и некоторые другие символы.
- **Комментарии.** Эти лексемы не влияют на выполнения алгоритма, заложенного в программу, они переносятся в листинг, а из анализируемого текста удаляются. Исключением являются макрокомментарии, которые начинаются с двух символов '; ;', как мы знаем, такие комментарии не переносятся в листинг.

В качестве примера ниже показано предложение Ассемблера, каждая лексема в нём выделена в прямоугольник:

```
Metka : mov ax, Mas + 67 [ bx ] ; Комментарий
```

На этапе лексического анализа выявляются *лексические ошибки*, когда некоторая группа символов не может быть отнесена ни к одному из классов лексем. Примеры лексических ошибок:²

```
134X      'A'38      B"C
```

Все опознанные (правильные) лексемы записываются в специальную *таблицу лексем*. Это делается в основном для того, чтобы на втором проходе уже двигаться по программе *по лексемам*, а не по отдельным составляющим их символам, что позволяет существенно ускорить просмотр программного модуля.

После разбиения предложения Ассемблера на лексемы начинается второй этап обработки предложения – этап *синтаксического анализа*. Этот этап выполняет программа, которая называется *синтаксическим анализатором*. Алгоритмы синтаксического анализа весьма сложны, и здесь мы не будем их рассматривать, это, как уже упоминалось, тема отдельного курса. Если говорить совсем коротко, то синтаксический анализатор пытается из лексем построить более сложные конструкции предложения: поля метки, кода операции и операндов. Особое внимание на этом этапе уделяется в программе именам пользователя, они заносятся синтаксическим анализатором в специальную *таблицу имён*. Вместе с каждым именем в эту таблицу заносятся и *атрибуты* (свойства) имени. Всего в Ассемблере у имени пользователя различают четыре атрибута, ниже перечислены эти атрибуты (не у каждого имени есть все из них).

- Атрибут сегмента *Segment*. Этот атрибут имеет формат *i16*, он задаёт адрес начала того сегмента, в котором описано имя, делённый на 16.
- Атрибут смещения *Offset*, он также имеет формат *i16* и задаёт смещение имени от начала того сегмента, в котором оно описано. Атрибуты *Segment* и *Offset* могут иметь только имена, определяющие области памяти и метки команд.

¹ В отличие от Паскаля, в Ассемблере нет *стандартных* имён (напомним, что стандартное имя имело заранее определённый смысл, но эти имена можно было *переопределить*).

² Здесь надо сказать, что компилятор не предполагает глубокого знания программистом теории компиляции, поэтому в листинге такие ошибки называются общим термином "синтаксические ошибки", хотя, как мы узнаем немного позже, "настоящие" синтаксические ошибки определяются другой частью компилятора – синтаксическим анализатором.

- Атрибут типа `Type`. С этим атрибутом имени мы уже знакомы, для переменных он равен длине переменной в байтах, а для меток равен `-1` и `-2` для близкой и дальней метки соответственно. Все остальные имена имеют тип ноль (в некоторых учебниках по Ассемблеру это трактуется как *отсутствие* типа, при этом говорится, что у таких имён атрибута `Type` **нет**).
- Атрибут значения `Value`. Этот атрибут определён только для имён сегментов, а также для констант и переменных периода генерации.

Приведём примеры описаний имён, которые имеют *первые три* из перечисленных выше атрибутов:

```
A    db    13; Не имеет атрибута Value !
B    equ   A
C:   mov   B, 1
D    equ   C
```

А теперь примеры имён, у которых есть только атрибут `Value` (и атрибут `Type = 0`):

```
N    equ   100
M    equ   N+1
      K=0
P    equ   K
Data segment
```

Рассмотрим теперь пример маленького, синтаксически правильного, но, вообще говоря, бессмысленного программного модуля, для которого построим таблицу имён пользователя:

```
Data segment
A    dw    19
B    db    ?
Data ends
Code segment
      extrn X:far
      mov   ax,Data
      mov   cx,R
      jmp   L
R    equ   -14
      call  X
L:   ret
Code ends
end
```

На рис. 13.1 показана таблица имён, которая построится синтаксическим анализатором при просмотре программы до предложения

```
mov   ax,Data
```

включительно. Атрибуты, которые не могут быть у данного имени, отмечены прочерком.

Как мы уже знаем, значения некоторых имён неизвестны на этапе компиляции, эти значения мы поставили в нашей таблице как `i16=?`. Для каждого заносимого в таблицу имени, кроме атрибутов, ещё запоминается и место в предложении, на котором встретилось имя. В Ассемблере имя пользователя может располагаться в поле метки (тогда это *определение* имени) и в поле операндов (тогда это *использование* имени).

Имя	Segment	Offset	Type	Value
Data	---	---	0	i16=?
A	Data	0	2	---
B	Data	2	1	---
Code	---	---	0	i16=?
X	i16=?	i16=?	-2	---

Рис. 13.1. Вид таблица имён модуля после анализа предложения `mov ax,Data`.

Итак, теперь синтаксический анализатор рассматривает предложение

```
mov cx, R
```

и заносит в таблицу имён имя R. Про это имя ещё ничего неизвестно, поэтому и все поля атрибутов в таблице имеют неопределённые значения:

R	?	?	?	?
---	---	---	---	---

Соответственно, невозможно определить и точный код операции команды `mov cx, R`, так как второй операнд этой команды может иметь формат `r16, m16` или `i16`. Здесь ярко видна необходимость второго просмотра программы: только на втором просмотре, после получения информации об атрибутах имени R, возможно определить правильный формат этой команды (а значит, определить и *длину* этой команды в байтах).

После полного просмотра программы синтаксический анализатор построит таблицу имён, показанную на рис. 13.2.

Имя	Segment	Offset	Type	Value
Data	—	—	0	i16=?
A	Data	0	2	—
B	Data	2	1	—
Code	—	—	0	i16=?
X	i16=?	i16=?	-2	—
R	—	—	0	-14
L	Code	19	-1	—

Рис. 13.2. Вид таблицы имён после анализа всей программы.

На этапе синтаксического анализа выявляются все синтаксические ошибки в программе, например:

```
mov ax, b1; Несоответствие типов
add A, A+2; Несуществующий формат команды
sub cx; Мало операндов
```

и т.д. Используя таблицу имён, синтаксический анализатор легко обнаруживает ошибки следующего вида:

1. Неописанное имя. Имя встречается только в поле операндов и не встречается в поле метки.
2. Дважды описанное имя. Одно и то же имя дважды (или большее число раз) встретилось в поле метки.¹
3. Описанное, но не используемое имя. Это *предупредительное* сообщение может выдаваться некоторыми "продвинутыми" Ассемблерами, если имя определено в поле метки, но ни разу не встретилось в поле операндов (видимо, программист что-то упустил).

Кроме того, синтаксический анализатор может обнаружить и ошибки, относящиеся к модулю в целом, например, превышение максимальной длины некоторого сегмента.

Итак, на втором проходе синтаксический анализатор может обнаружить все ошибки и однозначно определить формат каждой команды. Теперь можно приступить к последнему этапу работы транслятора – генерации объектного модуля. На этом этапе все числа преобразуются во внутреннее машинное представление, выписывается битовое представление всех команд, оформляется паспорт объектного модуля. Далее полученный объектный модуль записывается в библиотеку объектных модулей (обычно куда-нибудь в дисковую память).

На этом мы завершим наше краткое знакомство со схемой трансляции исходного модуля с языка Ассемблера на объектный язык.

¹ Исключением из этого правила, как мы знаем, являются имена сегментов и имена процедур, которые должны встретиться в поле метки два раза.

14. Понятие о мультипрограммном режиме работы.

Архитектура машин фон Неймана предполагает, что последовательно выполняются не только команды текущей программы, но и также и сами программы. Другими словами, пока одна программа полностью не заканчивалась, следующая программа не загружалась в память и не начинала выполняться. Такой режим счёта программ называется *пакетным* режимом работы ЭВМ. Разумеется, такое название этому режиму было дано только после того, как появились и другие режимы работы ЭВМ.

Сейчас мы познакомимся с весьма сложным понятием – мультипрограммным (иногда говорят, многопрограммным) режимом работы ЭВМ. Мультипрограммный режим работы означает, что в оперативной памяти компьютера одновременно находится несколько независимых друг от друга и готовых к счёту программ.¹ Особо следует подчеркнуть, что это могут быть и программы *разных* пользователей.

Мультипрограммный режим работы появился только на ЭВМ, начиная с 3-го поколения, на первых компьютерах его не было [3]. Сейчас нам сначала предстоит разобраться, для чего вообще нужно, чтобы в памяти одновременно находилось несколько программ. Этот вопрос вполне естественный, так как у подавляющего большинства компьютеров только *один* центральный процессор, так что одновременно может считаться только *одна* программа.

Частично мы уже обосновали необходимость присутствия в оперативной памяти нескольких программ, когда изучали систему прерываний. Как правило, при возникновении прерывания происходит автоматическое переключение на некоторую *другую* программы, которая тоже, конечно, должна находиться в оперативной памяти. Здесь, однако, можно возразить, что все программы, на которые производится автоматическое переключение при прерывании, являются *системными* программами (входят в операционную систему),² а при определении мультипрограммного режима работы мы особо подчёркивали, что в оперативной памяти могут одновременно находиться несколько разных программ обычных *пользователей*.

Следует указать две основные причины, по которым может понадобиться мультипрограммный режим работы. Во-первых, может потребоваться *одновременно* выполнять несколько программ. Например, это могут быть программы, которые в диалоговом режиме работают с разными пользователями (программисты Вася и Петя одновременно с разных терминалов отлаживают свои программы, см. рис. 14.1).

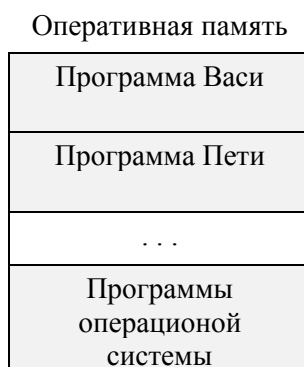


Рис. 14.1. Одновременное нахождение в памяти нескольких программ пользователей.

Правда, здесь возникает следующая трудность: так как центральный процессор на компьютере один, то в каждый момент времени может выполняться или программа Васи, или программа Пети (ну, или служебная программа операционной системы). Эта трудность преодолевается введением

¹ Эти программы могут присутствовать в оперативной памяти не целиком. Во-первых, они могут использовать схему динамической загрузки, и, во-вторых, работать на так называемой виртуальной памяти, при этом некоторые части программы могут временно отсутствовать в оперативной памяти.

² На ЭВМ первых поколений пользователям разрешалось писать свои собственные процедуры-обработчики прерываний, однако в операционных системах современных ЭВМ это, как правило, запрещено. Причина такого запрета будет понятна из нашего дальнейшего изложения мультипрограммного режима работы ЭВМ.

специального режима работы ЭВМ – режима *разделения времени*, который является частным случаем мультипрограммного режима. В режиме разделения времени, используя сигналы прерывания от встроенных в компьютер часов (таймера), процедура-обработчик этого прерывания переключает центральный процессор с одной задачи *пользователя* на другую по истечению определённого кванта времени (обычно порядка нескольких единиц или десятков миллисекунд). В этом режиме и у Васи и у Пети создаётся иллюзия, что только его программа всё время считается на компьютере (правда, почему-то медленнее ☺).

Если отвлечься от несколько шутливого примера с Васей и Петей, то можно заметить, что потребность в таком псевдопараллельном счёте нескольких программ на компьютере с одним центральным процессором весьма распространена. Пусть, например, наш компьютер предназначен для управления несколькими различными химическими реакторами на каком-нибудь заводе, или обслуживает запросы сразу многих абонентов в библиотеке и т.д.

Другая причина широкого распространения мультипрограммного режима заключается в следующем. Наряду с главной частью – центральным процессором и оперативной памятью – в компьютере существует и большое количество так называемых периферийных (внешних) устройств, это диски, клавиатура, мышь, печатающие устройства, линии связи и т.д. (см. рис. 14.2). Все эти периферийные устройства работают значительно более медленно, чем центральный процессор и оперативная память. Имеется в виду, что все они значительно медленнее манипулируют данными. Например, за то время, за которое быстрый лазерный принтер напечатает один символ, оперативная память способна выдать центральному процессору около 3 миллионов байт, а сам центральный процессор способен за это время выполнить порядка одного миллиона команд.



Рис. 14.2. Центральная и периферийная части компьютера.

Поэтому очевидно, что в то время, когда по запросу некоторой программы производится обмен данными с медленными внешними устройствами, центральный процессор не сможет выполнять команды этой программы, т.е. будет простаивать. Например, рассмотрим случай, когда в программе Васи выполняются операторы

```
Read(MyFile, X); Y:=X+1;
```

Очевидно, что оператор присваивания $Y := X + 1$ не сможет начать выполняться, пока из файла не будет прочитано значение переменной X . Вот здесь нам и пригодится способность компьютера автоматически переключаться на выполнение других программ, тоже расположенных в оперативной памяти. Пока одна программа выполняет свои команды на центральном процессоре, другая может выводить свои данные на принтер, третья – читать массив с диска в оперативную память, четвёртая – ждать ввода символа с клавиатуры и т.д. Правда, для того, чтобы обеспечить такую возможность, мало наличия на компьютере одной системы прерываний. Прежде всего, необходимо научить периферийные устройства компьютера работать *параллельно* и относительно *независимо* от центрального процессора.

Как мы говорили, на первых ЭВМ не было режима мультипрограммирования. Сейчас мы сформулируем необходимые требования, которые предъявляются к аппаратуре компьютера, чтобы на этом компьютере было возможно реализовать мультипрограммный режим работы. Сначала заметим, что требование параллельной работы центрального процессора и периферийных устройств, не являются *необходимым* для режима разделения времени, который, как мы уже говорили, является частным случаем мультипрограммного режима работы. Поэтому мы не будем включать это требование в перечень *обязательных* свойств аппаратуры ЭВМ для обеспечения работы в мультипрограммном режиме. Скажем, однако, что параллельная работа периферийных устройств и центрального процессора реализована на большинстве современных ЭВМ и на *всех* больших и супер-ЭВМ.

14.1. Требования к аппаратуре для обеспечения возможности работы в мультипрограммном режиме.

Итак, сформулируем необходимые требования к аппаратуре ЭВМ для обеспечения возможности мультипрограммной работы.

14.1.1. Система прерываний.

Система прерываний необходима как для режима разделения времени, так и для обеспечения параллельной работы центрального процессора и периферийных устройств, так как обеспечивает саму возможность реакции на события и автоматического переключения с одной программы на другую.

14.1.2. Механизм защиты памяти.

Этот механизм обеспечивает безопасность одновременного нахождения в оперативной памяти нескольких независимых программ. Защита памяти гарантирует, что одна программа не сможет случайно или же преднамеренно обратиться в память другой программы (по записи или даже по чтению данных). Очевидно, что без такого механизма мультипрограммный режим просто невозможен.¹

Механизм защиты памяти на современных ЭВМ устроен весьма сложно и часто связан с механизмом так называемой виртуальной памяти, с которым Вы познакомитесь в следующем семестре. Сейчас мы рассмотрим одну из простейших реализаций механизма защиты памяти, так эта защита была сделана на некоторых первых ЭВМ 3-го поколения, способных работать в мультипрограммном режиме.

В центральный процессор добавляются два новых *регистра защиты памяти*, обозначим их $A_{нач}$ и $A_{кон}$. На каждый из этих регистров можно загрузить любой адрес оперативной памяти. Предположим теперь, что после загрузки программы в оперативную память она занимает сплошной участок памяти с адресами от 200000_{10} до 500000_{10} включительно. Тогда загрузчик, перед передачей управления на первую команду программы (у нас это часто была команда с меткой *Start*), присваивал регистрам защиты памяти соответственно значения

$$A_{нач} := 200000_{10} \text{ и } A_{кон} := 500000_{10}$$

Далее, в центральный процессор добавлена способность, перед *каждым* обращением в оперативную память по физическому адресу $A_{физ}$ автоматически проверять условие

$$A_{нач} \leq A_{физ} \leq A_{кон}$$

Если условие истинно, т.е. программа обращается в *свою* область памяти, выполняется требуемое обращение к памяти по записи или чтению данных. В противном случае доступ в оперативную память не производится и вырабатывается сигнал прерывания по событию "попытка нарушения защиты памяти".

Описанный механизм защиты памяти очень легко реализовать, однако он обладает существенным недостатком: каждая программа может занимать только *один сплошной* участок в оперативной памяти. В то же время, как мы знаем, архитектура нашего компьютера допускает, чтобы каждый сегмент программы мог быть размещён на любом свободном месте оперативной памяти. В современных ЭВМ реализован специальный механизм виртуальной памяти, который позволяет выделять для программы любые участки адресов памяти, независимо от того, заняты ли эти, как говорят, *логические* адреса другими программами или нет.

14.1.3. Аппарат привилегированных команд.

Сейчас мы рассмотрим ещё одно *необходимое* свойство аппаратуры, без которого невозможно реализовать мультипрограммный режим работы ЭВМ. Это свойство заключается в следующем: все команды, которые может выполнять центральный процессор, разбиваются на два класса. Команды из одного класса называются обычными командами или *командами пользователя*, а команды из другого класса – *привилегированными* или *запрещёнными* командами.

¹ Даже если не принимать во внимание "вредных" программистов, которые специально захотят испортить или незаконно прочесть данные других программ, всегда существуют вероятность таких действий из-за ошибок в программах даже у "добропорядочных" программистов.

Далее, в центральном процессоре располагается специальный одnorазрядный регистр *режима работы*, который может, естественно, принимать только два значения. Значение этого регистра и определяют тот режим, в котором в данный момент работает центральный процессор: обычный режим (или режим пользователя) или привилегированный режим.¹ В привилегированном режиме центральному процессору разрешается выполнять *все* команды языка машины, а в режиме пользователя – только обычные (не привилегированные) команды. При попытке выполнить привилегированную команду в пользовательском режиме вырабатывается сигнал прерывания, а сама команда, естественно, не выполняется. Из этого правила выполнения команд легко понять и другое название для привилегированных команд – *запрещённые* команды, так как их выполнение запрещено в режиме пользователя. Объясним теперь, почему без аппарата привилегированных команд невозможно реализовать мультипрограммный режим работы ЭВМ.

Легко понять, что, например, команды, которые заносят на регистры защиты памяти $A_{нач}$ и $A_{кон}$ новые значения, должны быть привилегированными. Действительно, если бы это было не так, то любая программа могла бы занести на эти регистры адреса начала и конца оперативной памяти, после чего получила бы возможность записывать данные в любые области памяти.

Привилегированными должны быть и все команды, которые обращаются к внешним (периферийным) устройствам. Например, нельзя разрешать запись на диск в режиме пользователя, так как диск – это тоже общая память для всех программ, только внешняя, и одна программа может испортить на диске данные, принадлежащие другим программам. То же самое относится и к печатающему устройству: если разрешить всем программам бесконтрольно выводить свои данные на печать, то, конечно, разобраться в том, что же получится на бумаге, будет невозможно.

Итак, в мультипрограммном режиме программе пользователя запрещается выполнять команды, работающие с внешними устройствами (дисками, принтерами, линиями связи и т.д.). Как же быть, если программе необходимо, например, считать данные из *своего* файла на диске в оперативную память? Выход один – программа пользователя должна обратиться к служебной процедуре с просьбой, выполнить для неё ту работу, которую сама программа пользователя сделать не в состоянии. Служебная программа, естественно, должна работать в привилегированном режиме. Перед выполнением запроса служебная процедура проверяет, имеет ли программы пользователя право на запрашиваемое действие, например, что эта программа имеет необходимые *полномочия* на чтение из указанного файла.²

Переключение из привилегированного режима в режим пользователя обычно производится по специальной (не привилегированной) машинной команде. Значительно сложнее обстоит дело с такой опасной операцией, как переключение центрального процессора в привилегированный режим работы. Это переключение невозможно выполнить по какой-либо машинной команде (поймите, почему это так!). Обычно переключение в привилегированный режим производится автоматически при обработке центральным процессором сигнала прерывания, а иногда – при вызове специальных системных процедур, которые имеют *полномочия* для работы в привилегированном режиме.

14.1.4. Таймер.

Встроенные в компьютер электронные часы (таймер) появились ещё до возникновения мультипрограммного режима работы. Тем не менее, легко понять, что без таймера мультипрограммный режим невозможен. Действительно, это единственное внешнее устройство, которое гарантированно и *периодически* посылает центральному процессору сигналы прерываний. Без таких сигналов некоторые программы могли бы войти в выполнение бесконечного цикла (как говорят программисты – заикнуться), и ничто не могло бы вывести компьютер из этого состояния.

Итак, мы рассмотрели *аппаратные* средства, *необходимые* для обеспечения мультипрограммного режима работы ЭВМ. Остальные аппаратные возможности ЭВМ, которые часто называют студенты при ответе на этот вопрос (такие, как большая оперативная память, высокое быстродействие центрального процессора и другие) являются, конечно, желательными, но *не необходимыми*.

Разумеется, кроме перечисленных аппаратных средств, для обеспечения мультипрограммной работы совершенно необходимы и специальные *программные* средства, прежде всего операционная

¹ В архитектуре нашего компьютера регистр режима работы содержит два разряда и может принимать значения 0, 1, 2 и 3. Практически всегда, однако, используются только два из этих четырёх значений (0 и 3).

² Способы задания таких полномочий Вы будете изучать в следующем семестре на примере операционной системы Unix.

система, поддерживающая режим мультипрограммной работы. Такая операционная система является примерно на порядок более сложной, чем её предшественницы – операционные системы, не поддерживающие мультипрограммный режим работы. Всё это, однако, тема Вашего следующего семестра, а мы продолжаем изучать архитектуру современных ЭВМ.

15. Архитектурные особенности современных ЭВМ.

Исследуем сейчас следующий вопрос: оценим скорость работы различных устройств ЭВМ. Оперативная память современных ЭВМ способна читать и записывать данные примерно каждые 10 наносекунд (нс), $1 \text{ нс} = 10^{-9} \text{ сек.}$, а центральный процессор может выполнить команду примерно за 1–2 нс. После некоторого размышления становится понятным, что "что-то здесь не так".

Действительно, рассмотрим, например, команду `add ax, X`. Для выполнения этой команды центральный процессор должен сначала считать из оперативной памяти саму команду (это 4 байта), затем операнд X (это ещё 2 байта), потом произвести операцию сложения. Таким образом, центральный процессор потратит на выполнение этой команды $6 \cdot 10 + 2 = 62$ нс. Спрашивается, зачем делать центральный процессор таким быстрым, если всё равно 97% своего времени он будет ждать, пока команды и данные не будут считаны из оперативной памяти на регистры? Налицо явное несоответствие в скорости работы оперативной памяти и центрального процессора ЭВМ.

Данная проблема на современных ЭВМ решается несколькими способами, которые мы сейчас рассмотрим. Сначала оперативную память стали делать таким образом, чтобы за одно обращение к ней она выдавала не по одному байту, а по несколько байт сразу. Для этого оперативную память разбивают на блоки (обычно называемые банками памяти), которые могут работать параллельно. Этот приём называют *расслоением памяти*. Например, если память разбита на 8 блоков, то за одно обращение к ней можно сразу считать 8 байт, при этом байты с последовательными адресами располагается в разных блоках. Таким образом, за одно обращение к памяти можно считать несколько команд или данных.

Скорость работы оперативной памяти современных ЭВМ так велика, что требуется какое-то образное сравнение, чтобы это почувствовать. Легко подсчитать, что за одну секунду из памяти можно прочесть $8 \cdot 10^8$ байт. Если считать каждый байт символом текста и учесть, что на стандартной странице книги помещается примерно 2000 символов, то получается, что за 1 секунду центральный процессор можно прочесть целую библиотеку из 80 томов по 500 страниц в каждом томе.

Легко, однако, вычислить, что, несмотря на такую огромную скорость, оперативная память продолжает тормозить работу центрального процессора. Проведя заново расчёт времени выполнения команды `add ax, X` мы получим:

$$10 \text{ нс (чтение команды)} + 10 \text{ нс (чтение числа)} + 2 \text{ нс (выполнение команды)} = 22 \text{ нс.}$$

Как видим, хотя ситуация и несколько улучшилась, однако всё ещё примерно 90% своего времени центральный процессор вынужден ждать, пока из оперативной памяти поступят нужные команды и данные. Для того, чтобы исправить эту неприятную ситуацию, в архитектуру компьютера встраивается специальная память, которую называют *памятью типа кэш*, или просто кэшем.

Кэш работает так же быстро, как и центральный процессор, т.е. может, например, выдавать по 8 байт каждые 1–2 нс. Для программиста кэш является невидимой памятью в том смысле, что эта память не адресуемая, к ней нельзя обратиться из программы по какой-либо команде чтения или записи.¹ Центральный процессор работает с кэшем по следующей схеме.

Когда центральному процессору нужна какая-то команда или данное, то сначала он смотрит, не находится ли эта команда или данные в кэше, и, если они там есть, читает их оттуда, *не обращаясь* к оперативной памяти. Разумеется, если требуемой команды или данных в кэше нет, то центральный процессор вынужден читать их из относительно медленной оперативной памяти, однако копию прочитанного он обязательно оставляет в кэше. Аналогично, при записи данных центральный процессор помещает их в кэш. Особая ситуация складывается, если требуется что-то записать в кэш, а там нет свободного места. В этом случае по специальным алгоритмам, которые Вы будете изучать в следующем семестре, из кэша удаляются некоторые данные, обычно те, к которым дольше всего не было обращения. Таким образом, в кэше накапливаются, в частности, наиболее часто используемые

¹ Конечно, существуют *привилегированные* команды для работы с кэшем как с единым целым, это, например, команда очистки кэша от всех команд и данных.

команды и данные, например, все команды не очень длинных циклов после их первого выполнения будут находиться в памяти типа кэш.¹

Память типа кэш строится из очень быстрых и, следовательно, дорогих интегральных схем, поэтому её объём сравнительно невелик, примерно 5% от объёма оперативной памяти. Однако, несмотря на свой относительно малый объём, кэш вызывает значительное увеличение скорости работы ЭВМ, так как по статистике примерно 90-95% всех обращений за командами и данными производится в память типа кэш. Теперь наша команда `add ax, X` будет выполняться за $2+2+2=6$ нс.² Как видим, ситуация коренным образом улучшилась, хотя всё равно получается, что центральный процессор работает только 30% от времени выполнения команды, а остальное время ожидает поступления команд и данных. Для того, чтобы исправить эту ситуацию, нам придётся снова существенно изменить архитектуру центрального процессора.

15.1. Конвейерные ЭВМ.

Как мы уже говорили, современные ЭВМ могут одновременно выполнять несколько команд, для этого они должны иметь несколько центральных процессоров, либо центральный процессор такого компьютера строится по так называемой *конвейерной* (pipeline) архитектуре. Рассмотрим схему работы таких конвейерных ЭВМ.

Выполнение каждой команды любым центральным процессором можно разбить на несколько шагов. Можно выделить следующие основные шаги выполнения команды.

- Выбор команды из оперативной памяти (или кэша) на регистр команд.
- Определение кода операции (так называемое декодирование команды).
- Вычисление исполнительных адресов операндов.
- Выбор операндов из оперативной памяти (или кэша) на регистры арифметико-логического устройства.
- Выполнение требуемой операции (сложение, умножение, сдвиг и т.д.) над операндами на регистрах арифметико-логического устройства.
- Запись результата операции и выработка флагов.

В конвейерных ЭВМ центральный процессор состоит из нескольких блоков, каждый из которых выполняет один из перечисленных выше шагов команды. Теперь понятно, что эти блоки можно заставить работать параллельно, обеспечивая, таким образом, одновременное выполнение центральным процессором нескольких последовательных команд программы. На рис. 15.1 приведена схема работы центрального процессора конвейерной ЭВМ, направление движения команд на конвейере показано толстой стрелкой. Одновременно на нашем конвейере находится шесть команд.

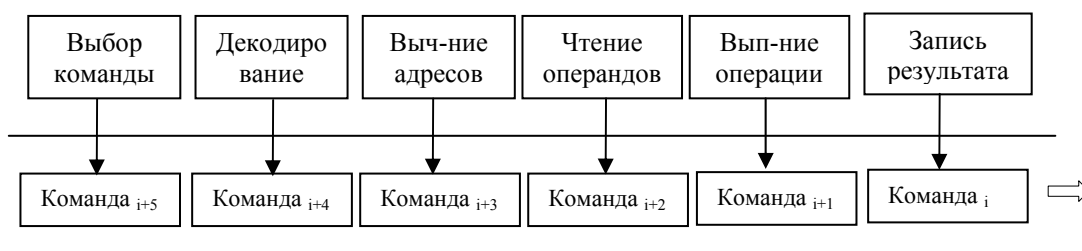


Рис. 15.1. Схема работы конвейера.

Из показанной схемы понятно, почему такие ЭВМ называются конвейерными. Как, например, на конвейере автомобильного завода одновременно находятся несколько машин в разной стадии сборки, так и на конвейере центрального процессора находятся несколько команд в разной стадии выполнения. Отметим хорошее свойство любого конвейера: хотя выполнение каждой команды, как в нашем примере, занимает шесть шагов, однако *на каждом шаге* с конвейера "сходит" полностью выполненная команда. Таким образом, использование такого рода конвейера позволяет, в принципе, в шесть раз повысить скорость выполнения программы.

¹ Как видно из этого алгоритма работы кэша, он весьма "болезненно" реагирует на прерывания, так как при этом производится переключение на другую программу, и данные в кэше необходимо полностью сменить.

² За одно обращение из памяти теперь читается не одна команда, а в среднем две последовательные команды, поэтому можно считать, что среднее время выполнения команды будет ещё меньше: $1+2+2=5$ нс.

Вот теперь мы достигли *соответствия* скорости работы центрального процессора и памяти. Действительно, предположим для простоты, что каждое из шести устройств на конвейере выполняет свой этап обработки команды за 1 нс, тогда каждая команда выполняется за 6 нс и за это время она успевает произвести все необходимые обмены командами и данными с памятью. В то же время, как мы уже отмечали, скорость выполнения потока команд центральным процессором получается в 6 раз больше за счёт работы конвейера.

Разумеется, не всё обстоит так хорошо, как кажется с первого взгляда. Первая неприятность поджидает нас, если одна из следующих команд использует результат работы предыдущей команды, а это случается очень часто по самой сути вычислительных алгоритмов. Например, пусть есть фрагмент программы:

```
add  al, [bx]
sub  X, al
inc  bx
inc  di
```

Для второй команды этого фрагмента нельзя выполнять операцию вычитания, пока первая команда фрагмента не запишет в `al` свой результат, т.е. не сойдёт с конвейера. Таким образом, вторая команда будет *задержана* на третьей позиции конвейера (на четвёртой позиции уже надо читать операнд `al`, а от ещё не готов). Вместе со второй командой из нашего примера остановится и выполнение следующих за ней команд и на конвейере образуются два "пустых места". Ясно, что скорость выполнения всей программы может при этом сильно упасть. Зная такую особенность работы конвейера центрального процессора "умный" компилятор может изменить порядок команд в машинной программе, получив, например, такой эквивалентный фрагмент:¹

```
add  al, [bx]
inc  bx
inc  di
sub  X, al
```

Здесь, как легко увидеть, конвейер уже не будет пустовать. Другая неприятность случается, когда на конвейер поступает команда *условного перехода*. Будет ли после выполнения этой команды производиться переход, или же продолжится последовательное выполнение команд, выяснится только тогда, когда команда условного перехода сойдёт с конвейера. Так спрашивается, из какой же ветви программы выбирать на конвейер следующую команду?

Обычно при конструировании конвейера принимается какое-либо одно из двух решений. Во-первых, можно выбирать команды из наиболее вероятной ветви условного оператора (например, очевидно, что для команды цикла `loop` повторение тела цикла значительно более вероятно, чем выход из цикла). Во-вторых, можно поочерёдно выбирать на конвейер команды из *обеих* ветвей (разумеется, в этом случае половина команд будет выполняться зря и их "недоделанными" придётся выбросить с конвейера).

Далее, как мы уже отмечали ранее, конвейер весьма болезненно реагирует на прерывания, так как при этом производится автоматическое переключение на другую программу и конвейер приходится очищать от частично выполненных команд предыдущей программы.²

На этом мы закончим наше краткое знакомство с архитектурными особенностями современных ЭВМ и перейдём к сравнению между собой ЭВМ разных классов.

15.2. ЭВМ различной архитектуры.

Отметим сейчас важное обстоятельство. До сих пор мы изучали, в основном, архитектуру центральной части компьютера, – центрального процессора и оперативной памяти. Практически не рассматривалась архитектура ЭВМ в целом, способы взаимодействия центральной части компьютера с периферийными устройствами, способы управления устройствами ввода/вывода центральным процессором. Такое "однобокое" изучение архитектуры ЭВМ имело свою причину. Дело в том, что,

¹ Такие компиляторы называются *оптимизирующими* компиляторами, оптимизация на машинном языке выполняется обычно на одном или нескольких дополнительных проходах программы перед получением объектного модуля.

² На супер-ЭВМ для этих целей обработку большинства прерываний обычно поручают одному из каналов ввода/вывода (периферийных процессоров), что позволяет не прерывать работу конвейера *центрального* процессора. Архитектуру ЭВМ с каналами ввода/вывода мы будем изучать далее.

несмотря на большое разнообразие архитектур центральных процессоров современных ЭВМ, различие в этих архитектурах всё же значительно меньше, чем в архитектурах компьютеров в целом.

Сейчас мы рассмотрим две архитектуры ЭВМ, которые в каком-то смысле являются противоположными, находятся на разных полюсах организации связи центральной части машины с её периферийными устройствами. Сначала изучим способ организации связи устройств компьютера, который получил название архитектуры с общей шиной.

15.2.1. Архитектура ЭВМ с общей шиной.

Эта архитектура была разработана, когда появилась необходимость в массовом производстве относительно простых компьютеров (их тогда называли мини- и микро- ЭВМ [11]). Основой архитектуры этого класса ЭВМ была, как можно легко догадаться из названия, *общая шина*. В первом приближении общую шину можно представить себе как набор электрических проводов (линий), снабженных некоторыми электронными схемами. В современных ЭВМ число линий в такой шине обычно порядка сотни. Все устройства компьютера в архитектуре с общей шиной соединяются между собой посредством подключения к такому общему для них набору электрических проводов – шине. На рис. 15.2 показана схема соединения устройств компьютера с помощью общей шины.



Рис. 15.2. Архитектура компьютера с общей шиной.

В этой архитектуре шина исполняет роль главного элемента, связующей магистрали, по которой производится обмен информацией между всеми остальными устройствами ЭВМ. Легко понять, что, так как обмен информацией производится по шине с помощью электрических сигналов, то в каждый момент времени только *два* устройства могут выполнять такой обмен. Обычно одно из этих устройств является ведущим (инициатором обмена данными), а другое – подчинённым (ведомым). Все устройства связаны с общей шиной посредством специальных электронных схем, которые называются *портами* ввода/вывода. Каждый порт имеет на шине уникальный номер (в нашей архитектуре этот номер имеет формат $i16$). Обычно у каждого устройства не один порт, а несколько, так как они специализированные: по некоторым портам устройство может читать данные с шины, по другим – записывать (передавать) данные в шину, а есть и универсальные порты, как для чтения, так и для записи.

При использовании шины устройствами может возникать конфликт, когда два или более устройств захотят одновременно обмениваться данными. Для разрешения таких конфликтов предназначен *арбитр шины* – специальная электронная схема, которая обычно располагается на одном из концов шины. Разрешение конфликтов производится по принципу приоритетов устройств, – устройству с большим приоритетом арбитром отдаётся предпочтение при конфликте. В простейшем случае приоритеты устройствам явно не назначаются, просто считается, что из двух устройств то имеет больший приоритет, которое расположено на шине *ближе* к арбитру. Исходя из этого, более "важные" устройства стараются подключить к шине поближе к арбитру.

Разберём схему обмена данными между двумя устройствами с помощью общей шины. Сначала ведущее устройство (инициатор обмена) делает так называемый *запрос шины*, т.е. посылает арбитру сигнал о желании начать обмен данными (или же читает из специального регистра флаг-признак занятости шины). Если шина занята, то устройство вынуждено ждать её освобождения, а если шина свободна, то устройство производит операцию *захвата шины* в своё монопольное использование.

После захвата шины ведущее устройство определяет, готово ли ведомое устройство для обмена данными. Для этого ведущее устройство посылает ведомому устройству специальный сигнал, или же читает из порта ведомого устройства его *флаг готовности*. Определив готовность ведомого

устройства, ведущее устройство начинает обмен данными. Каждая порция данных (в простейшем случае это один байт или одно слово) снабжается номером порта устройства-получателя.

Окончив обмен данными, ведущее устройство производит *освобождение шины*. На этом операция обмена данными между двумя устройствами по общей шине считается завершённой. Разумеется, арбитр следит, чтобы ни одно из устройств не захватывало шину на длительное время (например, устройство может сломаться, и оно поэтому "забудет" освободить шину).

Рассмотрим теперь, как видит общую шину программист. Как уже было сказано, у каждого периферийного устройства обязательно есть один или несколько портов с закреплёнными за этим устройством номерами. Программист может обмениваться с портами байтами или словами (в зависимости от вида порта). Для записи в некоторый порт используется команда

```
out op1, op2
```

Здесь операнд `op1` определяет номер нужного порта и может иметь формат `i8` (если номер порта небольшой и известен заранее) или быть регистром `dx` (если номер большой или становится известным только в процессе счёта программы). Второй операнд `op2` должен задаваться регистрами `al` (если производится обмен байтом) или `ax` (если производится обмен словом).

Для чтения данных из порта служит команда

```
in op1, op2
```

Здесь уже *второй* операнд `op2` определяет номер нужного порта и может иметь, как и в предыдущей команде, формат `i8` или быть регистром `dx`. Первый операнд `op1` должен задаваться регистрами `al` (если производится обмен байтом) или `ax` (если производится обмен словом). Далее мы рассмотрим небольшой пример использования этих команд.

Рассмотрим теперь общую архитектуру связи центрального процессора и периферийных устройств с точки зрения пользователей разного уровня.

- Конечный пользователь. Пользователь-непрограммист бухгалтер Иванов уверен, что в компьютере есть команда "Распечатать ведомость", так как именно это происходит каждый раз, когда он нажимает на кнопку меню "Печать ведомости".
- Прикладной программист. Программист Петров, который написал бухгалтерскую программу на языке Паскаль, только улыбнётся наивности Иванова. Уж он то точно знает, что даже для того, чтобы вывести только один, например, символ 'A', надо написать оператор стандартной процедуры `Write('A')`. Правда Петрову известно, что на самом деле его программа сначала переводится (транслируется) на машинный язык, поэтому он из любопытства поинтересовался у программиста на Ассемблере Сидорова, что тот напишет, чтобы вывести символ 'A'. Сидоров ответил, что обычно для этой цели он пишет предложение Ассемблера `outch 'A'`. Разница между этими двумя способами вывода символа показалась Петрову несущественной, например он читал о том, что, например, в языке C для этой же цели надо вызвать библиотечную функцию `printf("%c", 'A');`¹.
- Программист на Ассемблере. Сидоров, однако, знает, что предложение `outch 'A'` является не командой машины, а *макрокомандой*, на её место макропроцессор подставит макрорасширение, например, такого вида

```
mov    dl, 'A'
mov    ah, 02h
int    21h
```

Вот этот, как говорят, *системный вызов* и будет, с точки зрения Сидорова, выводить символ 'A' на стандартное устройство вывода.

- Системный программист. Системный программист (раньше иногда говорили *системный аналитик*) Антонов, однако снисходительно пояснит Сидорову, что системный вызов – это просто переход на служебную процедуру-обработчик прерывания с номером `21h`. А уж эта процедура и произведёт на самом деле вывод символа, используя, в частности, специальные команды обмена с внешними устройствами `in` и `out`.

¹ Это яркий пример того, как макросредства *повышают* уровень языка: макрокоманда вывода символа оказываеся для Петрова по внешнему виду (если отвечать от деталей синтаксиса), очень похожа на соответствующие операторы языков *высокого* уровня.

- Инженер-электронщик. Инженер Попов, внимательно прослушав разговор пользователей, скажет, что всё это неверно. *На самом деле* центральный процессор выводит символ на экран или печатающее устройство путём сложной последовательности действий, которая включает в себя такие операции с общей шиной, как запрос, захват, передача данных и освобождение этой шины. И только после этого символ, наконец, прибывает по назначению.

Как Вы догадываетесь, нельзя сказать, кто же из этих людей прав, и бессмысленно спрашивать, как всё происходит "на самом деле". Каждый из них прав со своего **уровня** видения архитектуры компьютера. И, как мы уже говорили, опускаться на более низкий уровень рассмотрения архитектуры следует только тогда, когда это абсолютно необходимо для дела.

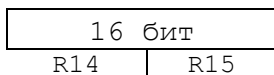
Разберём теперь простой пример реализации операции ввода/вывода на уровне системного программиста. Оставим в стороне пользователя-непрограммиста (он нам сейчас неинтересен) и рассмотрим, например, операцию позиционирования курсора на экране компьютера в позицию (X, Y) .

Для прикладного программиста, как Вы знаете, для этой цели надо выполнить, например, оператор стандартной процедуры Турбо-Паскаля `GotoXY(X, Y)`. Для программиста на Ассемблере позиционирование курсора можно выполнить с использованием такого системного вызова:

```
mov ah, 2
mov bl, 0
mov dl, X
mov dh, Y
int 10h
```

Как видим, параметры позиционирования X и Y передаются в регистрах `dl` и `dh`. Системный вызов `int 10h` может выполнять различные операции с экраном компьютера, в зависимости от своих параметров, передаваемых ему на регистрах. Рассмотрим (в сильно упрощённом виде) тот фрагмент процедуры-обработчика системного вызова, который выполняет запрос на позиционирование курсора.¹

Во-первых, нам необходимо понять, а как вообще дисплей (точнее, электронная схема – контроллер дисплея) "знает", куда необходимо в каждый момент времени поставить курсор. Оказывается, что у контроллера дисплея, как, впрочем, и у любого другого периферийного устройства, есть свои *регистры*. Нас будут интересовать регистры дисплея с номерами 14 и 15 (обозначим их $R14$ и $R15$), каждый из них имеет размер 8 бит, но их совокупность может хранить длинное целое число, как показано ниже



Далее, дисплей "считает",² что его экран имеет не 25 строк и 80 столбцов, как думают программисты, а 25×80 *знакомест*, в каждое из которых можно вывести один символ и поставить курсор. Знакоместа в первой строке экрана нумеруются не от 1 до 80, а от 0 до 79, во второй – от 80 до 159 и т.д. Другими словами, все позиции экрана рассматриваются как одномерный массив. Так вот, чтобы курсор переместился в нужную нам позицию (X, Y) в пару регистров $\langle R14, R15 \rangle$ необходимо записать число

$$80 * (Y-1) + (X-1)$$

Следовательно, сначала процедуре-обработчику прерывания необходимо вычислить это число, используя параметры X и Y из системного вызова:

```
mov al, 80
dec dh; Y-1
mul dh; ax:=80*(Y-1)
dec dl; X-1
add al, dh
adc ah, 0; ax:=80*(Y-1)+(X-1)
mov bx, ax; Спасём на bx
```

Теперь необходимо переслать содержимое регистров `bl` и `bh` соответственно в регистры $R15$ и $R14$ дисплея. Для этого мы будем использовать два порта дисплея (в каждый можно записывать для

¹ Предполагается, что компьютер работает в простейшем (как говорят, *незащищённом*) режиме.

² Мы рассматриваем, естественно, работу дисплея только в стандартном текстовом режиме.

передачи дисплею операнд размером в байт). Порт с шестнадцатеричным номером 3D4h позволяет выбрать номер регистра дисплея, в который будет производиться очередная запись данных. Для этого в этот порт необходимо записать номер соответствующего регистра (у нас это номера 15 и 14). После выбора номера регистра запись в него нового значения производится посредством посылки байта в "транспортный" порт дисплея с номером 3D5h. В итоге получается следующий фрагмент программы:

```

mov dx,3D4h; Порт выбора регистра
mov al,15
out dx,al; Выбираем R15
inc dx; Порт записи в регистр
mov al,b1; младший байт ВХ
out dx,al; Запись в R15
dec dx; Порт выбора регистра
mov al,14
out dx,al; Выбираем R14
inc dx; Порт записи в регистр
mov al,bh; старший байт ВХ
out dx,al; Запись в R14

```

Вот теперь курсор будет установлен в нужное место экрана, и можно возвращаться на команду, следующую за системным вызовом `int 10h`. Разумеется, наш алгоритм весьма примитивен. Например, после записи в 15-й регистр дисплея и до записи в 14-й регистр курсор прыгнет в непредсказуемое место экрана, так что по-хорошему надо было бы на время работы нашего фрагмента *заблокировать* для контроллера чтение данных из регистров дисплея. Это, разумеется, делается записью некоторого значения в определённый *управляющий* регистр дисплея, для чего понадобятся и другие команды `in` и `out`. Кроме того, хорошо бы предварительно убедиться, что дисплей вообще включён и работает в нужном нам режиме, для чего потребуется, например, считать некоторые *флаги состояния* дисплея.

Надеюсь, что этот простенький фрагмент реализации системного вызова не отобьёт у Вас охоту быть системным программистом и заниматься написанием драйверов внешних устройств ☺.

15.2.2. Достоинства и недостатки архитектуры с общей шиной.

Из рассмотренной схемы связи всех устройств компьютера с помощью общей шины легко увидеть как достоинства, так и недостатки этой архитектуры. Несомненным достоинством этой архитектуры является её простота и возможность лёгкого подключения к шине новых устройств. Для подключения нового устройства необходимо оборудовать его соответствующими портами, присвоив им свободные номера, благо этих номеров много – 2^{16} .

Главный недостаток этой архитектуры тоже очевиден: пока два устройства обмениваются данными, остальные должны простаивать. Можно сказать, что компьютер в какие-то периоды времени вынужден соизмерять скорость своей работы со скоростью *самого медленного* устройства на общей шине. Этот недостаток давно осознан конструкторами ЭВМ и с ним пытаются бороться. Например, наряду с главной шиной, соединяющей все устройства, вводят в архитектуру *вспомогательные* шины, соединяющие избранные самые быстрые устройства (например, центральный процессор и оперативную память). Ясно, однако, что невозможно соединить своими шинами всевозможные пары устройств, это просто экономически нецелесообразно, не говоря уже о том, что такую архитектуру практически невозможно реализовать.

Исходя из таких очевидных недостатков архитектуры с общей шиной была разработана и другая архитектура связи устройств компьютера между собой. Обычно она называется архитектурой с каналами ввода/вывода.

15.2.3. Архитектура ЭВМ с каналами ввода/вывода.

Архитектура ЭВМ с каналами ввода/вывода предполагает возможность параллельной работы нескольких устройств. Поймём сначала, какие же работы нам надо производить параллельно. Оказывается, что нужно обеспечить параллельный обмен данными нескольких устройств с оперативной памятью. Действительно, когда мы рассматривали мультипрограммный режим работы

ЭВМ, мы говорили, что для эффективного использования ресурсов необходимо обеспечить как можно более полную загрузку всех устройств компьютера.

Например, одна программа может выполнять свои команды на центральном процессоре, другая – читать массив данных с диска в оперативную память, третья – выводить результаты работы из оперативной памяти на печать и т.д. Как видим, здесь оперативная память параллельно работает с несколькими устройствами: центральным процессором (он читает из памяти команды и данные, а записывает в память результат выполнения некоторых команд), диском, печатающим устройством и т.д. Скорость работы оперативной памяти должна быть достаточна для такого параллельного обслуживания нескольких устройств (здесь, как мы уже говорили, сильно помогает расслоение оперативной памяти и использование кэша).

Как мы знаем, центральный процессор выполняет обращения к оперативной памяти, подчиняясь командам выполняемой программы. Ясно, что и все другие обмены данными с оперативной памятью должны выполняться под управлением достаточно "интеллектуальных" устройств ЭВМ. Вот эти устройства и называются *каналами ввода/вывода*, так как они управляют обменом данными между оперативной памятью и, как говорят, периферией. По-существу, канал ввода/вывода является *специализированным процессором* со своей системой команд (своим машинным языком).

В современной литературе по архитектуре ЭВМ у термина "канал ввода/вывода" есть много синонимов. Часто их называют процессорами ввода/вывода или периферийными процессорами (смысл этих названий легко понять из назначения данных устройств). Наиболее "навороченные" каналы называют иногда *машинами переднего плана* (front-end computer), здесь имеется в виду, что все внешние устройства, а, следовательно, и пользователи, могут общаться с центральной частью компьютера только через эти каналы. Кроме того, машины переднего плана могут разгрузить центральный процессор, взяв на себя, например, обработку прерываний от внешних устройств, весь диалог с пользователями, компиляцию программ в объектный код и т.д. Центральный процессор при этом будет выполнять только свою основную работу – быстрый счёт программ *пользователей*.

Чаще всего на компьютере есть несколько каналов ввода/вывода, так как эти каналы выгоднее делать *специализированными*. Обычно один канал ввода/вывода успевает обслуживать все медленные внешние устройства (клавиатура, печать, дисплеи, линии связи и т.д.), такой канал называется *мультиплексным*. Один или несколько других каналов работают с быстрыми внешними устройствами (обычно это дисковая память), такие каналы называются *селекторными*. В отличие от мультиплексного канала, который успевает, быстро переключаясь с одного медленного внешнего устройства на другое, обслуживать их все как бы одновременно, селекторный канал в каждый момент времени может работать только с одним быстрым внешним устройством. На рис. 15.3 показана схема ЭВМ с каналами ввода/вывода.

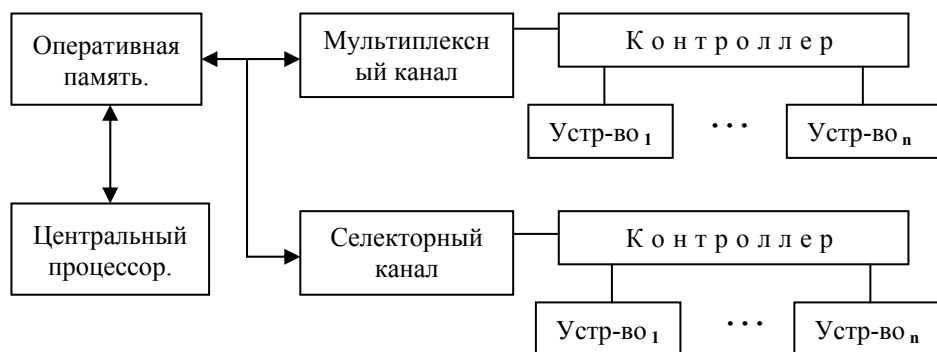


Рис. 15.3. Схема ЭВМ с каналами ввода/вывода.

Как видно из этого рисунка, внешние устройства подключаются к каналам не напрямую, а через специальные электронные схемы, которые называются контроллерами. Это связано с тем, что каналы являются универсальными, они должны допускать подключение внешних устройств, очень разных по своим характеристикам. Таким образом, канал как бы работает с некоторыми *обобщёнными* внешними устройствами, а все особенности связи с конкретными устройствами реализуются в контроллерах. Например, один контроллер предназначен для подключения к нему жёстких дисков, другой – архивных накопителей на магнитной ленте (так называемых стриммеров) и т.д.

Как мы уже говорили, для компьютеров с общей шиной при выполнении системного вызова центральный процессор переключается на процедуру-обработчика прерывания, и эта процедура

выполняет программу, реализующую требуемое действие, например, чтение массива с диска в оперативную память. Другими словами, во время выполнения процедуры-обработчика прерывания программа пользователя, естественно, не считается.

Совершенно по-другому производится обработка системного вызова на компьютере с каналами ввода/вывода. После того, как программа пользователя произведёт системный вызов, вызванная процедура-обработчик прерывания посылает соответствующему каналу приказ начать выполнение программы канала, реализующей требуемое действие, после чего производится немедленный возврат на выполнение программы пользователя. Далее начинается *параллельная* работа центрального процессора по выполнению программы пользователя и канала, выполняющего свою собственную программу по обмену с внешними устройствами, например, по чтению массива с диска в оперативную память.¹

Разумеется, параллельная работа нескольких устройств может привести к весьма неприятным последствиям. Например, если некоторая программа начнёт присваивать новые значения элементам массива, который в это время канал записывает на диск, то, как легко представить, ничего хорошего не получится. Для предотвращения таких ситуаций существуют особые аппаратные и программные средства, позволяющими, как говорят, *синхронизовать* параллельную работу нескольких устройств. С этими средствами Вы познакомитесь в следующем семестре.

15.3. Уровни параллелизма.

Как мы знаем, первые компьютеры удовлетворяли почти всем принципам Фон Неймана. В этих компьютерах поток последовательно выполняемых в центральном процессоре команд обрабатывал поток данных. ЭВМ такой простой архитектуры носят в литературе сокращённое название ОКОД (Один поток Команд обрабатывает Один поток Данных – английское сокращение SISD: Single Instruction Single Data). В настоящее время компьютеры, однако, нарушают почти все принципы Фон Неймана. Дело в том, что вычислительная мощность современных компьютеров базируется как на скорости работы всех узлов ЭВМ, так и, в значительной степени, на *параллельной обработке данных*. В заключение нашего краткого изучения архитектуры современных ЭВМ рассмотрим классификацию способов параллельной обработки данных на компьютере.

- Параллельное выполнения программ может производиться на одном компьютере, если он имеет несколько центральных процессоров. Как правило, в этом случае компьютер имеет и несколько периферийных процессоров (каналов). Существуют ЭВМ, у которых могут быть от нескольких десятков до нескольких сотен и даже тысяч центральных процессоров. В таких компьютерах много потоков команд одновременно обрабатывают много потоков данных, в научной литературе это обозначается сокращением МКМД (или по-английски MIMD).
- Параллельные процессы в рамках одной программы. Программа пользователя может породить несколько параллельных процессов обработки данных, каждый такой процесс для операционной системы является самостоятельной единицей работы. Процессы одной задачи могут псевдопараллельно выполняться в мультипрограммном режиме точно так же, как и задачи независимых пользователей.

В качестве примера рассмотрим случай, когда программисту необходимо вычислить сумму значений двух функций $F(x) + G(x)$, причём каждая из этих функций для своего вычисления требует больших затрат процессорного времени и производит много обменов данными с внешними запоминающими устройствами. В этом случае программисту выгодно распараллелить алгоритм решения задачи и породить в своей программе два *параллельных вычислительных процесса*, каждому из которых поручить вычисления одной из этих функций. Можно понять, что в этом случае вся программа будет посчитана за меньшее *физическое* время, чем при использовании одного вычислительного процесса. Действительно пока один

¹ Естественно, что *немедленное* продолжение выполнения программы пользователя возможно только в том случае, если ей не требуется сразу обрабатывать данные, которые должен предоставить канал. Например, если программа обратилась к каналу для чтения массива с диска в свою оперативную память, и пожелает тут же начать суммировать элементы этого массива, то такая программа будет переведена в состояние ожидания, пока канал не закончит чтения заказанного массива в память. Аналогично программа будет переведена в состояние ожидания, если она обратилась к каналу для *записи* некоторого своего массива из оперативной памяти на диск, и пожелала тут же начать присваивать элементам этого массива новые значения. Эти примеры, в частности, показывают, насколько усложняются большинство программ операционной системы в архитектуре с каналами ввода/вывода.

процесс будет производить обмен данными с медленными внешними устройствами, другой процесс может продолжать выполняться на центральном процессоре, а в случае с одним процессом вся программа была бы вынуждена ждать окончания обмена с внешним устройством. Стоит заметить, что скорость счёта программы с несколькими параллельными процессами ещё больше возрастёт на компьютерах, у которых более одного центрального процессора.

Подробно параллельные процессы Вы будете изучать в следующем семестре.

- Параллельное выполнение нескольких команд одной программы производится конвейером центрального процессора. В мощных ЭВМ центральный процессор может содержать несколько конвейеров. Например, один из конвейеров выполняет команды целочисленной арифметики, другой предназначен для обработки вещественных чисел, а третий – для всех остальных команд.
- Параллельная обработка данных в программе производится на так называемых *векторных ЭВМ*. У таких ЭВМ наряду с обычными (скалярными) регистрами есть и *векторные* регистры, которые могут хранить и выполнять операции над векторами целых или вещественных чисел. Пусть, например, у такой ЭВМ есть регистры axv и bxv , каждый из которых может хранить вектор из 64 чисел, тогда команда векторного сложения `addv axv, bxv` будет производить параллельное покомпонентное сложение всех элементов таких векторов по схеме $axv[i] := axv[i] + bxv[i]$. Можно сказать, что на векторных ЭВМ один поток (векторных) команд обрабатывает много потоков данных (поток векторных данных). Отсюда понятно сокращённое название ЭВМ такой архитектуры – ОКМД (по-английски SIMD).¹

Параллельная обработка команд и данных позволяет значительно увеличить производительность компьютера. Необходимо сказать, что в современных компьютерах обычно реализуется сразу несколько из рассмотренных выше уровней параллелизма. Познакомится с историей развития параллельной обработки данных можно, например, по книге [15]. Заметим, однако, что, несмотря на непрерывный рост мощности компьютеров, постоянно появляются всё новые задачи, для счёта которых необходимы ещё более мощные ЭВМ. Таким образом, к сожалению, рост сложности задач опережает рост производительности компьютеров.

Список литературы.

1. Г. Майерс. Архитектура современных ЭВМ (в 2-х книгах). – Мир, 1985.
2. Burks A.W., Goldstine H.H., von Neumann J. Preliminary Discussion of the Logical Design of an Electronic Computing Instrument. – Pt. I, vol. I, Institute for Advanced Study, Princeton, NJ, 1946.
3. Королёв Л.Н. Структуры ЭВМ и их математическое обеспечение. – Наука, 1978.
4. Любимский Э.З., Мартынюк В.В., Трифонов Н.П. Программирование. – Наука, 1980.
5. Пильщиков В.Н. Программирование на языке Ассемблера IBM PC. – Диалог-МИФИ, 1994.
6. Скэлтон Л.Дж. Персональная ЭВМ IBM PC и XT. Программирование на языке Ассемблера. – Радио и связь, 1991.
7. Абель П. Язык Ассемлера для IBM PC и программирования. – Высшая школа, 1992.
8. Нортон П., Соухэ Д. Язык Ассемлера IBM PC. – Компьютер, 1993.
9. Ю-Чжень Лю, Гибсон Г. Микропроцессоры семейства 8086/8088. – Радио и связь, 1987.
10. Донован Дж. Системное программирование. – Мир, 1975.
11. Брусенцов Н.П. Миникомпьютеры. – Наука, 1976, 272с.
12. Дейт К. Введение в системы баз данных. – Наука, 1980.

¹ Для полноты классификации можно упомянуть и архитектуру МКОД (MISD), при этом один поток данных обрабатывается многими потоками команд. В качестве примера можно привести специализированную ЭВМ, управляющую движением самолётов над аэропортом. В этой ЭВМ один поток данных от аэродромного радиолокатора обрабатывается многими параллельно работающими процессорами, каждый из которых следит за безопасностью полёта одного закреплённого за ним самолёта, давая ему при необходимости команды на изменение курса, чтобы предотвратить столкновение с другими самолётами. С другой стороны, однако, можно считать, что каждый процессор обрабатывает свою *копию* общих данных и рассматривать это как частный случай архитектуры МКМД.

13. Успенский В.А. Нестандартный, или нетрадиционный анализ. – Знание, серия Математика, кибернетика № 8, 1983.
14. Девис М. Прикладной нестандартный анализ. – Мир, 1980.
15. Головкин Б.А. Параллельные вычислительные системы. – Наука, 1980.
16. Г. Майерс. Архитектура современных ЭВМ (в 2-х книгах). – Мир, 1985.
17. Burks A.W., Goldstine H.H., von Neumann J. Preliminary Discussion of the Logical Design of an Electronic Computing Instrument. – Pt. I, vol. I, Institute for Advanced Study, Princeton, NJ, 1946.
18. Королёв Л.Н. Структуры ЭВМ и их математическое обеспечение. – Наука, 1978.
19. Любимский Э.З., Мартынюк В.В., Трифонов Н.П. Программирование. – Наука, 1980.
20. Пильщиков В.В. Программирование на языке Ассемблера IBM PC. – Диалог-МИФИ, 1994.
21. Скэлтон Л.Дж. Персональная ЭВМ IBM PC и XT. Программирование на языке Ассемблера. – Радио и связь, 1991.
22. Абель П. Язык Ассемлера для IBM PC и программирования. – Высшая школа, 1992.
23. Нортон П., Соухэ Д. Язык Ассемблера IBM PC. – Компьютер, 1993.
24. Ю-Чжень Лю, Гибсон Г. Микропроцессоры семейства 8086/8088. – Радио и связь, 1987.
25. Донован Дж. Системное программирование. – Мир, 1975.
26. Брусенцов Н.П. Миникомпьютеры. – Наука, 1976, 272с.
27. Дейт К. Введение в системы баз данных. – Наука, 1980.
28. Успенский В.А. Нестандартный, или нетрадиционный анализ. – Знание, серия Математика, кибернетика № 8, 1983.
29. Девис М. Прикладной нестандартный анализ. – Мир, 1980.