Глава 1. Понятие об архитектуре ЭВМ

Это книга называется "Введение в архитектуру ЭВМ и системы программирования" и сначала нам нужно определить, что мы будем понимать под *архитектурой* компьютера. На бытовом уровне термин "архитектура" у большинства людей прочно ассоциируется с различными зданиями и другими инженерными сооружениями. Так, можно говорить об архитектуре готического собора, Эйфелевой башни или оперного театра. В других областях этот термин применяется достаточно редко, однако для компьютеров понятие "архитектура ЭВМ" уже прочно устоялось и широко используется начиная с 70-х годов прошлого века.

Заметим сначала, что в более широком смысле архитектура существует у любого достаточно сложного объекта, состоящего из отдельных, взаимодействующих между собой частей (компонентов). Так, определяя на бытовом уровне понятие "архитектура ЭВМ", обычно говорят, что архитектура – это все компоненты компьютера, их устройство, выполняемые ими функции, а также взаимосвязи между этими компонентами. Сейчас мы разберёмся, почему такое поверхностное определение архитектуры нас не будет удовлетворять.

Дело в том, что понятие *архитектуры* чего-либо существует не само по себе, а только в паре с другим понятием. Известно, например, что понятие *алгоритм* неразрывно связано с понятием *исполнитель алгоритма*.[1] 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 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 не без основания полагает, что этот дом состоит из фундамента, стен и крыши, имеет этажи, на каждом этаже есть квартиры, присутствует лестница, лифт, в квартирах есть комнаты, окна, двери 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, каким образом к дому подводятся инженерные коммуникации и многое другое.

Отсюда можно сделать вывод, что, изучая какой-либо объект, часто бывает удобно выделить различные **уровни** рассмотрения архитектуры этого объекта. Обычно удобно выделить три таких уровня, назовём их *внешний*, *концептуальный* и *внутренний*.[2] В качестве примера давайте 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 обязательно заливать бензин, знает назначение дворников на ветровом стекле, ремней безопасности и т.д. И этого эму 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 определённой марки и воду. Водитель знает назначение всех органов управления машиной, марку топлива, температуру окружающего воздуха, ниже которой необходимо заливать в машину не обычную воду, а

---

[1] Студенты факультета Вычислительной математики и кибернетики МГУ изучают эту тему в курсе первого семестра "Алгоритмы и алгоритмические языки".

[2] Число три является особым в человеческом мышлении, достаточно вспомнить "у отца было три сына", "загадать три желания" и т.д. По-видимому это связано с тем, что при рассуждениях для человека два – это ещё "слишком мало", а четыре – уже "слишком много" для первичного деления сложного объекта на части. В области компьютеров и программного обеспечения такие три уровни можно использовать, например, при описании баз данных. При этом описание структуры хранящихся данных (так называемые схемы данных) могут рассматриваться на внешнем, концептуальном и внутреннем уровнях [12].

так называемый антифриз и т.д. Обычно водитель обладает также некоторыми знаниями, позволяющими выполнить несложный ремонт машины. Ясно, наш водитель видит архитектуру своего 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 марки топлива, допустимую нагрузку на отдельные узлы автомобиля, антикоррозийные свойства внешнего корпуса, особенности работы системы безопасности и многое другое. Ясно, что обычный водитель машины, а тем более её пассажир, вполне может обойтись без всех этих знаний.

Не надо думать, что один уровень в*и*дения архитектуры ″хороший″, а другой – ″плохой″. Каждый из них необходим и достаточен для конкретного применения рассматриваемого объекта.[1] 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.

Для уровня университетского образования необходимо, чтобы наши выпускники, изучая какой-либо объект, достаточно ясно представляли себе, на каком уровне они его рассматривают и достаточен ли этот уровень для практической работы с этим объектом. При необходимости, разумеется, надо перейти на более глубокий уровень рассмотрения изучаемого объекта.

Перейдём теперь ближе к предмету нашего курса – архитектуре компьютеров. Все люди, которые так или иначе используют компьютеры в своей деятельности и имеют понятие об их архитектуре, обычно называются **пользователями**. Ясно, что в зависимости от того, на каком уровне они видят архитектуру компьютера, всех пользователей можно, хотя, конечно, и достаточно условно, разделить на уровни или группы (наверное, Вы уже не будете удивлены, что этих групп тоже три). Как правило, в научной литературе выделяют следующие группы пользователей.

1. **Конечные пользователи** (называемые также пользователи-непрограммисты). Для успешного использования компьютеров этим пользователям, как видно из названия, не нужно уметь программировать. 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* (application software) Это различные базы данных, текстовые редакторы, пакеты прикладных программ, системы автоматического перевода, обучающие, игровые и музыкальные программы и т.п. Этим пользователям достаточно видеть архитектуру компьютеров на *внешнем* уровне, этих людей абсолютное большинство, более 90% от общего числа всех пользователей. Вообще говоря, что бы там себе не воображали пользователи других уровней, компьютеры разрабатываются и выпускаются для нужд именно этих конечных пользователей-непрограммистов.
2. **Прикладные программисты**. Как уже ясно из их названия, 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 достаточно подробно знакомиться в нашем курсе). Прикладным программистам достаточно видеть архитектуру компьютеров на *концептуальном* уровне. Можно примерно считать, что прикладных программистов менее 10% от числа всех пользователей. Изучив программирование на языке Паскаль, Вы должны уже достаточно хорошо представлять себе архитектуру компьютера на этом уровне. Попробуйте, используя Ваш программистский опыт, сформулировать отличия в*и*дения архитектуры ЭВМ на этом уровне, по сравнению с предыдущим уровнем пользователей-непрограммистов.

---

[1] Не следует путать различные уровни рассмотрения объекта с рассмотрением этого же объекта с *разных сторон* (с разных точек зрения). Например, мы можем рассматривать автомобиль, сравнивая его с другими автомобилями, с точки зрения его экономичности, по дизайну и удобству эксплуатации, соотношению цены и качества и т.д. При рассмотрении объекта с некоторой стороны остальные стороны могут и совсем не приниматься во внимание. Здесь можно вспомнить известную восточную притчу о трёх слепых, которых подвели к слону и попросили описать его. Один слепой ощупал бок слона и сказал, что он похож на стену, второй, который стоял у ноги, утверждал, что слон похож на колонну, а третий сказал, что слон похож на шланг, так как держался за хобот.

3. **Системные программисты**. Это самая небольшая (менее одного 1%), но наиболее квалифицированная в программировании группа пользователей, которая видит архитектуру ЭВМ на *внутреннем* уровне. Основная деятельность системных программистов заключается в разработке *системного программного обеспечения*, которое предназначено для автоматизации процесса программирования и для эффективного управления аппаратурой ЭВМ. 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 разрабатывают и пишут свои программы для конечных пользователей. Системы программирования, по аналогии с промышленным производством, можно образно сравнить со *средствами производства* остальных программ.

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. На нашем факультете изучаются в основном второй и третий уровня, но иногда, в качестве примеров, мы совсем немного будем рассматривать и эти другие уровни в*и*дения 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, а также использование чертежей, графиков, рисунков, блок-схем и т.д. Именно таким способом в научной литературе обычно и описывается архитектура ЭВМ для *пользователей* разного уровня.
2. В качестве другого способа описания архитектуры компьютера на *внутреннем* уровне можно с определённым 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, достаточное для написания простейших программ.
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 различия в понимании этого понятия разными людьми. В основном формальные языки используются для описания архитектуры ЭВМ и её компонентов на инженерном уровне, эти языки достаточно 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 такого учебного компьютера).

Вы, конечно, уже знаете, что сейчас производятся самые разные компьютеры. Определим теперь, архитектуру **каких** именно ЭВМ мы будем изучать в нашем курсе. Сначала необходимо отметить, что мы будем рассматривать только так называемые универсальные цифровые (дискретные) компьютеры, которые в настоящее время составляют абсолютное большинство из всех выпускаемых и используемых ЭВМ. О других способах организации ЭВМ, в частности, о так называемых аналоговых (непрерывных) компьютерах, можно совсем немного (как говорится, "для общего развития") почитать в дополнительной главе в конце этой книги.

Итак, в нашем курсе мы построим изучение архитектуры ЭВМ таким образом.

1. Сначала мы рассмотрим архитектуру некоторой **абстрактной** машины (машины Фон Неймана).
2. Далее мы изучим специальные **учебные ЭВМ**, которые по своей архитектуре близка к самым первым из выпускавшихся цифровых компьютеров. На основе этой учебной ЭВМ мы изучим основные подходы к организации архитектуры компьютеров. Одну из этих учебных машин

мы рассмотрим более подробно, чтобы научиться писать для неё простейшие полные программы на языке машины.

3. Затем мы достаточно подробно изучим архитектуру первой (младшей) модели того **конкретного компьютера**, на котором Вы выполняете свои практические задания по программированию.
4. В заключение нашего курса мы рассмотрим отличительные особенности архитектуры современных компьютеров, а также проведём некоторый достаточно простой **сравнительный анализ** архитектуры основных классов универсальных ЭВМ.

Глава 2. Машина Фон Неймана


Джон фон Неймин 1903-57.

В 1946 работающий в то время в Англии венгерский математик 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, от хорошо известной Вам машины Тьюринга). Машина Тьюринга может обрабатывать входные данные любого объёма, поэтому этот исполнитель алгоритма принципиально нельзя реализовать из-за входящей в его архитектуру бесконечной ленты. Машина Фон Неймана не поддаётся реализации по другой причине: многие детали в архитектуре этого исполнителя алгоритма *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*. Для абстрактных структур данных при их использования необходимо предварительно выполнить конкретную реализацию: произвести отображение на структуры данных хранения и реализовать соответствующие операции над этими данными.[1]

Можно сказать, что в машине Фон Неймана зафиксированы те прогрессивные особенности архитектуры, которые в той или иной степени должны быть присущи, по мнению авторов этой абстрактной машины, всем компьютерам того времени. Разумеется, практически все современные ЭВМ по своей архитектуре в той или иной степени 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 мы тоже часто будем обращать 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 нескольких *поколений* ЭВМ [3].

На рис. 2.1 приведена схема машины Фон Неймана, как она изображается в большинстве учебников, посвящённых архитектуре ЭВМ. На этом рисунке толстыми (двойными) стрелками показаны *потоки команд и данных*, а тонкими – передача между отдельными устройствами компьютера *управляющих сигналов*. Машина Фон Неймана состоит из памяти, устройств ввода/вывода и *центрального процессора* (ЦП). Как видно из этого рисунка, основными блоками компьютера является память, устройства ввода и вывода, а также центральный процессор. Центральный процессор, в свою очередь, состоит из *устройства управления* (УУ) и *арифметико-логического устройства* (АЛУ). Сейчас мы последовательно рассмотрим устройство машины Фон Неймана и выполняемые ими функции, формулируя при этом принципы Фон Неймана.

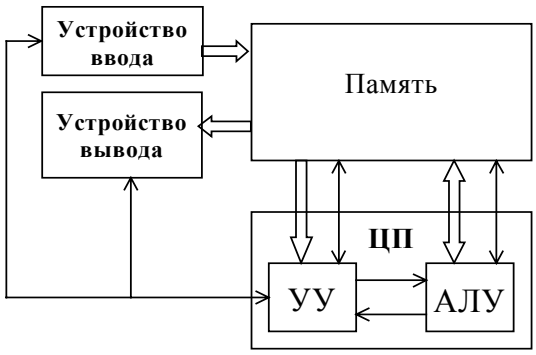


Рис. 2.1. Схема машины фон Неймана.

2.1. Память

Принцип линейности и однородности памяти.

Память машины Фон Неймана – это линейная (упорядоченная) однородная последовательность некоторых элементов, называемых *ячейками*. В любую ячейку памяти другие устройства машины (по толстым стрелкам на схеме рис. 2.1) 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).[2] Такая память в современных компьютерах называется *памятью с произвольным доступом* (Random Access Memory, RAM). На практике многие современные ЭВМ могут иметь участки памяти разных видов. Например, некоторые области памяти поддерживают только чтение информации (по-английски эта память называется Read Only Memory, ROM), данные в такую память записываются один раз при изготовлении этой памяти. Другие области памяти могут допускать запись, но за значительно большее время, чем в обычную память (это так называемая *полупостоянная* память, такой памятью комплектуется популярные в настоящее время карты флэш-памяти) и др.

Ячейки памяти в машине Фон Неймана нумеруются от нуля до некоторого положительного числа `N` (это и означает, что память *линейная*), причём число `N` в "настоящих" ЭВМ обычно является степенью двойки, минус единица. *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 одну *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 одну двоичную цифру или один *бит* информации. Восемь бит составляют один *байт*. Сам Фон Нейман тоже был сторонником использования двоичной системы счисления, что позволяло хорошо описывать архитектуру узлов ЭВМ с помощью логических (булевских) выражений.

Содержимое ячейки называется *машинным словом*. С точки зрения архитектуры, машинное слово – это минимальный объём данных, которым могут обмениваться между собой различные узлы машины по толстым стрелкам на схеме (не надо, однако, забывать о передаче управляющих сигналов по тонким стрелкам). Из каждой ячейки памяти можно считать *копию* машинного слова и передать её в другое устройство компьютера, 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. Дело в том, что в этой памяти (она называется *динамической* памятью, её использование экономически более выгодно) при чтении оригинал разрушается, и его приходится восстанавливать при каждом чтении данных. В компьютерах может использоваться и другой вид памяти, которая называется *статической* памятью, при чтении из неё оригинал не разрушается. Статическая память, по сравнению с динамической, является к тому же и более быстрой, однако и более дорогой.

Приведём типичные *характеристики памяти* современных ЭВМ.

1. Объём памяти – от сотен миллионов до нескольких миллиардов ячеек (обычно восьмиразрядных).
2. Скорость работы памяти: это *время доступа* (минимальная задержка на чтение слова) и *время цикла* (минимальная задержка на повторное чтение из одной и той же ячейки) – порядка единиц и десятков наносекунд (1 секунда = 10^9 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 нашего изучения архитектуры ЭВМ.

---

[1] Студенты факультета Вычислительной математики и кибернетики МГУ изучают эту тему в курсе первого семестра "Алгоритмы и алгоритмические языки".

[2] Разумеется, время чтения из ячейки памяти может не совпадать со временем записи в неё.

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

Из неразличимости команд и данных вытекает очевидное следствие – **принцип хранимой программы**. Этот принцип является очень важным, его суть состоит в том, что программа хранится в памяти вместе с числами. Чтобы понять важность этого принципа рассмотрим, а как программы вообще появляются в памяти машины. Понятно, что, во-первых, команды программы могут, наравне с числами, вводится в память "из внешнего мира" с помощью устройства ввода (заметим, что этот способ был основным в первых компьютерах). А теперь вспомним, что, как следует из материала предыдущего семестра, на вход алгоритма можно в качестве входных данных подавать запись некоторого другого алгоритма (в частности, свою собственную запись). Остаётся сделать последний шаг в этих рассуждениях и понять, что *выходными* данными алгоритма тоже может быть запись некоторого другого алгоритма.[1] Таким образом, одна программа может в качестве результата своей работы поместить в память компьютера другую программу.

Ещё одним следствием принципа хранимой программы является то, что сама программа, может *изменяться* во время счёта этой программы. Говорят также, что программа может *самомодифицироваться* (то есть изменять сама себя) во время своего счёта. В настоящее время самомодифицирующиеся программы применяются крайне редко, в основном для обеспечения очень компактного объёма программы, например в бортовых компьютерах небольших космических аппаратов. В то же время на первых ЭВМ, как мы увидим далее на примере одной из учебных машин, использование самомодифицирующихся программ часто было единственным способом реализации алгоритма на языке машины.

Заметим также, что, когда Джон фон Нейман писал эту свою основополагающую работу, многие из тогдашних ЭВМ хранили программу в памяти одного вида, а числа – в памяти другого вида, поэтому этот принцип являлся революционным. В современных ЭВМ и программы, и данные, как правило, хранятся в одной и той же памяти[2].

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. Обратите внимание, что 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.[3] *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 её команд. Например, язык одной из наших учебных машин УМ-3 будет содержать 19 команд, а машинный язык компьютера, на котором Вы выполняете свои практические задания, содержит более трёхсот команд.

Принцип последовательного выполнения команд. Устройство управления выполняет некоторую команду *от начала до конца*, а затем по определённому правилу выбирает следующую коман-

---

[1] В частности, можно написать программу, которая выводит запись своего собственного текста, Вы можете попробовать сделать это на некотором языке программирования высокого уровня (например, на Паскале). Как это делается можно посмотреть в книге [21].

[2] В современных ЭВМ некоторые программы, называемые Базовыми процедурами ввода/вывода (BIOS) крайне нежелательно изменять (стирать) во время работы компьютера. Такие программы располагают в уже упомянутой ранее памяти типа ROM, закрытой для записи.

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

ду для выполнения, затем следующую и т.д. При этом каждая команда либо сама явно указывает на команду, которая будет выполняться следующей (такие команды называются командами перехода), либо следующей будет выполняться команда из ячейки, расположенной в памяти непосредственно вслед за выполняемой командой. 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, например для машины Тьюринга это чтения из клетки ленты символа, которого нет в заголовке ни одной колонки таблицы.

2.3. Арифметико–Логическое Устройство

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

1. Считать содержимое некоторой ячейки памяти (машинное слово), т.е. поместить копию этого машинного слова в некоторую другую ячейку, расположенную в самом АЛУ. Если такие ячейки памяти расположенны не в основной памяти, а в других устройствах ЭВМ, то они называются *регистровой памятью* или просто *регистрами.*
2. Записать машинное слово в некоторую ячейку памяти – поместить копию содержимого одного из своих регистров в эту ячейку памяти. Когда не имеет значения, какая операция (чтение или запись) производится, говорят, что происходит *обмен* машинным словом между регистром и основной памятью ЭВМ.
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 этой операции на третий регистр (называемый в русскоязычной литературе, как правило, сумматором `S`).[1]

2.4. Взаимодействие УУ и АЛУ

Революционность идей Джона Фон Неймана заключалась в строгой *специализации*: каждое устройство компьютера отвечает за выполнение только своих функций. Если раньше, например, память ЭВМ 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 данными в своих регистрах, устройство ввода вводило данные из "внешнего мира" в память и т.д. Таким образом, Джон Фон Нейман предложил жёстко распределить выполняемые ЭВМ функции между различными устройствами, что существенно упростило схему машины, и сделало более понятным её работу.

Устройство управления тоже имеет свои регистры, оно может считывать команды из памяти на специальный *регистр команд* `RK` (instruction register), на котором всегда хранится *текущая* выполняемая команда. Регистр УУ с именем `RA` называется *регистром адреса* (или *счётчиком адреса* – instruction counter), при выполнении текущей команды в него по определённым правилам записывается адрес *следующей* выполняемой команды (первую букву в сокращении слова регистр мы будем в дальнейшем изложении часто записывать латинской буквой `R`).

Рассмотрим, например, схему выполнения команды, реализующей оператор присваивания с операцией сложения двух чисел `z:=x+y` (здесь `x`, `y` и `z` – адреса ячеек памяти, в которых хранятся, соответственно, операнды и будет помещён результат операции сложения). После получения из памяти такой команды на регистр команд 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 устройства управления АЛУ производит операцию сложения чисел, находящихся 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` – переменные, обозначающие регистры АЛУ, `ПАМ` – массив ячеек, условно обозначающий память ЭВМ, а $\otimes$ – бинарная операция (в нашем случае это сложение, т.е. $\otimes = +$).

---

[1] В англоязычной литературе этот регистр называется `Accumulator` и сокращается до буквы `A`, с этим обозначением мы столкнёмся при изучении языка Ассемблера.

```
R1 := ПАМ[x]; R2 := ПАМ[y]; S: = R1⊗R2; ПАМ[z] := S;
```

В дальнейшем конструкция `ПАМ[A]`, как это принято в научной литературе по архитектуре ЭВМ, для краткости будет обозначаться как `<A>`, тогда наш пример выполнения команды перепишется так:

```
R1: = <x>; R2: = <y>; S: = R1⊗R2; <z> := S;
```

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

```
RK := <RA>; считать из ячейки памяти с адресом RA очередную команду на регистр команд RK;
RA := RA+1; увеличить счётчик адреса на единицу;
Выполнить очередную команду в регистре RK.
```

Затем по такой же схеме из трёх шагов выполняется следующая команда и т.д. Заметим, что после выполнения очередной команды ЭВМ "не помнит", какую именно команду она только что выполнила. Напомним, что по такому же принципу выполняли свои "команды" и знакомые нам абстрактные исполнители алгоритмов – машина Тьюринга и Нормальные алгоритмы Маркова.

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 одной из наших учебных машин).

Современные ЭВМ в той или иной степени нарушают практически **все** принципы Фон Неймана, исключение, пожалуй, составляет только принцип хранимой программы. Например, существуют компьютеры, которые различают команды и данные. В них каждая ячейка основной памяти кроме собственно машинного слова хранит ещё специальный признак, называемый *тэгом* (tag), который и определяет, чем является это машинное слово. Так нарушается принцип неразличимости команд и чисел. В такой архитектуре при попытке выполнить число как команду, либо складывать команды как числа, центральным процессором будет зафиксирована ошибка. Очевидно, что это позволяет повысить надёжность программирования на языке машины, не допуская, как часто говорят, случайного "выхода программы на константы".

Практически все современные ЭВМ нарушают принцип однородности и линейности памяти. Память может, например, состоять из двух частей со своей независимой нумерацией ячеек в каждой такой части (с такой архитектурой мы вскоре познакомимся); или быть двумерной, когда адрес ячейки задаётся не одним, а двумя числами; либо ячейки памяти могут вообще не иметь адресов (такая память называется *ассоциативной*)[1] и т.д.

Все современные достаточно мощные компьютеры нарушают и принцип последовательного выполнения команд: они могут одновременно 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 зафиксированы и другие принципы, которые самим Джоном Фон Нейманом явно не формулировались, так как, безусловно, считались самоочевидными. Так, например, предполагается, что во время выполнения программы не меняется число узлов компьютера и взаимосвязи между ними, не меняется число ячеек в оперативной памяти. Далее, например, безусловно считалось, что машинный язык при выполнении программы не изменяется (например, "вдруг" не появляются новые виды машинных команд) и т.д.[2]. 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. Кроме того, во время работы программы могут, как изменяться, так и появляются новые связи между элементами ЭВМ (например, в так называемых *транспьютерах*) и т.д.

В заключение нашего краткого рассмотрения машины Фон Неймана обратим внимание на одно важное свойство компьютеров, которое часто ускользает от внимания читателей. Закончив выполнение текущей команды, машина начисто "забывает" о том, что это была за команда, где она располагалась в па-

---

[1] Поиск нужной ячейки в ассоциативной памяти производится не по её адресу, а по содержимому хранящегося в этой ячейки машинного слова (например, считать из памяти первое машинное слово, хранящее целое число, кратное пяти и т.п.).

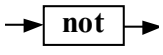
[2] В теории алгоритмов аналогом такого экзотического компьютера является, например, машина Тьюринга, у которой во время работы могут появляться и исчезать строки и столбцы её таблицы, изменяться содержимое клеток и т.д.

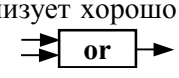
мяти, с какими операндами работала и т.д.[1] Выполнение каждой следующей команды практически начинается "с чистого листа". Это важное свойство позволяет, например, надолго прерывать выполнение программы, запомнив относительно небольшой объём информации (текущее состояние регистров компьютера, сведения об открытых файлах и т.д.). В дальнейшем в любой момент возможно возобновление счёта этой программы с прерванного места (разумеется, сама программа и её данные в памяти компьютера должны сохраниться, или же должны быть восстановлены в прежнем виде). Как мы узнаем дальше в нашем курсе, это свойство является основой для так называемого мультипрограммного режима работы компьютера.

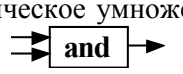
На этом мы закончим краткое описание машины Фон Неймана и принципов её работы. Отметим, что принципы Фон Неймана были положены в основу архитектуры одного из первых компьютеров под названием EDVAC [3].

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 машина может автоматически выполнять различные операции над данными, неужели она такая умная?

Аппаратура современных ЭВМ конструируется из некоторых относительно простых элементов, называемых в русскоязычной литературе *вентилями* (по-английски – circuits). Каждый вентиль является достаточно простой (электронной) схемой и реализует одну из логических операций, у него есть один или два *входа* (аргументы операции) и один *выход* (результат). На входах и выходе могут быть электрические сигналы двух видов: низкое напряжение (трактуется как ноль или логическое значение **false**) и высокое (ему соответствует единица или логическое значение **true**)[2]. Основные вентили следующие.

1. *Отрицание*, этот вентиль имеет один вход и один выход, он реализует хорошо известную Вам операцию отрицания **not** (НЕ) языка Паскаль. Другими словами, если на вход такого вентиля подаётся импульс высокого напряжения (значение **true)**, то на выходе получится низкое напряжение (значение **false)** и наоборот. Будем в наших схемах изображать этот вентиль так: 

2. *Дизъюнкция* или логическое сложение, этот вентиль реализует хорошо известную Вам операцию Паскаля **or** (И), мы будем изображать его как 

3. И, наконец, вентиль, реализующий *конъюнкцию* или логическое умножение, в Паскале это операция **and** (И), мы будем изображать его как 

Заметим, что чисто с технической точки зрения инженерам легче создавать вентили, задающие логические функции НЕ-И и НЕ-ИЛИ (для их реализации требуется на один транзистор меньше, чем для вентилей И и ИЛИ), но нас это интересовать не будет.

Далее, можно считать, что каждый вентиль срабатывает (т.е. преобразует входные сигналы в выходные) не непрерывно, а только тогда, когда на этот вентиль по специальному управляющему проводу приходит так называемый *тактовый импульс*. Заметим, что по этому принципу работают ЭВМ, которые называются *дискретными*, в отличие от *аналоговых* компьютеров, схемы в которых работают непрерывно (всё время). Подавляющее число современных ЭВМ являются дискретными, только их мы и будем изучать. О принципах работы аналоговых ЭВМ немного рассказывается в последней главе этой книги. Более подробно об этом можно прочесть в книгах [1,3].

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

---

[1] Некоторые команды могут изменять значения специальных флагов (или регистра результата выполнения команды), но это не меняет сути дела, т.к. не сохраняется информации, какая именно команда и когда изменила эти значения.

[2] В принципе вентили могут быть не только электрическими, но и, например, световыми, тогда на их вход по так называемым световодам подаются импульсы света различной интенсивности или поляризации.

Большинство современных интегральных схем собираются на одной небольшой (прямоугольной) пластинке полупроводника с размерами порядка сантиметра. Под микроскопом такая пластинка СБИС похожа на рельефный план большого города, со своими кварталами, домами, широкими проспектами и более узкими улицами. Интегральная схема может иметь от нескольких десятков до нескольких сотен внешних контактов.

Дадим представление о количестве вентилей в различных электронных устройствах. Например, для того, чтобы реализовать схему электронных часов, необходимо порядка 1000 вентилей, из 10000 вентилей уже можно собрать самый простой центральный процессор, а, например, центральные процессоры современных персональных ЭВМ реализуются на интегральной схеме, состоящей из нескольких миллионов вентилей.

В качестве примера рассмотрим очень простую интегральную схему, которая реализует функцию сложение двух одноразрядных двоичных целых чисел. Входными данными этой схемы являются значения одноразрядных переменных x и y, а результатом – их сумма, которая, в общем случае, является двухразрядным числом (обозначим разряды этого числа как a и b), формирующаяся как результат сложения x+y.[1] Одноразрядные величины можно трактовать как логические значения, поэтому можно считать, что наша схема реализует некоторую логическую функцию. Запишем таблицу истинности для этой логической 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 такими логическими формулами:

```
a = x<>y = (x or y) and not(x and y)
b = x and y
```

Реализуем нашу схему двоичного сумматора как набор вентилей, связанных проводниками (см. рис. 2.2. а). Здесь мы воспользовались тем обстоятельством, что входной электрический сигнал (например, приходящий на вход x) с помощью проводов можно "продублировать" и одновременно подать на вход сразу двух вентилей.[2]

Теперь, если реализовать этот двоичный сумматор в виде интегральной схемы (ИС), то она будет иметь не менее 7-ми внешних контактов (см. рис. 2.2 б). Это входные контакты x и y, выходные a и b, один контакт для подачи тактовых импульсов (о них мы уже говорили, когда объясняли работу вентилей), 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 последовательного прихода трёх тактовых импульсов (как говорят, за три такта). Современные компьютеры обычно реализуют более сложные схемы, которые выполняют суммирование многоразрядных целых чисел за один такт или два такта.

Заметим, что основная память компьютеров также реализуется в виде набора нескольких сверхбольших интегральных схем. Пользователи, заглядывавшие внутрь персонального компьютера, должны представлять себе вид маленьких плат такой памяти, на каждой из которых расположено несколько (обычно восемь) интегральных схем основной памяти.

---

[1] Более строго, в научной литературе такая схема называется полусумматором, но для простоты изложения мы не будем принимать это во внимание.

[2] Замечание для продвинутых студентов. Построение этой схемы легче понять, если преобразовать логические формулы, выражающие зависимость результата (b и a) от аргументов (x и y) в так называемую префиксную форму записи. В префиксной форме записи операнды бинарных операций записываются не между знаками операций, а перед ними. Например, операция `x and y` будет теперь записываться как `x y and`, аналогично вместо одноместной операции `not x` записывается `x not`. Эта форма записи называется ещё прямой польской записью в честь предложившего её польского математика и логика Лукашевича. В префиксной форме записи можно не использовать круглые скобки, так как выражение всегда вычисляется слева-направо, при этом операция выполняется, как только перед ней образуется необходимое число операндов.. В такой записи величины a и b будут определяться такими логическими формулами:

```
a = x y or x y and not and
b = x y and
```

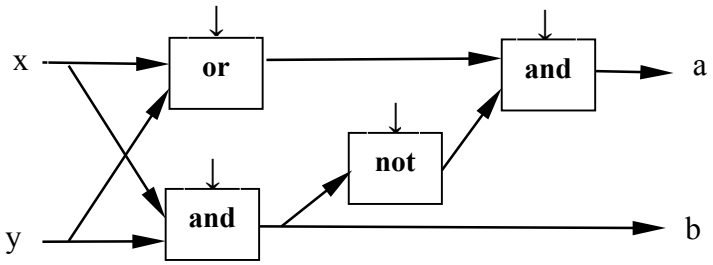


Рис. 2.2. а) Сборка двоичного сумматора из вентилей, ↓ – тактовые импульсы.

Ясно, что скорость работы интегральной схемы напрямую зависит от частоты прихода тактовых импульсов, называемой *тактовой частотой*. У современных ЭВМ не очень высокой производительности тактовые импульсы приходят на схемы основной *памяти* с частотой несколько сотен миллионов раз в секунду, а на схемы *центрального процессора* – ещё примерно в 10 раз чаще. Это должно дать Вам представление о скорости работы электронных компонент современных ЭВМ

Рис. 2.2. б). Пример ИС двоичного сумматора.

Теперь Вы должны почувствовать разницу в в*и*дении, например, центрального процессора, системным программистом (на внутреннем уровне по нашей классификации), от в*и*дения того же процессора инженером-конструктором ЭВМ. Для системного программиста архитектура центрального процессора представляется в виде устройств УУ и АЛУ, имеющих регистры, обменивающихся командами и числами с основной памятью, выполняющим последовательность команд программы по определённым правилам и т.д. В то же время для инженера-конструктора центральный процессор представляется в виде сложной структуры, состоящей из узлов-вентилей, соединённых проводниками и такими радиотехническими элементами, как сопротивлениями, конденсаторами и индуктивностями.[1] По этой структуре распространяются электрические сигналы, которые в дискретные моменты времени (при приходе на вентили тактовых импульсов) преобразуются логическими операциями.

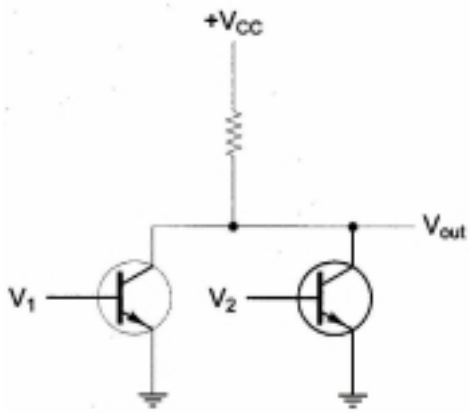


Рис. 2.3. Радиотехническая схема вентиля "не или".

Итак, мы совсем немного заглянули на *инженерный* уровень архитектуры ЭВМ. Разумеется, можно и дальше продолжать спуск по этим уровням, например, на рис. 2.3 показана простейшая *ра-*

[1] В математике такая структура называется графом – это любой набор узлов, соединённых дугами.

диотехническая схема, которая имеет два входа V_1 и V_2 и один выход V_{out} и функционирует аналогично вентилю "не или" `Vout:=not(V1 or V2)` (без комментариев!).

Глава 3. Учебная машина

Теперь мы, используя принципы Фон Неймана в качестве базовых, начнём конструировать свою собственную учебную ЭВМ. Для изучения особенностей архитектуры ЭВМ нам понадобятся несколько различных учебных машин, однако, одну из них мы изучим более подробно, чтобы иметь возможность писать для неё полные программы на языке машины. Будем называть эту машину УМ-3 (смысл этого названия – учебная машина трёхадресная), эта наша учебная машина будет удовлетворять *всем* принципам Фон Неймана.

Для построения конкретной ЭВМ нам необходимо строго определить, как будет устроена память этой машины, задать набор машинных команд, определить правила взаимодействия с внешним миром (возможности ввода/вывода) и т.д.

Сначала скажем, как будет устроена память нашей машины. Память учебной машины будет состоять из 512 ячеек, имеющих адреса от 0 до 511. Каждая ячейка состоит из 32 двоичных разрядов.[1] Записанное в ячейке машинное слово может трактоваться как одно целое или вещественное *число* (представляются они, естественно, по-разному) или одна *команда*. Машинное слово, трактуемое как команда, будет состоять из четырёх полей и представляться в следующей форме:

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

Здесь первое поле с именем КОП – содержит число от 0 до 31, которое задаёт номер (код) операции машинной команды, а поля с именами A1, A2 и A3 – задают адреса операндов. Таким образом, в каждой команде задаются адреса двух аргументов (это A2 и A3) и адрес результата операции A1.

Определим, какие регистры находятся в устройстве управления:

- RA – регистр, называемый *счётчиком адреса,* он имеет 9 разрядов и после выполнения текущей команды будет содержать адрес команды, которая должна будет выполняться следующей;
- RK – регистр команд имеет 32 разряда и содержит текущую выполняемую команду, которая, как мы определили, имеет код операции КОП и адреса операндов A1, A2 и A3;
- w – регистр длиной два бита с названием «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 следующему правилу (S – числовой результат арифметической операции, полученной на регистре-сумматоре АЛУ):

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

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

В таблице 3.1 приведёны все команды учебной машины УМ-3. Множество всех команд называется, как мы уже говорили, *языком машины*.

Как мы вскоре увидим, даже этого маленького набора команд окажется достаточно для программирования практически всех задач, которые Вы выполняли на языке Паскаль.[2] Более того, внимательное рассмотрение языка нашей машины выявляет даже его *избыточность*. Так, например, вместо команды безусловного перехода использовать команду условного перехода, в которой все три адреса совпадают. Можно сказать, что при разработке архитектуры нашей учебной машины мы отда-

---

[1] По современным понятиям это очень маленькая память (всего 2 Kb), однако, в первых ЭВМ для реализации даже такой памяти требовалось несколько больших шкафов, набитых электронными лампами (по одной лампе на бит – более 16 тысяч электронных ламп). Для тех студентов, которые никогда не видели электронных ламп, можно сказать, что по размерам они походили на маленькие электрические лампочки (например такие, как используются в холодильниках).

[2] На самом деле этот набор команд полный: его достаточно для программирования всех задач, для решения которых хватает ресурсов нашей учебной машины. И пусть Вас не смущает отсутствие многих типов данных, привычных нам в Паскаля, так как, например, символы можно хранить в виде целых чисел – номеров символов в алфавите, логические значения – в виде целых чисел 0 и 1 и т.д.

ли некоторое предпочтение удобству программирования в ущерб экономии электронных схем центрального процессора. В то же время мы не пошли по этому пути дальше, например, не стали вводить в язык машины команду вычисления абсолютной величины или квадратного корня (такие команды часто реализовывались в ЭВМ первого поколения).

Таблица 3.1. Команды учебной машины.

КОП	Смысл операции 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: <A1>:=<A3>
10	ЦЕЛ – вещественное в целое: <A1>:=Round(<A3>)
20	ВЕЩ – целое в вещественное: <A1>:=Real(<A3>)
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 **for** i:=1 **to** A2 **do** Read(<A1+i-1>)
15	ВЫВ – вывод A2 вещественных чисел, начиная с адреса A1 **for** i:=1 **to** A2 **do** Write(<A1+i-1>)
06	ВВЦ – ввод массива целых чисел, аналогично ВВВ
16	ВЫЦ – вывод массива целых чисел, аналогично ВЫВ

3.1. Схема выполнения команд

Рассмотрим теперь схему работы центрального процессора. В соответствие с принципом Фон Неймана последовательного выполнения команд, центральный процессор выполняет одну команду за другой, пока не выполнит команду СТОП или же пока очередная команда не зафиксирует при своём выполнении аварийную ситуацию. Каждая команда выполняется в центральном процессоре по следующему алгоритму:

repeat

1. RK := <RA>; чтение очередной команды из ячейки памяти с адресом RA на регистр команд RK в устройстве управления.
2. RA := RA + 1; увеличение счётчика адреса на единицу, чтобы следующая команда выполнялась из следующей ячейки памяти.
3. Выполнение операции, заданной в коде операции (КОП) в АЛУ. При несуществующем коде операции или других ошибках при выполнении команды (например, делении на ноль) производится присваивание Err := 1.

until (Err=1) **or** (КОП=СТОП)

Уточним теперь работу арифметико-логического устройства в нашей УМ-3. Все *бинарные* операции (т.е. те, которые имеют два аргумента и один результат) выполняются в АЛУ нашей учебной машины по схеме: <A1>:=<A2>⊗<A3> (⊗ – любая бинарная операция). Для реализации таких операций в арифметико-логическом устройстве предусмотрены регистры первого (R1) и второго (R2) операндов, а также регистр сумматора (S), на котором размещается результат операции. Таким образом, как и в машине Фон Неймана, АЛУ, подчиняясь управляющим сигналам со стороны УУ, выполняет бинарную операцию по схеме

```
R1 := <A2>; R2 := <A3>; S := R1⊗R2; <A1> := S;
```

Формирование регистра `w` для арифметических операций

Теперь нам осталось определить условие начала работы программы. Прежде всего, программа каким-то образом должна оказаться в памяти ЭВМ. Вообще говоря, исходя из принципа неразличимости команд и данных, нашу программу можно, например, представить в виде массива целых чисел и ввести в память, используя команду ввода (с кодом операции 15, см. таблицу 3.1). Заметим, однако, что для *начального* ввода программы в память мы не можем использовать *команды ввода* из языка машины, так как для этого в памяти уже должна находиться хотя бы одна команда. Чтобы разорвать этот замкнутый круг, в современных ЭВМ обычно часть памяти отводят для постоянного хранения специальной небольшой программы, которая называется программой *начальной загрузки*. Эта программа начинает автоматически выполняться при включении ЭВМ.[1] Память, занимаемая программой начальной загрузки, недоступна для записи, такую память, как мы уже говорили, обычно обозначают английской аббревиатурой ROM – Read Only Memory. Ясно, что наличие программы начальной загрузки нарушает принцип Фон Неймана однородности памяти.

В учебной ЭВМ мы не сможем пойти по этому пути загрузки программы по двум причинам. Во-первых, наша память и так очень маленькая, и отводить часть её для постоянного хранения программы начальной загрузки нерационально, а, во-вторых, мы не хотим отступать от принципа Фон Неймана однородности памяти. В УМ-3 первичной загрузкой программы в память и формированием начальных значений регистров в устройстве управления занимается *устройство ввода*. Для этого на устройстве ввода имеется специальная кнопка ПУСК ( ☺). При нажатии этой кнопки **устройство ввода** самостоятельно (без сигналов со стороны устройства управления, которое ещё не функционирует) выполняет следующую последовательность действий:

1. Производит ввод расположенного на устройстве ввода массива машинных слов в память, начиная с *первой ячейки*; этот массив машинных слов заканчивается специальным признаком *конца ввода массива*.[2]
2. `RA := 1`; первой будет выполняться команда из ячейки а номером один.
3. `w := 0`; начальное значение признака результата нулевое.
4. `Err := 0`; признак ошибки сбрасывается (устанавливается равным **`false`**).

Как видим, при нажатии кнопки ПУСК устройство ввода выполняет функции упомянутой выше программы начальной загрузки современных ЭВМ. Вот теперь всё готово для автоматической работы центрального процессора по загруженной в память программе, и за дело принимается устройство управления.

Таким образом, мы полностью определили условия начала и конца работы нашей учебной машины как алгоритмической системы. Здесь по аналогии хорошо вспомнить, как условия начала работы определялись, скажем, для машины Тьюринга: на ленте расположено входное слово, головка установлена у первого слева символа этого слова, таблица готова к работе и машина находится в первом состоянии.

Заметим, что 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 средине прошлого века.

3.2. Примеры программ для учебной машины

Далее мы рассмотрим несколько простых полных программ для нашей учебной машины. Эти программы призваны проиллюстрировать основные приёмы программирования на языке машины для

---

[1] Перед вводом в память первой после включения компьютера программы, эта программа начальной загрузки предварительно проверяет правильность функционирования основных узлов ЭВМ и выполняет ещё некоторые действия.

[2] В качестве внешних носителей информации в первых ЭВМ часто использовались специальные картонные *перфокарты*, на которых отверстиями кодировались хранимые данные. Например, для нашей УМ-3 каждое машинное слово будет занимать на перфокарте 32 позиции, причём в тех позициях, где машинное слово имеет разряд единицу, пробивается отверстие. Признак конца ввода массива является специальным словом, которое имеет дополнительные (сверх 32-х) позиции (обычно признак конца ввода располагался на отдельной перфокарте, в память он не вводится). Заметим, что такой массив с признаком конца ввода аналогичен более современному понятию последовательного файла машинных слов.

первых ЭВМ, удовлетворяющих всем принципам Фон Неймана. Эти примеры позволят нам лучше понять способ работы учебной ЭВМ и, следовательно, её архитектуру.

3.2.1. Пример 1. Оператор присваивания

Составим программу, которая вводит целое число x и реализует арифметический оператор присваивания языка Паскаль:

```
y := (x+1)^2 mod (x-1)^2
```

На языке Паскаль эта программа может выглядеть, например, так:

```
Program First(input,output);
   Var x,y: integer;
Begin Read(x); y:=sqr(x+1) mod sqr(x-1); Write(y) end.
```

Текст нашей первой программы с комментариями приведён на рис. 3.1.

№	Команда				Комментарий
001	06	101	001	000	Read(x)
2	11	103	101	009	r1 := (x+1)
3	13	103	103	103	r1 := (x+1)^2
4	12	104	101	009	r2 := (x-1)
5	13	104	104	104	r2 := (x-1)^2
6	24	102	103	104	y := r1 mod r2
7	16	102	001	000	Write(y)
8	31	000	000	000	Стоп
9	00	000	000	001	Целая константа 1

Рис 3.1. Текст программы первого примера.

Подробно прокомментируем составление этой программы. Видно, что 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 под хранения переменных самим. Сделаем сначала 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). Поэтому, начиная со 101 ячейки, память будет свободна. Пусть тогда для хранения значения переменной x мы выделим 101 ячейку, а переменной y – 102 ячейку памяти. Остальные переменные при необходимости будем размещать в последующих ячейках памяти. Для реализации этой программы нам понадобятся дополнительные (как говорят, *рабочие*) переменные r1 и r2, которые мы разместим в ячейках 103 и 104 соответственно.

Поймём теперь, что 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, по принципу Фон Неймана однородности памяти ничто не помешает 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.

Важно понять, что в начале работы программы имеют значения только те ячейки памяти, в которые произведён ввод данных по кнопке ПУСК. Отсюда следует, что константы, как и переменные с начальным значением, следует располагать в *тексте программы* ("маскируя" их под команды) и загружать в память вместе с программой при нажатии кнопки ПУСК. Разумеется, такие константы и 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.

Далее, мы воспользовались принципом неразличимости команд и чисел, и представили целочисленную константу 1 в виде команды `00 000 000 001` (скоро мы изучим внутреннее машинное представление целых чисел и поймём, что как машинные слова они совпадают). Этот приём позволил нам ввести эту константу в виде команды (обычно так и поступали программисты первых ЭВМ).

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 неизвестными (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 программистом в последнюю очередь, будут обозначаться подчёркиванием.

Для программиста запись программы состоит из строк, каждая строка снабжается номером ячейки, куда будет помещаться это машинной слово (команда, константа или переменная с начальным значением) при загрузке программы. Вслед за номером задаются все поля команды, затем программист может указать комментарий (разумеется, комментарий *не вводится* в память ЭВМ, для него нет места в машинном слове).. Номера ячеек, кодов операций и адреса операндов будем для своего удобства записывать в десятичном виде, хотя первые программисты использовали для этого 8-ую или 16-ую системы счисления. Кроме того, так как числа в памяти неотличимы от команд, то, как уже говорилось, будем записывать их тоже чаще всего в виде команд.

Для ввода программы в ЭВМ первые программисты кодировали все машинные слова в двоичном виде и переносили их на какой-нибудь *носитель данных* для устройства ввода (например, на картонные перфокарты). После этого оставалось снабдить такую, как говорили, *колоду перфокарт*, перфокартой-признаком конца ввода, поместить на устройство ввода и нажать кнопку ПУСК. В нашем случае надо поместить на устройство ввода два массива машинных слов – саму программу (9 машинных слов с признаком конца ввода) и число `x` (одно машинное слово). Как мы уже говорили, первый массив заканчивался специальным признаком конца ввода, так что устройство ввода само узнает, 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 значения 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 номера указывать его мнемоническое обозначение, приведённое в таблице команд 3.1. Разумеется, потом, перед вводом программы необходимо будет заменить эти мнемонические обозначения соответствующими им двоичными числами.

Для определения того, является ли значение некоторого числа `x` больше, меньше или равным константе `2`, мы будем выполнять команду вычитания `x–2`, получая как побочный эффект этой операции в регистре признаке результата `w` значение `0` при `x=2`, значение `1` при `x<2` и значение `2` при `x>2`. При этом сам результат операции вычитания нам не нужен, но по принятому в УМ-3 формату команд указание адреса ячейки для записи результата является обязательным. Для записи таких ненужных значений мы будем чаще всего использовать ячейку с номером `0`.[1] 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 возвращается ноль, а запись в ячейку с адресом ноль физически не осуществляется (на практике такой принцип работы с этой ячейкой почти всегда удобнее).

Для хранения переменных `x` и `y` выделим, например, ячейки `100` и `101` соответственно. Заметим, что программист сам определяет порядок размещения в программе трёх ветвей нашего условного оператора присваивания. Мы в нашем примере будем сначала располагать вторую ветвь (для `x=2`), затем первую (`x<2`), а потом третью (`x>2`). Заметим, что значение `x+2` понадобится нам при программировании, как в первой, так и в третьей ветви условного оператора, поэтому более рационально вычисление выражения `x+2` расположить *перед* вычислением условного оператора, что мы и сделали в нашем примере. На рис. 3.2 приведён текст этой программы.

---

[1] Заметим, что в нашей архитектуре программа специально вводится в память по кнопке ПУСК, начиная с первой ячейки, оставляя программисту для подобных целей свободной ячейку с номером ноль.

Обратите внимание, что целочисленная константа 2 располагается в таком месте программы (за командой безусловного перехода), где она не может начать выполняться как команда. Заметим также, что эта константа неотличима от команды ПЕР 000 000 002 пересылки содержимого второй ячейки памяти в нулевую ячейку (что и позволило нам записать эту константу в виде такой команды). Именно эта команда и выполнялась бы, если бы эта константа была (случайно) выбрана на регистр команд устройства управления.

№	Команда				Комментарий
001	ВВЦ	100	001	000	Read(x)
2	СЛЦ	101	100	011	y := x+2
3	ВЧЦ	000	100	011	<000> := x–2; формирование w
4	УСЛ	005	007	009	**Case** w **of** 0: **goto** 005; 1: **goto** 007; 2: **goto** 009 **end**
5	ВЫЦ	011	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	011	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.

Разберёмся теперь, на каком же языке мы написали наш второй пример для УМ-3. Ясно, что это не язык машины, так как он по определению состоит только из 32-хразрядных двоичных чисел, а у нас вместо кодов операции стоят мнемонические обозначения, адреса написаны в десятичной форме записи, да ещё и комментарии присутствуют. Такой язык программирования для первых ЭВМ обычно называли *псевдокодом*, что хорошо отражало его сущность. Для того чтобы выполнить программу на псевдокоде, её надо предварительно перевести (или, как принято говорить, *оттранслировать*) на язык машины. Очевидно, что для этого необходимо заменить все мнемонические обозначения на соответствующие им числовые коды операций, а затем перевести все десятичные числа в двоичную систему счисления, а также отбросить номера ячеек из первой колонки и комментарии. Для первых ЭВМ эту работу выполняли сами программисты. В дальнейшем псевдокоды постепенно развивались и превратились в языки низкого уровня – *ассемблеры* (в русскоязычной литературе их сначала называли *автокодами*, подразумевая, что они *автоматически* переводят (*кодируют*) программу с псевдокода на язык машины). Наши следующие программы для УМ-3 мы также будем для удобства писать не на "чистом" языке машины, а на таком псевдокоде.

3.2.3. Пример 3. Реализация цикла

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

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

Наша программа должна вводить значение переменной n и выводить результат работы y. Для хранения переменных n,y и параметра цикла i выделены ячейки 100, 101 и 102 соответственно. В этом алгоритме мы реализуем *цикл с предусловием*, поэтому при вводе значения n<1 тело цикла не будет выполняться ни одного раза, и наша программа будет выдавать нулевой результат, что не противоречит математическому смыслу нашей суммы. На рис. 3.3 приведена возможная программа для решения этой задачи.

Сделаем некоторые замечания к этой программе. Каждый член ряда является по смыслу задачи вещественным числом, в то время как параметр цикла i – целая величина. В машинном языке у нас нет команды деления вещественного числа на целое, поэтому при вычислении величины очередного слагаемого 1.0/i нам пришлось отдельной командой ВЕЩ 000 000 102 преобразовать значение целой переменной i в вещественной значение. Обратите также внимание, что для нашей учебной машины мы ещё не определили формат представления вещественных чисел (мы сделаем это позже),

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	ВЧЦ	000	102	100	<000> := i-n; формирование w
5	УСЛ	006	006	011	**if** i>n **then goto** 011
6	ВЕЩ	000	000	102	<000> := Real(i)
7	ДЕВ	000	014	000	<000> := 1.0/<000>
8	СЛВ	101	101	000	y := y+<000>
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	<1.0>				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:

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

Сделаем естественное предположение, что длина нашей программы не будет превышать 199 ячеек, и поместим массив x, начиная с 200-ой ячейки памяти. Вещественную переменную S с начальным значением `0.0` и целую переменную n с начальным значением `100` поместим в конце текста программы (после команды СТОП), и будем вводить в память вместе с командами при нажатии кнопки ПУСК. Таким образом, в качестве счётчика цикла для нас будет удобнее использовать не целую переменную i, изменяющуюся от `1` до `100`, как в приведённой выше формуле, а переменную n с начальным значением `100`, которая будет изменяться от `100` до `1`. Другими словами, мы реализуем цикл, который на Паскале можно было бы записать в виде

```
repeat S:=S+x[101-n]; n:=n-1 until n=0
```

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 четвёртого примера.

Рассматриваемая программа выделяется новым приёмом программирования, она является *самомодифицирующейся программой*, о которых мы говорили при изучении машины Фон Неймана. Обратим внимание на третью строку программы. Содержащаяся в ней команда изменяет исходный код программы (меняет команду в ячейке с адресом 2) для организации цикла перебора элементов

массива.[1] При первом выполнении этой команды она адресуется к первому элементу массива, затем ко второму и т.д. Для перехода от одного элемента массива к следующему модифицируемая команда рассматривается как *целое число*, к которому прибавляется специально подобранная *константа переадресации*.[2] 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 повышает сложность программирования, может привести к путанице и вызвать появление ошибок. Заметим также, что такую программу нельзя повторно выполнить, просто передав управление на её первую команду,[3] так как нужно будет предварительно восстановить исходный вид всех модифицированных команд. Кроме того, самомодифицирующуюся программу трудно понимать и вносить в неё изменения. Представьте, что Вам необходимо составить алгоритм для машины Тьюринга, в которой можно изменять команды в клетках таблицы (например, заменить команду движения головки по ленте влево на движение вправо). Настоящий кошмар для программиста!

В нашей учебной машине, однако, самомодифицирующаяся программа – это *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 в современных компьютерах, как уже говорилось, обычно не используется.

3.3. Формальное описание учебной машины

При описании архитектуры учебной ЭВМ на естественном языке многие вопросы остались нераскрытыми. Что, например, будет происходить после выполнения команды из ячейки с адресом `511`? Какое значение после нажатия кнопки ПУСК имеют ячейки, расположенные вне введённого массива машинных слов? Как представляются целые и вещественные числа? Как будет, например, выполняться такая машинная команда ввода массива: `ВВВ 500 100 000`?

Для ответа на почти все такие вопросы мы приведём *формальное описание* нашей учебной машины. При этом в качестве метаязыка мы будем использовать Турбо-Паскаль, на котором Вы выполняете Ваши практические работы. Другими словами, мы напишем программу, выполнение которой *моделирует* работу нашей учебной машины, т.е. наша машина, по определению, работает "почти так же", как и эта написанная нами программа-модель на Паскале.

Ниже приведена реализация учебной машины на языке Турбо-Паскаль:

```
program УМ_3(input, output);
  const
    N = 511;
  type
    Address = 0..N;
    Tag = (kom, int, fl);  {В машинном слове может хранится команда, целое
                            или вещественное число}
    Komanda = packed record
                  KOP: 0..31;
                  A1, A2, A3: Address;
              end;
```

---

[1] Как уже говорилось, в теории алгоритмов некоторым аналогом такого исполнителя, например, является такая модификация машины Тьюринга, которая в процессе выполнения алгоритма может изменять клетки своей таблицы. Доказана теорема о том, что все задачи, которые может решать такая модифицированная машина Тьюринга, может решить и обычная машина Тьюринга, т.е. они эквивалентны.

[2] В выборе этой константы есть небольшая тонкость. Как мы вскоре узнаем, целые числа в нашей машине представляются в так называемом *дополнительном коде* (что это такое, мы скоро изучим). В дополнительном коде первый бит целого числа определяет его знак, поэтому команды с кодами операций больше 15 будут *отрицательными* целыми числами, что следует учитывать при подборе константы переадресации. В нашем примере, однако, команда сложения с кодом операции СЛВ=01 является *положительным* целым числом и всё будет хорошо.

[3] Повторное выполнение находящейся в памяти программы может оказаться весьма полезным, например, при счёте нескольких вариантов задачи с разными входными данными.

```
    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: { СЛВ }
            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;
         31: { СТОП } ;
         { Реализация остальных кодов операций }
        else
          Err := True;
        end; { case }
       until Err or (KOP = 31)
end.
```

Прокомментируем эту программу, описывающую работу учебной машины. Отметим сначала, что некоторая трудность возникает при моделировании начального ввода программы в память учебной машины при нажатии кнопки ПУСК. В нашей модели для задания такого начального ввода мы использовали вызов процедуры с именем `Input_Program`, описание этой процедуры не приводится.[1]

[1] В архитектуре ЭВМ принято выделять центральную часть, куда входит основная (оперативная) память и центральный процессор. Вся остальная аппаратура ЭВМ относится к так называемой периферии (периферийным устройствам). Таким образом, наша модель на Паскале формально описывает только центральную часть учебной машины.

Для хранения машинных слов учебной машины мы описали тип данных `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] Таким образом, этот тип данных позволяет нам реализовать в паскалевской программе неразличимость представления команд, целых и вещественных чисел учебной машины.

Наша программа ведёт себя почти так же, как учебная машина. Одно из немногих мест, где это поведение расходится, показано в тексте программы комментариями с вопросительным знаком, например, при реализации команды сложения вещественных чисел. Программа на Паскале при переполнении (когда результат сложения не помещается в переменную `S`) просто производит аварийное завершение программы, а учебная машина сначала присваивает регистру `Err` значение `1,` а затем (неаварийно) останавливает выполнение программы.

Заметим, что наше формальное описание отвечает и на вопрос о том, как в учебной машине представляются целые и вещественные числа: точно так же, как в переменных соответствующих типов программы на Турбо-Паскале. Это представление мы подробно изучим в нашем курсе несколько позже.

В нашей модели мы реализовали выполнение только некоторых типичных команд из языка учебной машины. Реализацию остальных команд Вы легко можете выполнить сами.

Обратите также внимание, как в нашей модели вычисляется адрес следующей выполняемой команды: `RA: = (RA+1) mod (N+1)`. Такое правило перехода к следующей по порядку команде программы позволяет считать, 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.

Вопросы и упражнения

1. Объясните, как в нашей учебной машине должна выполняться такая команда ввода массива вещественных чисел `BBB 100 500 000`. Напишите соответствующую ветвь в формальном описании УМ-3 для реализации команды ввода вещественных чисел.
2. Объясните, почему для машины УМ-3 при решении некоторой задачи нельзя сделать такое распределение памяти: "Пусть массив `X` располагается в ячейках с адресами от 100 до 199, а константа `n=100` – в ячейке с адресом 200" ?
3. Реализуйте в модели на Паскале выполнение команд ввода/вывода учебной машины.

[1] Более привычный программистам тип `real` Турбо-Паскаля здесь не подходит, потому что имеет длину 48 бит, а не 32 бита, как нам нужно.

Глава 4. Введение в архитектуру ЭВМ

В этой главе мы введём основные определения и рассмотрим важные базовые понятия, которые будут нам необходимы для дальнейшего изучения архитектуры ЭВМ. С этой целью мы построим и изучим несколько учебных машин различной архитектуры, как бы кратко повторяя историю начального этапа развития вычислительной техники.

4.1. Адресность ЭВМ

Как мы уже упоминали, число адресов в машинной команде называется *адресностью* ЭВМ. Разнообразие архитектур ЭВМ предполагает, в частности, и разную адресность их систем команд. Рассмотрим схему выполнения команд на компьютерах с различным числом адресов операндов. По сравнению с изученной ранее учебной ЭВМ УМ-3 мы несколько увеличим аппаратные возможности (объём памяти и число возможных кодов операций), чтобы приблизить эти параметры к характеристикам "настоящих" машин первого поколения.

Будем предполагать, что для хранения кода операции в команде учебной машины отводится один байт (это 8 двоичных разрядов, что достаточно для представления 256 различных кодов операций), а для хранения каждого из адресов – по 3 байта (это обеспечивает доступ к объёму адресуемой памяти в 2^{24} ячеек). Ниже приведены форматы команд для ЭВМ различной адресности и схемы выполнения этих команд для случая бинарных (двуместных) операций (у таких операций два операнда и один результат, это, например, привычные для нас арифметические операции сложения, вычитания, умножения и деления). В дальнейшем для обобщённого обозначения любой бинарной операции мы будем использовать символ ⊗.

- **Трёхадресная машина.**

Трёхадресные команды являются наиболее естественными при составлении программы на машинном языке, так как большинство элементарных шагов в алгоритмах являются именно бинарными операциями. Пусть команда в трёхадресной машине имеет такой формат:

КОП	A1	A2	A3
8 разрядов	24 разряда	24 разряда	24 разряда

= 10 байт

Это модификация уже знакомой нам трёхадресной ЭВМ УМ-3, однако, теперь длина каждой команды увеличена до 10 байт. Схема выполнения команд такой машины её центральным процессором нам уже известна, она такая же, как на нашей учебной ЭВМ УМ-3:[1]

```
R1 := <A2>; R2 := <A3>; S := R1 ⊗ R2;
<A1> := S; {⊗ – любая бинарная операция}
```

- **Двухадресная машина.**

Сократим теперь число адресов в машинной команде с трёх до двух и мысленно построим двухадресную учебную машины (можно по аналогии с предыдущей учебной машиной назвать её УМ-2). Длина каждой команды для двухадресной ЭВМ будет равна 7 байт:

КОП	A1	A2
8 разрядов	24 разряда	24 разряда

= 7 байт

Схема выполнения бинарных двухадресных команд в центральном процессоре нашей машины УМ-2 будет такой:

```
R1 := <A1>; R2 := <A2>; S := R1 ⊗ R2; <A1> := S;
```

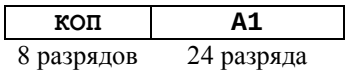
Заметим, что теперь, так как адресов у нас только два, для выполнения бинарной операции первый и второй операнды задаются в команде *явно* в качестве адресов, а местоположение результата операции задаётся неявно или, как говорят, *по умолчанию*. В рассмотренном выше случае двухадресных команд результат бинарной операции по умолчанию помещается на место первого операнда, уничтожая его старое значение (заметим, что большинство современных двухадресных машин, в том числе и та, на которой Вы сейчас работаете, функционируют именно по такой схеме).

---

[1] В схемах выполнения команд разной адресности мы не будем учитывать, что некоторые команды могут также вырабатывать признак результата (для УМ-3 этот признак помещался в регистр `w`).

- **Одноадресная машина.**

При дальнейшем сокращении числа адресов в машинной команде у нас получится уже одноадресная учебная машина УМ-1, команды которой имеют такую структуру:



= 4 байта

Длина каждой команды этой машины равна 4 байта. Схема выполнения одноадресных команд будет такой:

```
R1 := <A1>; S := S ⊗ R1;
```

Как видим, при выполнении бинарных операций в одноадресной ЭВМ уже только один второй операнд задаётся в команде явно, а первый операнд и результат операции задаются неявно – это регистр сумматора, обозначенный нами буквой S.

Для работы в одноадресной машине необходимы ещё две специальные одноадресные команды, которые имеют один операнд и один результат. Эти команды реализуют унарные (одноместные) операции. У нас это команда чтения числа из памяти на регистр сумматора:

```
СЧ A1
```

Она выполняется по схеме

```
S := <A1>
```

и команда записи значения из сумматора в память:

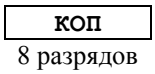
```
ЗП A1
```

Она выполняется по схеме

```
<A1> := S
```

- **Безадресная машина.**

Логическим завершением процесса уменьшения числа адресов в машинной команде является построение нульадресной (или безадресной) учебной ЭВМ (назовём её УМ-0). Почти все команды этой машины состоят только из кода операции и имеют длину один байт:



= 1 байт

В отличие от других рассмотренных выше учебных машин, новая безадресная машина кроме основной памяти, в которой, как обычно, хранится программа и данные, использует при своей работе также аппаратно реализованный в компьютере *стек*. Для обмена данными между основной памятью и стеком в язык машины вводятся две дополнительные *одноадресные* команды длиной по 4 байта. Это команда записи машинного слова в вершину стека из любой ячейки памяти

```
ВСТЕК A1
```

которая выполняется по схеме

```
R1 := <A1>; ВСТЕК(R1)
```

и команда чтения машинного слова из вершины стека в ячейку основной памяти (как обычно, при чтении машинное слово удаляется из стека)

```
ИЗСТЕКА A1
```

которая выполняется по схеме

```
R1 := ИЗСТЕКА; <A1> := R1
```

Вообще говоря, кроме указанных выше одноадресных команд записи в стек и чтения из стека, для удобства программирования в безадресной ЭВМ могут добавляться и некоторые другие одноадресные команды (например, команды безусловного и условного переходов, команда вызова процедуры и другие).[1] Таким образом, за исключением этого небольшого набора одноадресных команд, которые, как и в нашей предыдущей учебной одноадресной машине, 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)
```

[1] В отличие от команд записи из памяти в стек и чтения из стека в память, остальные одноадресные команды являются *избыточными*. Так, например, одноадресную команду условного перехода на ячейку с адресом A1 можно заменить двумя командами: одноадресной командой записи в стек значения адреса перехода A1 (в виде целого числа) и безадресной командой условного перехода по адресу, записанному в вершине стека.

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.[1]

Кроме рассмотренных выше видов машин, существовали и архитектуры ЭВМ с другим числом адресов. В качестве примера упомянем **четырёхадресные** машины, в четвёртом адресе которых дополнительно хранился ещё и адрес *следующей* выполняемой команды (для таких ЭВМ не нужна команда безусловного перехода). Собственно, адресов может быть и больше, с помощью таких команд можно, например, реализовать уже не бинарные операции, а функции от многих переменных.

Далее, существуют архитектуры ЭВМ, которые различаются не только *количеством* адресов в машинной команде, но и наличием в такой команде нескольких *кодов операций*. Такие ЭВМ обычно называются машинами с *очень длинным командным словом* (VLIW – Very Large Instruction Word).[2] В этих компьютерах, например, некоторые команды могут реализовывать операторы 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, содержащая два кода операции и четыре адреса аргументов, в наших предыдущих предположениях о размере адреса и кода операции, имеет длину 14 байт и, например, такой формат:

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
```

Здесь следует отметить, что компьютеры с архитектурой VLIW являются, вообще говоря, *специализированными* ЭВМ. Специализированные ЭВМ, в отличие от универсальных, могут эффективно использоваться только в достаточно узкой, как говорят, предметной области. Например, можно понять, что компьютеры с архитектурой VLIW будут наиболее эффективны при проведении научных расчётов в таких областях, как линейная алгебра, газовая динамика, физика твёрдого тела и т.д.

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. Заметим, что этот критерий для первых ЭВМ с их маленькой памятью был более важным, чем, например, удобство программирования на языке машины. Исследуем этот вопрос, составив небольшой фрагмент программы для наших учебных ЭВМ с различной адресностью. В качестве примера рассмотрим реализацию оператора присваивания, который содержит типичный набор арифметических 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 используемых величин (целые или вещественные), предположим, что это тоже не влияет на размер программы. Вам необходимо внимательно просмотреть текст этих программ.

- **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 этой программы в байтах: (3 команды)*10 байт = 30 байт.

---

[1] На первый взгляд может показаться, что в стековых ЭВМ нарушается принцип Фон Неймана однородности памяти, так как в стеке возможен доступ только к его вершине. Однако это не так: для большинства стековых ЭВМ сам стек является частью основной памяти, поэтому к любой ячейке стека возможен и прямой доступ, т.е. команды ВСТЕК и ИЗСТЕКА в качестве своих операндов A1 могут указывать адрес произвольной ячейки памяти, в том числе и в любом месте стека.

[2] Заметим, что, несмотря на экзотичность такой архитектуры, она может удовлетворять всем принципам Фон Неймана.

- **Двухадресная машина.**

ПЕР R A	R := a; R - рабочая переменая
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 этой программы: (5 команд)*7 байт = 35 байт (плюс одна дополнительная рабочая переменная R).

- **Одноадресная машина.**

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

Длина этой программы: (8 команд)*4 байта = 32 байта.[1]

- **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 два числа: $(a+b)^2$,a
ОБМЕН	Поменять местами два верхних элемента стека
ДЕЛ	В стеке одно число: $a/(a+b)^2$
ИЗСТЕКА x	Запись результата из стека в x

В данной программе использовались команды разной длины: 3 одноадресные для обмена со стеком и 6 безадресных команд. Как видно из этого примера, в безадресной ЭВМ есть и свои специфические "стековые" команды (дублировать вершину стека, поменять местами две верхние ячейки стека и др.). Длина всей программы: (3 команды)*4 байта + (6 команд)*1 байт = 18 байт.

Видно, что с уменьшением количества адресов в команде увеличивается число команд программы, зато каждая команда становится более короткой. Наше небольшое исследование показало, что архитектура ЭВМ с безадресными командами даёт наиболее компактные программы, кроме того, центральный процессор у них устроен проще, чем у двух и трёхадресных ЭВМ. Именно поэтому в начале развития вычислительной техники такие компьютеры были весьма распространены, их, в частности, выпускала известная фирма Барроуз (Burroughs) [3]. Однако, в дальнейшем были предложены ЭВМ с другой архитектурой, которая позволила писать не менее компактные машинные программы, и при этом обладала дополнительными достоинства, поэтому в настоящее время такие стековые ЭВМ практически не используются.

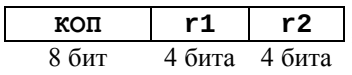
[1] Как видим, длина программы для одноадресной ЭВМ получилась примерно такая же, как и для трёх и двухадресных машин. В то же время легко понять, что центральный процессор одноадресной ЭВМ будет устроен проще, так как проще схема выполнения каждой команды, требуется меньше служебных регистров и т.д. Далее, простота центрального процессора позволяет, при тех же затратах, увеличить скорость его работы за счёт реализации на более дорогих схемах. Отсюда понятна привлекательность одноадресной архитектуры для разработчиков ЭВМ первых поколений. Например, одноадресная отечественная ЭВМ БЭСМ-6 в конце 60-х годов прошлого века была одной из самых быстродействующих машин в мире [3].

4.3. Дробно-адресная архитектура

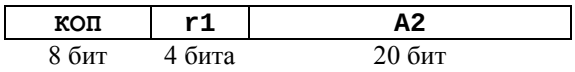
Далее мы рассмотрим архитектуру ЭВМ, которые называются компьютерами с адресуемыми регистрами, в русскоязычной литературе они часто называются *дробно-адресными* [3,4] (смысл этого названия мы скоро выясним). Эти компьютеры должны давать возможность писать такие же компактные программы, как и компьютеры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

Компьютеры дробно-адресной архитектуры нарушают принцип Фон Неймана линейности и однородности памяти. В этих компьютерах память, к которой может непосредственно (по адресам) обращаться центральный процессор за операндами команд, состоит из двух частей, каждая со своей независимой нумерацией ячеек (это и есть нарушение линейности памяти). 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 большой основной памяти.

Например, рассмотрим учебную двухадресную ЭВМ этой новой архитектуры, которую назовём УМ-Р (учебная машина с адресуемыми регистрами). Пусть регистровая память этой учебной ЭВМ состоит из 16 ячеек. В этом случае адрес каждого регистра лежит в диапазоне `0÷15,` и будет помещаться в 4 бита, а основная память пусть содержит 2^{20} ячеек, тогда адрес каждой ячейки занимает `20` двоичных разрядов. В такой ЭВМ в качестве адресов операндов каждой команды могут быть или адреса двух регистров, или один адрес регистра и один адрес ячейки основной памяти.[1] Адреса регистров на схемах команд будем обозначать `r1` и `r2`, а адреса основной памяти, как и раньше, `A1` или `A2`. Первый вид команд будем называть командами формата *регистр-регистр* (обозначается RR), а второй – формата *регистр-память* (обозначается RX).[2] В этом случае для одного кода операции (например, сложения) мы получим команды двух форматов длины 2 и 4 байта соответственно:



= 2 байта



= 4 байта

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, такое расположение обеспечивает быстрый доступ к хранимым на регистрах данным (не требуется делать обмен с расположенной отдельно от центрального процессора основной памятью). Как правило, время доступа к регистру у всех ЭВМ примерно в 10 раз меньше, чем к ячейке основной памяти (это и есть нарушение принципа однородности памяти). Всё это позволяет существенно повысить быстродействие такого компьютера.

Заметим, что, кроме *адресуемых* регистров, номера которых явно указываются в команде, в арифметико-логическом устройстве по-прежнему могут быть и не адресуемые регистры, например, уже знакомые нам регистры первого и второго операнда `R1` и `R2`, а также регистр сумматора `S`. Таким образом, команда формата регистр-память `КОП r1,A2` может выполняться, например, по схеме

```
R2 := <A2>; S := r1⊗R2; r1 := S
```

Отметим, что содержимое регистров, в отличие от содержимого ячеек основной памяти, принято записывать без угловых скобок, так что, например, запись `r1` в приведённом выше примере обозначает содержимое регистра `r1`, а не номер (адрес) этого регистра.

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 4/20=1/5.

---

[1] Заметим, что *команды* устройство управления по-прежнему может читать только из основной памяти.

[2] Мы вскоре выясним, что третий логически допустимый формат команд (память-память), когда оба адреса принадлежат основной памяти, не даёт никаких преимуществ при программировании в этой архитектуре. Исходя из этого мы не будем его реализовывать в нашей дробноадресной учебной машине.

Из сказанного 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, этого принципа мы и будем всегда придерживаться в наших примерах программ. 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, а r1 и r2 обозначают номера адресуемых регистров. Точка с запятой, как это принято в языке Ассемблера, к изучению которого мы вскоре приступим, задаёт начало *комментария* 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
        ...
```

В этом примере r1 и r2 – это номера (т.е. адреса) каких-либо двух разных адресуемых регистров нашей машины. Длина этого фрагмента программы равна

(3 команды)*4 байта + (3 команды)*2 байта = 18 байт.

Как видим, данная архитектура, обладая отмеченными выше преимуществами, не уступает стековой (безадресной) архитектуре по компактности получаемых программ.

Рассмотрим теперь главный недостаток дробно-адресной архитектуры ЭВМ. Заметим, что 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 центрального процессора, который отныне должен поддерживать бóльшее количество операций. Однако преимущества дробно-адресной архитектуры настолько очевидны, что её имеют большинство современных машин.

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.

Разумеется, с развитием вычислительной техники появилось и много новых особенностей, некоторые из которых мы рассмотрим далее в нашем курсе.

4.4. Способы адресации

Перейдём теперь к изучению следующего важного в архитектуре ЭВМ понятия. 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 усвоения этого очень важного понятия модифицируем описанную нами ранее одноадресную учебную ЭВМ УМ-1, введя в её язык команды сложения целых чисел с разными способами адресации. Мнемоника кодов операций сложения будет указывать на способ адресации.

- **Прямой способ адресации.**

СЛ	2	S := S + <2>

При этом способе адресации (заметим, что только этот способ мы с Вами использовали до сих пор во всех учебных машинах) число на месте операнда задаёт *адрес* ячейки основной памяти, в котором и содержится необходимый в команде операнд. Мы будем, как обычно 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, хранится не *число* 2.

- **Непосредственный способ адресации.**

СЛН	2	`S := S + 2`

При таком способе адресации поле адреса команды содержит, как говорят, *непосредственный* операнд. Таким образом, число 2 в нашем примере обозначает не ячейку памяти с адресом 2, а непосредственно целочисленное значение 2. Разумеется, такие непосредственные операнды могут быть только (неотрицательными) целыми числами, по величине не превышающими максимального значения, которое можно записать в поле адреса. Использование непосредственного метода адресации позволяет не располагать (целочисленные) константы в ячейках памяти, а помещать их внутрь команд, (на место адреса операнда), что может сильно сэкономить память.[1]

- **Косвенный способ адресации.**

СЛК	2	`S := S + <<2>>`

Здесь число на месте операнда задаёт *адрес* ячейки памяти, содержимое которой, в свою очередь, трактуется как целое число – адрес необходимого операнда в памяти ЭВМ. Таким образом, число 2 в нашем примере является *косвенным адресом* операнда. При таком способе адресации для доступа к операнду центральному процессору необходимо дважды обратиться к основной памяти.

В качестве примера, для лучшего усвоения этого нового понятия, выполним несколько команд сложения с различными способами адресации для некоторой гипотетической учебной одноадресной ЭВМ, и рассмотрим значение регистра сумматора S после выполнения этих команд (см. рис. 4.1). В комментариях к каждой команде показаны производимые этой командой действия. Справа на этом рисунке показаны первые ячейки памяти и хранимые в них целые числа.

```
          . . .
СЧ   0;  S:=<0>=0
СЛ   2;  S:=S+<2>=3
СЛН  2;  S:=S+2=5
СЛК  2;  S:=S+<<2>>=13
          . . .
```

Адрес	Значение
000	0
001	2
002	3
003	8
. . .	. . .

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

4.5. Многообразие форматов данных

Современные ЭВМ позволяют совершать операции над целыми и вещественными числами разной длины. Это вызвано чисто практическими соображениями. Например, если нужное нам целое число помещается в один байт, то неэкономно использовать под его хранение два или большее число байтов памяти (экономия будет особенно заметной, если необходимо хранить большой массив таких чисел). Во избежание такого неоправданного расхода памяти введены соответствующие *форматы данных*, отражающие представление в памяти ЭВМ чисел разной длины. Например, в зависимости от размера целого числа, оно может занимать в памяти 1, 2, 4 и более байт. Приведённая ниже таблица иллюстрирует *многообразие форматов данных*, для представления целых чисел на той машине, где Вы сейчас работаете.

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

Многообразие форматов данных требует усложнения архитектуры как устройства управления (резко возрастает число команд в языке машины), так и арифметико-логического устройства, в частности, регистровой памяти. Теперь регистры должны уметь хранить, а само арифметико-логическое устройство – обрабатывать данные *разной длины*.

---

[1] Заметим, что это же позволяет лучше защитить константы от случайной порчи при ошибочной записи в те ячейки памяти, где расположены константы. Это, разумеется, повышает надёжность программирования на таких ЭВМ. Само программирование на языке машины также упрощается, так как теперь не надо производить распределение памяти под хранение таких констант.

4.6. Форматы команд

Для операций с разными способами адресации и разными форматами данных необходимо введение различных *форматов команд*, которые по-разному задают местонахождение операндов, и, естественно, имеют разную длину. Для широко распространённых сейчас двухадресных ЭВМ 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).[1]

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

Возвращаясь к нашей учебной ЭВМ УМ-3 теперь можно, используя новые изученные понятия, сказать, что это трёхадресная машина с прямым способом адресации, одним форматом команд и одним форматом данных.

4.7. Базирование адресов

Для дальнейшего уменьшения объёма программы современные ЭВМ используют новый способ адресации, основанный на принципе *базирования адресов*. Изучение этого нового для нас и очень важного понятия проведём на следующем примере. Пусть в программе на одноадресной учебной машине УМ-1 необходимо реализовать арифметический оператор присваивания $Z:=(X+Y)^2$. Ниже приведена эта часть программы с соответствующими комментариями (напомним, что S – это регистр сумматора одноадресной ЭВМ):

```
    ...
СЧ X; S:=X
СЛ Y; S:=X+Y
ЗП R; R:=X+Y - запись в рабочую переменную
УМ R; S:=(X+Y)²
ЗП Z; Z:=(X+Y)²
    ...
```

Так как в нашей одноадресной учебной ЭВМ имеется 2^{24} (примерно 16 миллионов) ячеек памяти, то будем, не теряя общности, считать, что наш фрагмент программы располагается где-то примерно в средине памяти. Пусть, например, наши переменные при распределении памяти оказались помещены соответственно в следующих ячейках памяти (как и ранее, адреса для удобства даны в десятичной системе счисления):

```
X – в ячейке с адресом 10 000 000
Y – в ячейке с адресом 10 000 001
Z – в ячейке с адресом 10 000 002
R – в ячейке с адресом 10 000 003
```

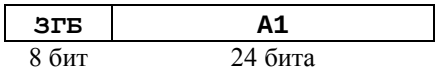
Тогда приведённый выше фрагмент программы после замены мнемонических обозначений переменных их адресами будут выглядеть следующим образом:

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

Из этого примера видно, что большинство адресов в нашей программе можно представить в виде $B+\Delta$, где число B назовём *базовым адресом* программы или просто *базой* (в нашем случае $B=10\ 000\ 000$), а число Δ – *смещением* адреса относительно этой базы. Здесь налицо существен-

[1] Использования для мнемонического обозгачения операнда в памяти сразу двух букв X и S связано с особенностями машинных форматов этих команд, пока надо не обращать на это внимания и просто запомнить.

ная *избыточность* информации в программе. 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 нашей одноадресной ЭВМ команду *загрузки базы* (длина этой команды 4 байта):

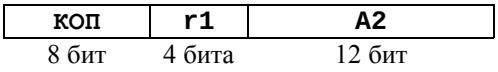


Тогда наш фрагмент программы будет иметь такой вид:

```
    ...
ЗГБ 10000000
    ...
СЧ  000; S:=X
СЛ  001; S:=X+Y
ЗП  003; R:=X+Y
УМ  003; S:=(X+Y)^2
ЗП  002; Z:=(X+Y)^2
    ...
```

Как видим, в большинстве команд можно теперь вместо длинного *адреса* ячейки памяти указывать только короткое *смещение* Δ этой ячейки относительно базы. Это позволит значительно уменьшить размер программы. Заметим, однако, что теперь при выполнении *каждого* обращения за операндом в основную память, центральный процессор должен *вычислять* значение адреса этого операнда по формуле A=B+Δ. Это вычисление производится в устройстве управления и, естественно, усложняет его, не говоря уже о том, что и выполнение всей команды замедляется. Например, адрес переменной X вычисляется как Адрес(X)=10000001=B+Δ=10^7+1.

Осталось выбрать оптимальную длину поля смещения Δ в поле адреса команды. Для этого вернёмся к рассмотрению учебной дробно-адресной ЭВМ, для которой теперь будет реализовано базирование адресов основной памяти (не регистровой, там это не нужно, так как адреса регистров и так маленькие). Например, пусть под запись смещения мы 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* Δ относительно базы. Тогда все команды, которые обращаются за операндом в основную память, будут в нашей дробно-адресной ЭВМ более короткими:



Рассмотрим схему выполнения такой команды для формата регистр-память (⊗ как обычно задаёт какой-то код бинарной операции):

r1 := r1 ⊗ <B+A2>

или для формата память-регистр:

<B+A2> := <B+A2> ⊗ r1

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 разбиения памяти на такие участки – *сегментированием* памяти.

Сегментирование позволяет значительно уменьшить объём памяти для хранения программ, но оно имеет и один существенный недостаток: теперь каждая команда может обращаться не к любой ячейке оперативной памяти, как это было у нас раньше, а только к тем из ячеек, до которых "дотягивается" смещение относительно своей базы. В нашем предыдущем 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). Такой недостаток, например, существенно затрудняет работу с массивами длиной больше, чем 2^{12} ячеек. Однако, несмотря на указанный недостаток, практически все современные ЭВМ в целях уменьшения объёма программы производят сегментирование памяти и базирование адресов команд для уменьшения длины программы.

Заметим также, что отмеченный выше недостаток в большинстве архитектур современных ЭВМ исправляется путём реализации нескольких сегментных регистров, а также *переменной* длины смещения (например, разрешается смещение длиной в 1, 2 или 4 байта). Это, однако, ещё более увеличивает набор команд языка машины и усложняет центральный процессор.

Итак, ещё раз отметим очень важное обстоятельство нового способа адресации. В новой архитектуре для осуществления любого доступа к памяти ЭВМ необходимо, чтобы ячейка, к которой осуществляется доступ, находилась в сегменте, на начало которого указывает некоторый сегментный регистр. Как уже говорилось, современные ЭВМ обеспечивают одновременную работу с несколькими сегментами памяти и, соответственно, имеют несколько сегментных регистров.

В некоторых архитектурах адресуемые 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, поэтому во многих архитектурах используются *специализация* регистров, т.е. определённые регистры являются только сегментными (на них нельзя, например, складывать числа), на других могут производиться операции, третьи используются в качестве счётчиков циклов и т.д. Именно так будет обстоять дело в той конкретной ЭВМ, которую мы будем вскоре изучать.

Упражнение. Добавьте в язык учебной машины УМ-3 новую команду *косвенной* пересылки:

ПЕРК A1,A2,A3

Эта команда использует косвенную адресацию по своему третьему адресу и выполняется по правилу:

<A1> := <<A3>>

Покажите, что с помощью этой команды можно обрабатывать массивы без использования самомодифицирующихся программ. Переделайте для этого программу суммирования всех элементов массива из примера на рис. 3.4.

Глава 5. Понятие семейства ЭВМ

Компьютеры могут применяться в самых различных областях человеческой деятельности (как уже говорилось, 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 сложными устройствами (ракетами, доменными печами и др.)

Мы уже упоминали, что компьютеры, архитектура которых в основном 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. В нашем курсе мы 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). Таким образом, пользователь может выбирать между дешёвыми моделями с относительно небольшими аппаратными возможностями, и более дорогими моделями с большей производительностью.
2. Все модели семейства обладают *программной совместимостью*:
   1) снизу-вверх – старшие модели поддерживают все команды младших (любая программа, написанная для младшей модели, безошибочно выполняется и на старшей модели). Это свойство называется ещё *обратной совместимостью*;
   2) сверху-вниз – на младших моделях выполняются программы, написанные для старших, если выполнены следующие 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 должно работать на всех выпускаемых в настоящее время моделях семейства).
4. Модели семейства организованы по принципу *модульности*, что позволяет в определённых пределах расширять возможности ЭВМ, увеличивая, например, объём памяти или повышая производительность путём замены центрального процессора более быстродействующим.
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.[1] В нашем курсе для упрощения изложения будут рассматриваться в основном младшие модели семейства ЭВМ компании Intel. Соответственно все наши примеры программ должны правильно выполняться для всех моделей этого семейства, поэтому мы ограничимся лишь архитектурой и системой команд самой младшей модели этого семейства [9].

Одной из главных особенностей семейства ЭВМ следует считать программную совместимость, которая позволяет гарантировать, что все разработанные ранее программы будут правильно и без переделок выполняться и на всех последующих моделях ЭВМ этого семейства. Это требования должно безукоснительно соблюдаться по чисто экономическим соображениям, так как стоимость уже разработанного программного обеспечения более, чем на порядок, превышает стоимость всего аппаратного обеспечения. В то же время требования учитывать в новых моделях семейства все те устаревшие

[1] Вообще говоря, можно ещё потребовать, чтобы все модели семейства выпускались одной фирмой. В то же всемя часто случается, что некоторая другая фирма начинает выпускать своё семейство ЭВМ, *программно совместимое* с уже выпускающимся семейством. В качестве примера можно привести семейство ЭВМ фирмы Intel и семейство ЭВМ фирмы AMD. Важно понять, что такие семейства ЭВМ одинаковы на *внутреннем* уровне в*и*дения архитектуры (при программировании на языках низкого уровня), но их архитектура различна ни *инженерном* уровне.

архитектурные решения, которые были приняты ранее, становится всё более тягостной и трудноразрешимой задачей.

Ясно, что такое положение вещей не сможет долго продолжаться, и рано или поздно от принципа программной совместимости придётся отказаться. Новые модели необходимо строить по самым современным архитектурным схемам, учитывающим глубокий параллелизм в обработке данных. В то же время, нельзя и потерять возможность выполнять старые программы для предшествующих моделей семейства.

Очевидно, эту проблему будут решать следующим способом. Новые процессоры будут иметь совершенно другую архитектуру и, следовательно, другую систему команд, однако предусмотрена их работа в двух режимах. В основном режиме процессор может выполнять команды только своего нового машинного языка, однако во вспомогательном режиме он имеет возможность аппаратно *интерпретировать* (полностью имитировать выполнение) программ на языке машины предыдущих моделей семейства. Разумеется, интерпретация значительно (в несколько раз) снижает скорость выполнения старых программ. Основную надежду здесь возлагают на то, что старые программы, написанные на языках *высокого уровня*, могут быть достаточно легко исправлены так, чтобы быть заново откомпилированы уже на новый машинный язык. Кроме того, возможность значительно ускорить выполнение своих программ, перейдя на новую архитектуру, должна быть хорошим стимулом для программистов. Ну, а всем остальным "не передовым" пользователям можно гарантировать, что все их старые программы на новых моделях будут считаться почти так же быстро, как и на старых (за счёт повышения вычислительной мощности новых процессоров).

Описанный выше подход сейчас начинает претворяться в жизнь. Уже в 2001-03 годах компанией Intel был выпущен первый 64-х разрядный процессор, получивший название Itanium. В новом 64-битном режиме он имеет новую передовую архитектуру и совершенно другую систему команд, рассчитанную на задание явного параллелизма в программах на машинном языке.[1] В то же время в "старом" 32-битном режиме он полностью интерпретирует машинный язык прежних моделей нашего семейства.[2]

---

[1] Эта архитектура называется EPIC, она является дальнейшим развитием архитектуры компьютеров VLIW (с очень длинным командным словом), о которой мы немного рассказывали в начале этой книги.

[2] Вообще говоря, процесс эмуляции старых моделей семейства на новых (правда, без смены машинного языка) практикуется уже давно. Например, когда Вы выполняете свои программы на Ассемблере, рассчитанном на младшую модель нашего семейства, операционная система старшей модели выделяет для этого так называемую *виртуальную машину*, которая полностью эмулирует для нас стиль работы на ЭВМ младших моделей. Именно поэтому в наших ассемблерных программах мы, например, можем менять вектор прерывания. На новых моделях это можно сделать только в так называемом привилегированном (защищённом) режиме, о чём мы будем говорить в главе, посвящённой мультипрограммированию.

Глава 6. Архитектура младшей модели семейства Intel

Сейчас мы переходим к изучению архитектуры младшей модели того семейства компьютеров, на которых Вы выполняете свои практические задания. Нами будут последовательно рассмотрены устройство памяти, форматы обрабатываемых данных и работа центрального процессора этой ЭВМ.

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, при этом каждая команда или данные располагаются в одной или нескольких последовательных (с возрастающими адресами) ячейках этой памяти. Устройство регистровой памяти будет рассмотрена немного позже.

6.2. Форматы данных

- **Целые числа.**

Целые числа могут занимать в памяти 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, там это содержимое ячейки памяти!).

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 обычно производится работа с малым количеством целых чисел, поэтому вводится только один формат (например, 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 машине и языке Ассемблера отсутствует. И пусть Вас не вводит в заблуждение запись `'A'` в языке Ассемблера, она обозначает не константу символьного типа данных, а является *целочисленной* константой и просто эквивалентна выражению `Ord('A')` языка Паскаль.

- **Массивы (строки).**

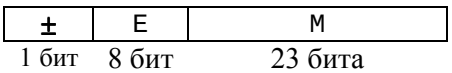
Допускаются только одномерные массивы, которые могут состоять из коротких или длинных целых чисел. Массив коротких целых чисел может рассматриваться программистом как *символьная строка*. В машинном языке присутствуют команды для обработки *элементов* таких массивов, если такую команду поставить в цикл,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

- **Вещественные числа.**

На современных ЭВМ чаще всего используются три формата вещественных чисел: короткие (длиной 4 байта), длинные (8 байт) и сверхдлинные (16 байт) вещественные числа. Стоит отметить следующий важный факт. В то время как целые числа в различных ЭВМ по чисто историческим причинам часто имеют разное внутреннее представление, то на момент массового выпуска ЭВМ с командами для работы с вещественными числами, уже существовал международный стандарт на внутреннее представление этих чисел (ANSI/IEEE standart 754-1985), и почти все современные машины придерживаются этого стандарта на представление вещественных чисел.

6.3. Вещественные числа

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



[1] В настоящее время существуют алфавиты (например, алфавит Unicode), 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, разбивается на два поля: *машинный порядок* E и *мантиссу* M. Мантисса задаёт двоичное число, значение которого по модулю считается меньше единицы, другими словами, это число можно записать как 0.M. И вот теперь каждое представимое в этом формате вещественное число A (кроме вещественного нуля 0.0) может быть записано в виде произведения двух сомножителей: $A=\pm 1.M*2^{E-127}$. Такое представление вещественного числа называется *нормализованным*: его первый сомножитель удовлетворяет неравенству:

$$1.0 \le 1.M < 2.0$$ [1]

Нормализация необходимо для однозначного представления ненулевого вещественного числа в виде двух сомножителей. Нулевое же число представляется нулями во всех позициях, за исключением, быть может, первой позиции знака числа (при этом числа -0.0 и +0.0 считаются равными).

В качестве примера переведём десятичное число –13.25 во внутреннее машинное представление. Для этого сначала переведём 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 нашего числа будет иметь вид $10101000000000000000000_2$, и осталось вычислить машинный порядок E: $3 = E-127;\ E = 130 = 128+2 = 10000001_2$. Теперь, учитывая знак, получаем вид внутреннего машинного представления числа -13.25_{10} (мы запишем его в виде двоичного и, как это часто делается для компактной записи, шестнадцатеричного значения):

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

Шестнадцатеричные числа в языке Ассемблера принято записывать с буквой h на конце, при этом, если такое число начинается с буквы, то впереди записывается незначащий ноль, чтобы отличить запись такого числа от *имени*:

$$C1500000_{16} = 0C1500000h$$

Таков формат короткого вещественного числа. Исходя из этого формата, машинный порядок E изменяется от 0 до 255, однако, как мы увидим далее, значения машинного порядка 0 и 255 зарезервированы сля специальных целей, следовательно, представимый диапазон порядков коротких вещественных чисел равен $2^{-126}..2^{127} \approx 10^{-38}..10^{38}$.

Как и для целых чисел, машинное представление которых мы рассмотрим чуть позже, число представимых вещественных чисел *конечно*. Действительно, легко понять, что таких чисел не больше, чем 2^{32}, а на самом деле, как мы вскоре увидим, даже несколько меньше. 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 число (и наоборот).

Заметим, что из-за конечной длины представления вещественных чисел, действия с ними выдают приближённый результат. Одним из следствий этого является нарушение при работе с машинными вещественными числами ассоциативного и дистрибутивного законов арифметики. Другими словами, возможны случаи, когда (a+b)+c ≠ a+(b+c) и (a+b)*c ≠ a*c+b*c. Чтобы показать, насколько привычная нам арифметика отличается от машинной, рассмотрим решение простейшего уравнения X+A=A. Естественно, что в обычной математика у такого уравнения для любого значения величины A существует только один корень X=0, однако при решении этой задачи на компьютере можно получить и ненулевые корни такого уравнения! И не следует думать, что такие "неправильные" корни будут какими-нибудь маленькими числами, скажем 10^{-6}, нет, вполне возможен случай, когда корень такого уравнения X будет равен, скажем, 10^{+6}.

Упражнение. Используя какой-нибудь язык программирования высокого уровня (скажем, Паскаль) получите такое значение константы A, чтобы для корня $X=10^{-6}$ выполнялось машинное равенство X+A=A.

[1] Мантисса вместе с единичным битом перед точкой имеет в английском языке специальное название significand.

Как уже упоминалось выше, некоторые комбинации нулей и единиц в памяти, отведённой под хранения вещественного числа, используются для служебных целей. В частности, значение машинного порядка E=255 при мантиссе M ≠ 0 обозначает специальное значение "*не число*" (NAN – not a number). При попытке производить арифметические операции над такими "числами" в арифметико-логическом устройстве возникает аварийная ситуация. Например, значение "не число" может быть присвоено программистом вещественной переменной после её порождении, если эта переменная не имеет "настоящего" начального значения (как говорят, *не инициализирована*). Такой приём позволяет избежать тяжёлых семантических ошибок, возникающих при работе с неинициализированными переменными, которые при порождении, как правило, имеют случайные значения.

Отметим ещё одну специальную комбинацию нулей и единиц в представлении вещественных чисел: машинный порядок E=255 при мантиссе M = 0 задаёт, в зависимости от знака числа, специальные значения ±∞. Эти значения выдаются в качестве результата арифметических операций с вещественными числами, 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

$A \pm\infty = \pm\infty;\ A * \pm\infty = \pm\infty\ \pm\infty/\pm\infty = 0/0 = NAN$ и т.д.

Отметим, что при этом правильно учитывается знак числа, например, $(-8.0)*(-\infty)= +\infty$. Для любознательных студентов заметим, что существует и нетрадиционное построение математического анализа, в котором, как и в нашей ЭВМ, бесконечно большие величины ±∞ (а также бесконечно малые величины ±ε) определяются не в виде пределов, как в обычном, привычном нам анализе, а существуют в виде "настоящих" вещественных чисел.[1]

Разберём теперь, что происходит, если после выполнения операции над вещественными числами получился такой результат, который, хоть и не равен нулю, но не может быть представлен в виде *нормализованного* числа. Другими словами, для случая рассмотренного нами формате представления вещественных чисел длиной в четыре байта, этот результат по модулю меньше $1.0*2^{-126}$., т.е. должен иметь нулевой машинный порядок. В этом случае центральный процессор пытается представить этот результат уже как *ненормализованное* число, т.е. в виде $0.M*2^{-127}$, и только в том случае, если результат и для такого представления слишком мал по модулю, выдаёт в качестве ответа машинный ноль.[2] Использование для таких маленьких по модулю чисел ненормализованного представления позволяет расширить диапазон машинных чисел. Можно показать, что если использовать нулевой машинный порядок для представления нормализованных чисел, то самое маленькое число будет $1.0*2^{-127}$. В то же время, если зарезервировать нулевой машинный порядок для представления ненормализованных чисел, но наименьшее машинное вещественное число будет равно

$$0.M_{min}*2^{-127}=0.00000000000000000000001*2^{-127}=2^{-150}.$$

Такое использование ненормализованных вещественных чисел позволяет более "мягко" приближаться к нулю при проведении вычислений с вещественными числами, что часто уменьшает погрешность таких вычислений.[3] В заключение рассмотрения машинного представления вещественных чисел отметим, что при изучении архитектуры ЭВМ они не будут представлять для нас большого интереса, и поэтому (а также, в основном, из-за недостатка времени) операции над вещественными числами мы изучать не будем.

6.4. Целые числа

Мы уже знаем, что хранимые в памяти машинные слова (наборы битов) могут трактоваться поразному. При вызове в устройство управления этот набор битов трактуется как команда, а при вызове

---

[1] С изложением нетрадиционного математического анализа можно, например, ознакомиться по книгам [13,14].

[2] Вообще говоря, во многих ЭВМ существует возможность задать режимы работы центрального процессора с вещественными числами, при которых выработка в качестве результата операции таких особых значений, как ненормализованное число или ±∞ будет вызывать аварийную ситуацию (сигнал прерывания). Эти аварийные ситуации носят названия "потеря значимости" (flowing underflow) и "вещественное переполнение" (flowing overflow) соответственно.

[3] Использование ненормализованных чисел имеет существенный недостаток, оно влечёт за собой уменьшение *точности* представления таких чисел, т.е. в этих числах меньшее число *значащих* цифр.

в арифметико-логическое устройство – как число. В дополнении к этому в рассматриваемой нами архитектуре каждое хранимое целое число может трактоваться программистом как *знаковое* или *беззнаковое* (неотрицательное). По внешнему виду *невозможно* определить, какое число храниться в определённом месте памяти, только сам программист может знать, как он *рассматривает* это число (вспомним соответствующий принцип Фон Неймана). Таким образом, определены две различные машинные *системы счисления* для представления знаковых и беззнаковых целых 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 формате одного байта, 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* (complementary) код, который можно получить из обратного кода прибавлением единицы. Например, получим дополнительный код числа `–13`:

```
Прямой код            00001101
Обратный код          11110010
                    +        1
                      ---------
Дополнительный код    11110011
```

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 целого числа (в нашем предыдущем примере целое число имеет длину один байт и `N=8`). Таким образом, дополнительный код числа `–13` можно вычислить и так:

$$2^8-13 = 256-13 = 100000000-00001101 = 11110011$$

Отметим интересное свойство дополнительного кода: если сложить его с прямым кодом, то получится ноль и "лишняя " единица, не помещающаяся в отводимое число разрядов (именно поэтому этот код и называется *дополнительным*). Возвращаясь к представлению числа `–13`, имеем:

```
Дополнительный код    11110011
                    +
Прямой код              00001101
                      ---------
                      100000000
```

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

Заметим также, что 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 по абсолютно одинаковым алгоритмам, что и 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 в центральном процессоре нашей ЭВМ двух чисел длиной в один байт. В первом столбике будет записано внутреннее двоичное представление чисел, а во втором и третьем – беззнаковое и знаковое представления этих же чисел в привычной для нас десятичной системе счисления.

- **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 (1 вместо правильной суммы 257). Это произошло потому, что при сложении мы получим девятизначное двоичное число, "не умещающееся" в один байт, поэтому левый бит пришлось отбросить.[1] Так как центральный процессор "не знает", как программист будет трактовать складываемые числа, то он "на всякий случай" будет сигнализировать о том, что при сложении беззнаковых чисел произошла ошибка.

Для обозначения таких (и некоторых других) ситуаций в архитектуре нашего компьютера введено понятие *флагов*. Каждый флаг занимает один бит в специальном *регистре флагов* с именем FLAGS. Для рассмотренного выше примера флаг CF (carry flag) после сложения примет значение, равное единице (иногда говорят, что флаг *поднят*), сигнализируя программисту о том, что при беззнаковом сложении произошла ошибка. Рассматривая результат нашего примера в знаковых числах, мы получили *правильный* ответ, поэтому соответствующий флаг результата знакового сложения OF (overflow flag) будет положен равным нулю (или, как говорят, *опущен*). Флаг CF называется *флагом переноса*, а OF – *флагом переполнения*.

- **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 после сложения соответственно значения 0 (флаг опущен, ошибки нет) и 1 (флаг поднят, была ошибка).

- **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, поэтому формируется содержимое флагов: CF = OF = 1. Легко придумать пример, когда результат сложения будет правильный как для знаковых, так и для беззнаковых чисел (сделайте это самостоятельно!), после такого сложения оба флага будут опущены.

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 всегда заносится знаковый (крайний левый) бит результата, таким образом, при знаковой трактовке чисел этот флаг сигнализирует, что результат получился отрицательным.[2] Кроме того, при программировании часто представляет интерес и флаг ZF, который устанавливается в 1, если результат тождественно равен нулю, в противном случае этот флаг устанавливается в 0. Заметим, что флаги в нашей архитектуре выполняют ту же роль, что и регистр признака результата ω в изученной ранее учебной ЭВМ УМ-3.

Представление отрицательных чисел в дополнительном коде неудобно для программистов, однако, позволяет существенно упростить арифметико-логическое устройство. Другими словами, удобство программирования было принесено в жертву простоте реализации центрального процессора. Заметим, например, что вычитание при этом представлении можно выполнять как сложение с дополнительным кодом числа.

Основная причина использования двух систем счисления для представления целых чисел заключается в том, что при *одновременном* использовании в программе *обеих* систем счисления диапазон представимых целых чисел увеличивается в полтора раза. Это было весьма существенно для первых ЭВМ с их весьма небольшим объёмом памяти. Сейчас это уже не имеет такого большого значения при программировании, однако, мы не можем отказаться от этих двух систем счисления для пред-

[1] В этом примере при сложении знаковых чисем мы тоже отбрасываем левый бит результата, однако здесь отбрасываемая двоичная единица является *незначащей* двоичной цифрой суммы, и её можно отбросить без изменения значения этой *отрицательной* суммы.

[2] При этом флаг переполнения OF обязательно должнен быть нулевым, иначе очевидно, что анализировать флаг SF не имеет смысла, так как правильный результат не получен.

ставления целых чисел из-за принципа программной совместимости старших моделей нашего семейства ЭВМ с младшими, несмотря на то, что эти младшие модели уже давно не выпускаются.[1]

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 больше). Это означает, что есть четыре *сегментных* регистра, которые указывают на определённые области памяти. Каждый сегментный регистр имеет длину 16 разрядов, а в то же время для адресации любого места нашей памяти необходимо, как мы уже говорили, 20 разрядов. Для того чтобы сегмент мог располагаться на любом месте оперативной памяти, адрес начала сегмента получается после умножения значения сегментного регистра на число 16. Правда, как легко понять, при таком способе задания начала сегмента, он может начинаться не с любого места оперативной памяти, а только с адресов, кратных 16 (в некоторых книгах по Ассемблеру такие участки памяти называются *параграфами*).

Итак, производить обмен с памятью можно только относительно одного из них этих сегментных регистров. Таким образом, физический адрес числа или команды вычисляется центральным процессором по формуле

$$A_{физ} := (SEG*16 + A)_{mod\ 2^{20}},$$

где SEG – значение сегментного регистра, а A –заданное в команде *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 *служебные* имена:[2] кодовый сегментный регистр (CS), сегментный регистр данных (DS), сегментный регистр стека (SS) и дополнительный сегментный регистр (ES). Смысл этих названий скоро прояснится. Каждый из этих регистров может адресовать сегмент памяти длиной от 1 до 2^{16} байт (напомним, что вся память состоит из 2^{20} ячеек), при этом "настоящая" длина сегмента известна только программисту. Так как физический адрес в приведённой выше формуле берётся по модулю 2^{20}, то, очевидно, что память, как и в нашей учебной ЭВМ, "замкнута в кольцо". Другими словами, 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.

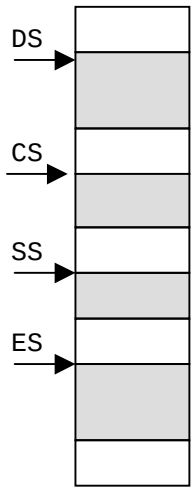


Рис. 6.1. Пример расположения сегментов в памяти.

Стоит отметить, что сегментные регистры являются специализированными, предназначенными только для хранения адресов сегментов, поэтому арифметические операции (сложение, вычитание и др.) 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.

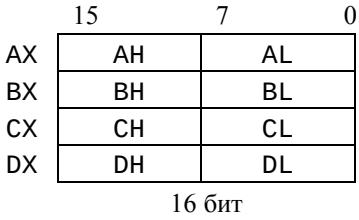
6.6. Мнемонические обозначения регистров

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

---

[1] Для любознательных студентов отметим, что, например, в семейства ЭВМ IBM-360/370, которые начали выпускать свою первую модель позже, чем в нашем семействе, уже принят "естественный" для программиста формат целого числа, где крайний левый бит задаёт знак числа, а затем записывается прямой код модуля.

[2] 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, кроме того, который определён в языке.

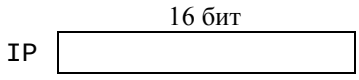


Каждый из регистров AH, AL, BH, BL, CH, CL, DH и DL может быть использован в машинных командах как самостоятельный регистр, на них можно выполнять операции сложения и вычитания. В дальнейшем условное обозначение r8 мы будем использовать для обозначения любого короткого (8-разрядного) адресуемого регистра.

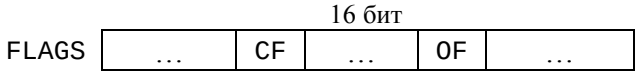
Существуют также четыре регистра с именами SI, DI, SP и BP, которые также могут использоваться для проведения сложения и вычитания, но они уже не делятся на половинки:

В основном эти четыре регистра используются как *индексные*, т.е. на них обычно храниться положение конкретного элемента в некотором массиве. Условное обозначение r16 будем использовать для указания любого из регистров AX, BX, CX, DX, SI, DI, SP и BP.

Кроме перечисленных выше регистров программист имеет дело с регистром IP (instruction pointer), который называется счётчиком адреса (в учебной машине мы обозначали его как RA). Этот регистр содержит адрес следующей исполняемой команды (точнее, содержит *смещение* этой команды относительно начала кодового сегмента, адрес начала этого сегмента равен значению сегментного регистра CS, умноженному на 16).

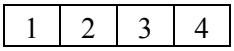


И, наконец, как уже упоминалось, архитектурой изучаемой ЭВМ предусмотрен регистр флагов с именем FLAGS. Он содержит шестнадцать одноразрядных флагов, например, ранее упоминавшиеся флаги CF и OF. Конкретные номера битов, содержащих тот или иной флаг, для понимания архитектуры несущественны, и приводиться здесь не будут. Заметим также, что эти номера нам не надо будет знать и при программировании на языке Ассемблера.



Как уже показывалось на рисунках, все биты в регистрах пронумерованы справа налево: в шестнадцатибитных – от 0 до 15, в восьмибитных – от 0 до 7. Как и в Паскале, в языке Ассемблера принято такое же соглашение по семантике всех имён: регистр символов не различается. Таким образом, имена можно писать как большими, так и маленькими буквами, например, имена AX,Ax,aX и ax обозначают в нашем языке Ассемблера *один и тот же* регистр.

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



Теперь перешлём это число в память в ячейки с адресами, например, 100 и 101. Так вот: в ячейку с адресом 100 при такой пересылке запишется число 34h, а в ячейку с адресом 101 запишется число 12h. Говорят, что целое число представлено в основной памяти (в отличие от регистров) в *перевёрнутом* виде. Это связано с тем, что в младших моделях ЭВМ при каждом обращении к памяти в центральный процессор читался всего один байт. Таким образом, для того, чтобы считать двухбайтное целое число, было необходимо дважды обратиться к памяти, поэтому было удобно (напри-

мер, для проведения операция сложения "в столбик") получать из памяти сначала младшие цифры числа, а затем – старшие. В современной архитектуре за одно обращение из памяти получают сразу 8 или 16 байт, но из-за совместимости моделей семейства пришлось оставить *перевёрнутое* представление чисел, что, конечно, неудобно для программистов. 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, что позволяет задавать до 64 различных операций. Далее следуют однобитные поля с именами d и w, где d – так называемый бит *направления*, а w – бит *размера аргумента*, последующие два бита для этого формата всегда равны 11, а последние две части (по 3 бита каждая) задают номера регистров-операндов команды.

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. В своих программах мы будем пользоваться и другими видами формата регистр-регистр, например, командой `mov ds,ax`, но выписывать их внутреннее представление не будем.

Как видно из приведённой выше таблицы, архитектурой нашего компьютера не предусмотрены операции формата `КОП r8,r16`, т.е. операции над регистрами разной длины запрещены, например, команды типа `add AL,BX` являются неправильными. Исходя из этого, для проведения операций над числами разной длины появляется необходимость *преобразования типов* из короткого целого в длинное, и из длинного в сверхдлинное (и наоборот). Такое преобразование, как можно (и нужно!) понять, зависит от знаковой или беззнаковой трактовки числа.

Беззнаковое число всегда расширяется из короткого формата в более длинный слева нулями, а для знакового числа слева размножается его знаковый бит. Таким образом, Вам необходимо понять, что для *знаковых* чисел незначащими двоичными цифрами будут 0 для неотрицательных и 1 для отрицательных значений. Для преобразования знаковых целых чисел из более короткого формата в более длинный 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 значения регистра AL до AX и AX до значения пары регистров <DX,AX> (так называемой *регистровой пары*), которые в этом случае рассматриваются как один длинный 32-х 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, т.е. будет иметь то же значение, что и исходной число, если слева будут отброшены только *незначащие* биты. Вам нужно обязательно понять, что для беззнаковых чисел незначащими будут всегда нулевые биты, а для знаковых – это биты, совпадающие по значению со знаковым битом *усечённого* числа.

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

КОП	R1	A2

Второй операнд A2 может в этом формате иметь один из приведённых ниже трёх видов:

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

Здесь A – задаваемое в команде число (смещение) длиной 1 или 2 байта (заметим, что нулевое смещение иногда может не задаваться и совсем 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 в учебных ЭВМ *адресом* мы обычно называли физический номер ячейки в памяти машины. В нашей новой машине *адресом* принято называть *смещение* ячейки относительно начала того сегмента, в котором она находится. Для обозначения же полного адреса будем употреблять термин *физический адрес*.

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

$$A_{физ} := (SEG*16 + A)_{mod\ 2^{20}},$$

где SEG обозначает значение одного из сегментных регистров (как для команды выбирается один конкретный из четырёх существующих сегментных регистров мы будем говорить далее).

Запись A2 = A[M1] означает использование *регистра-модификатора,* которым может быть любым из следующих регистров: BP, BX, SI, DI. Адрес операнда в сегменте при этом будем называть *испольнительным адресом* $A_{исп}$, он вычисляется так:

$$A_{исп} := (A + \langle M1\rangle)_{mod\ 2^{16}},$$

где вместо <M1> подставляется содержимое одного из четырёх указанных выше регистров модификаторов. В этом случае физический адрес операнда A2 будет вычисляться по формуле

$$A_{физ} := (SEG*16 + A_{исп})_{mod\ 2^{20}}$$

Запись A2 = A[M1][M2] обозначает использование в команде двух регистров-модификаторов[1] и вычисление исполнительного и физического адресов по формулам:

$$A_{исп} := (A + \langle M1\rangle + \langle M2\rangle)_{mod\ 2^{16}})$$

$$A_{физ} := (SEG*16 + A_{исп})_{mod\ 2^{20}}$$

На месте M1 можно указывать любой из регистров BX или BP, а на месте M2 – любой из регистров SI или DI (вообще говоря, можно сделать и наоборот, на месте M1 указывать SI или DI, а на месте M2 указывать BX или BP). Заметьте однако, что использование, например, регистров BX и BP (как и SI и DI) *одновременно* в качестве модификаторов запрещено (мы вскоре поймём, почему так сделано). Таким образом, смещение операнда в сегменте вычисляется как сумма двух или трёх чисел (взятая по модулю 2^{16}). Такой адрес, как мы указали выше, называется исполнительным адресом операнда и по аналогии с физическим адресом обычно обозначается как $A_{исп}$. Скажем также, что в

[1] В нашем языке Ассемблера допускается эквивалентная запись выражения A[M1][M2] в виде A[M1+M2] и даже, к сожалению, в виде [A+M1+M2], в то время, как, например, запись A[M1,M2] запрещена, что вносит сумятицу в мысли студентов.

старших моделях нашего семейства ЭВМ почти все ограничения на использование регистров в качестве модификаторов также были сняты (за счёт увеличения длины команды).

Возвращаясь к способу вычисления исполнительного и физического адресов можно заметить, что память сегмента, как и вся оперативная память, как бы замкнута в кольцо. Другими словами, при последовательном увеличении исполнительного адреса мы с последнего байта сегмента попадаем в начало этого же сегмента (на его нулевой байт).[1] Как уже отмечалось, это же касается и всей оперативной памяти, которая тоже может, как и в нашей учебной машине, считаться замкнутой в кольцо.

Регистры-модификаторы имеют и другое широко используемое название – *индексные* регистры, так как эти регистры часто используются для доступа к элементам массивов (как говорят, для индексации элементов массивов). При этом сам способ задания адресов с использованием индексных регистров называется **индексированием**. По сути, базирование и индексирование, – очень похожие способы адресации, однако, они преследуют разные цели. Как мы уже говорили, базирование используется для уменьшения объёма программного кода, в то время как индексирование предоставляет программисту удобный инструмент для работы с массивами. Заметим, что при использовании индексирования отпадает необходимость делать самомодифицирующиеся программы для обработки массивов. Теперь, изменяя значение индексного регистра, мы получаем доступ к нужным нам элементам массивов без изменения внешнего вида самой 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 (эти значения, как это часто делается в Ассемблере, записаны в шестнадцатеричном виде, напомним, что в этом случае перед числом записывается цифра ноль, если оно начинается с цифр A–F):

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

Тогда

$A_{исп} := (6 + 0FA00h + 0880h)_{mod\ 2^{16}} = 0286h$

$A_{физ} := (2000h*16 + A_{исп})_{mod\ 2^{20}} = (20000h + 0286h)_{mod\ 2^{20}} = 20286h$

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 машинное представление формата команды регистр-память. Длина этой команды 2, 3 или 4 байта:

8 бит			2 бита	3 бита	3 бита	8 бит	8 бит
КОП	D	w	mod	R1	mem	a8	a8->a16

где `mod` – двух битовое поле, называемой полем модификатора, `mem` – трёх битовое поле способа адресации, `a8` и `a16` – это обозначение одно- или двухбайтного смещения. Биты `d` и `w` уже знакомы нам из предыдущего формата регистр-регистр и имеют тот же смысл. Все возможные комбинации значения полей `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	Это уже 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	

Данная таблица показывает, как зависит способ адресации от значения полей `mem` и `mod`. Видно, что поле с именем `mod` фактически определяет, сколько байт в команде отводится под запись собст-

[1] Важно понять, что это касается только сегментов максимальной длины 2^{16} байт. В языке Ассемблера можно описывать и использовать сегменты и меньшей длины, которые, конечно, уже нельзя считать замкнутыми в кольцо.

венно смещения A (0, 1 или два байта). Как видим, эта таблица 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 только, что рассматриваемые нами форматы команд имеют следующие мнемонические обозначения:

- **RR** – (регистр – регистр);
- **RX** – (регистр – память или память – регистр, в зависимости от значения бита d в команде);
- **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 низкого уровня).

6.8.1. Команды пересылки

Команды пересылки – одни из самых распространённых команд в языке машины. Все они пересылают значение одного или двух байт из одного места памяти в другое. Для более компактного описания синтаксиса машинных команд введём следующие условные обозначения (с некоторыми из них мы уже знакомы):

`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`, `m32` – операнды, расположенные в основной памяти длиной `1`,`2` и `4` байта;
`i8`, `i16`, `i32` – непосредственные операнды в самой команде длиной `1`, `2` и `4` байта;
`SR` – один из трёх сегментных регистров `SS`, `DS`, `ES`;
`CS` – кодовый сегментный регистр.

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

```
mov op1,op2; op1 := op2
```

Здесь в комментарии мы указали семантику выполнения этой команды: второй операнд пересылается на место первого операнда. Существуют следующие допустимые форматы первого и второго операндов команды пересылки, запишем их в виде таблицы, где во второй колонке перечислены все возможные вторые операнды, допустимые для операнда из первой колонки:

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

Любопытно отметить, что запрещена команда пересылки вида `mov CS,op2`, так как по своей сути это будет уже команда передачи управления, а это уже совсем другой класс команд машины.

Команды пересылок не меняют флаги в регистре `FLAGS`. Как видим, в языке машины существует несколько десятков команд пересылок различных форматов. Из приведённой выше таблицы следует, что команды пересылок с кодом операции **mov** бывают форматов `RR`, `RX` (и `XR`), `RI` и `SI`. Существует также команда пересылки формата `SS` (память-память), но она имеет другой код операции и будет изучаться в разделе, посвящённом так называемым строковым (или цепочечным) командам.

Отметим также полезную команду *обмена* содержимым двух операндов

xchg `op1,op2;` обмен значениями операндов: `op1` ↔ `op2`.

Таблица допустимых операндов для этой команды:

op1	op2
r8	r8,m8
m8	r8

r16	r16, m16
M16	r16

Эта команда также не меняет флаги.

6.8.2. Арифметические команды

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

кoп op1,op2, где **кoп** = **add**, **sub, adc**, **sbb**.

Команды с кодами операций **add** (сложение) и **sub** (вычитание) выполняются по схеме:

op1 := op1 ± op2

Команды с кодами операций **adc** (сложение с учётом флага переноса) и **sbb** (вычитание с учётом флага переноса) имеют три операнда, два из которых задаются в команде явно, а третий по умолчанию является значением флага переноса CF:

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 всех этих операций всегда 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. учебники [5,9]), но это нас здесь интересовать не будет.

Далее рассмотрим команды умножения и деления целых чисел. Формат этих команд накладывает сильные ограничения на месторасположение их операндов, эти команды очень похожи на команды одноадресной ЭВМ. Первый операнд всех команд этого класса явно в команде не указывается и находится в фиксированном регистре, заданном *по умолчанию*. В младшей модели семейства есть следующие команды умножения и деления, в них, как и в уже знакомой нам одноадресной ЭВМ, явно задаётся только второй операнд (т.е. 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 самой команде явно задаётся только второй операнд (т.е. второй сомножитель или делитель). Этот операнд op2 может быть форматов r8 и m8 (соответственно говорят о коротком умножении или делении) или форматов r16 и m16 (это длинное умножение и деление). Обратите особое внимание на то, что операнд op2 не может быть форматов i8 и i16 (это типичная ошибка начинающих программистов). Как можно заметить, в отличие от команд сложения и вычитания, умножение и деление знаковых и беззнаковых целых чисел выполняются *разными* командами (по разным алгоритмам). Это легко понять, если, например, вспомнить знакомое нам ещё со школы правило умножения знаковых чисел "минус на минус даёт плюс".

В случае с короткими целыми операндами r8 и m8 при умножении вычисление производится по формуле:

AX := AL * op2

При делении на короткий операнд форматов r8 и m8 производятся следующие действия (операции **div** и **mod** понимаются в смысле языка Паскаль):

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

В случае с длинным вторым операндом формата r16 и m16 при умножении вычисление производится по формуле:

<DX,AX> := AX * op2

Как видим, произведение располагается сразу в двух регистрах <DX,AX> (как мы уже упоминали, это называется *регистровой парой*).

При делении на длинный операнд формата r16 и m16 вычисление производится по формулам:

```
AX := <DX,AX> div op2
DX := <DX,AX> mod op2
```

В этих командах операнд запись <DX,AX> обозначает 32-разрядное целое число, расположенное сразу в двух регистрах DX и AX, а op2, как уже говорилось, может иметь формат r16 или m16. Заметьте, что команды деления *одновременно* получают два результата (частное и остаток).

Как видим, команды умножения *всегда* дают точный результат, так как под хранение произведения выделяется в два раза больше места, чем под каждый из сомножителей. В то же время команды деления могут вызывать аварийную ситуацию, если частное не помещается в отведённое для него место, т.е. в регистры AL и AX соответственно. Такая ситуация называется целочисленным *переполнением*, при этом происходит аварийное прекращение выполнения программы. Разумеется, похожую аварийную ситуацию вызывает и деление на ноль. В то же время 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 значениях флагов CF=OF=1 произведение коротких целых чисел не помещается в регистр AL и частично "переползает" в регистр AH. Аналогично 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 (это двоичные нули для положительных и двоичные единицы для отрицательных произведений). Другими словами, при CF=OF=0 в качестве результата произведения можно взять только его младшую половину, что может оказаться полезным при программировании.

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" определённые флаги (в частности, портятся "полезные" флаги CF,OF,ZF,SF).

Для написания программ на Ассемблере нам будут нужны 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;

Здесь операнд op1 может быть форматов r8, m8, r16 и m16. Применение этих команд вместо соответствующих по действию команд вычитания и сложения приводит к более компактным программам. Необходимо однако отметить, что компактные команды **inc** op1 и **dec** op1, в отличие от эквивалентных им более длинных команд **add** op1,1 и **sub** op1,1 никогда не меняют флаг CF.[1]

[1] Это получилось потому, что схемы центрального процессора, выполняющие команды **inc** и **dec** (которые, как говорят, производят инкремент и декремент своего операнда) для экономии аппаратуры используются и при выполнении некоторых других команд, которые не должны менять этот флаг.

Глава 7. Язык Ассемблера

7.1. Понятие о языке Ассемблера

Наличие большого количества форматов данных и команд в архитектурах некоторых современных ЭВМ приводит к существенным трудностям при программировании на машинном языке. Для упрощения процесса написания программ для ЭВМ был разработан язык-посредник, названный *Ассемблером*,[1] 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[2] используется специальная программа-переводчик, также называемая *Ассемблером* (от английского слова "assembler" – "сборщик"). В зависимости от контекста, мы, если это не будет вызывать неоднозначности, в разных случаях под словом "Ассемблер" будет пониматься или сам 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 во всех наших примерах.

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

- заменять мнемонические обозначения кодов операций на соответствующие машинные коды операций (например, для нашей учебной машины, `ВЧЦ` → `002`);
- автоматически распределять память под хранение переменных, что позволяет программисту не заботиться о конкретном адресе переменной, если ему всё равно, где она будет расположена (внутри заданного сегмента при сегментной организации памяти);
- подставлять в программе вместо имён переменных их значения (обычно значение имени переменной – это адрес этой переменной в некотором сегменте);
- преобразовывать числа, написанные в программе в различных системах счисления во внутреннее машинное представление (в машинную систему счисления).

В конкретном Ассемблере обычно существует много полезных возможностей для более удобного написания программ, что возлагает на Ассемблер дополнительные функции, 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);
- возможность выполнить любую команду из языка машины.[3]

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. Так, считается, что на один оператор языка высокого уровня при программировании на Ассемблере приходится тратить примерно 10 предложений машинного языка. Кроме того,

---

[1] Как уже говорилось ранее, предшественниками языков Ассемблер были так называемые языки псевкокодов, такой язык псевдокода мы использовали при изучении архитектур учебных машин.

[2] Как мы узнаем позже, на самом деле производится перевод не на язык машины, а на специальный промежуточный язык, который называется *объектным* языком.

[3] Современные ЭВМ могут работать в так называемом привилегированном (или защищённом) режиме [18]. В этом режиме программы обычных пользователей, не имеющие соответствующих привилегий, не могут выполнять некоторое подмножество особых (привилегированных) команд из языка машины. Аналогично на современных ЭВМ существует так называемый режим защиты памяти, при этом одна программа не может иметь доступ (писать и читать) в память другой программы. Привилегированный режим работы и защиту памяти мы будем изучать в нашем курсе в главе, посвящённой мультипрограммному режиму работы ЭВМ.

отлаживать и модифицировать программы на Ассемблере значительно труднее, чем на языках высокого уровня.

На рис. 7.1 показана взаимосвязь языков программирования высокого уровня (их ещё называют машинно-независимыми), языков низкого уровня (машинно-ориентированных) и языка машины.

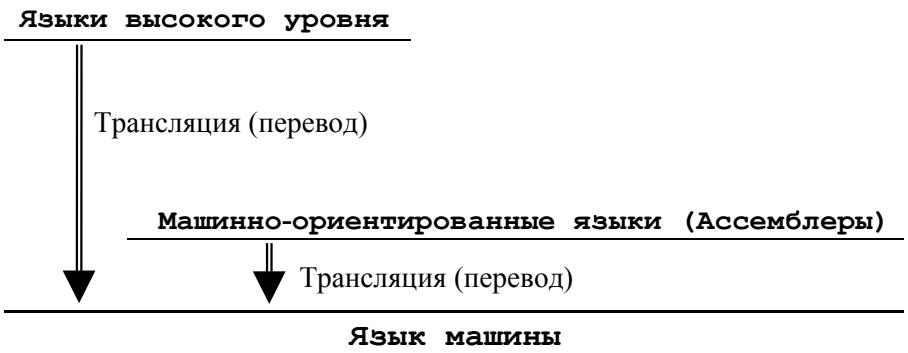


Рис. 7.1. Взаимосвязь языков программрования разных уровней.

Естественно, возникает вопрос, когда у программиста может появиться необходимость при написании своих программ использовать не более удобный язык программирования высокого уровня, а перейти на язык низкого уровня (Ассемблер). 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, как правило, 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 программе на языке высокого уровня. Кроме того, обычно от драйверов требуется, чтобы они были компактными и выполняли свою работу за минимально возможное время.

Вторая область применения Ассемблера связана с оптимизацией времени выполнения *больших* программ. Очень часто программы-переводчики (трансляторы) с языков высокого уровня дают весьма неэффективную программу на машинном языке. Особенно это касается программ вычислительного характера, в которых большую часть времени выполняется очень небольшой по длине (порядка 2-5%) участок программы (обычно называемый главным циклом). Для повышения эффективности выполнения таких программ могут использоваться так называемые многоязыковые системы программирования, которые позволяют записывать части программы на разных языках. Обычно основная часть оптимизируемой программы записывается на языке программирования высокого уровня (Фортране, Паскале, С и т.д.), а критические по времени выполнения участки программы – на Ассемблере. Скорость работы всей программы при этом может значительно увеличиться. Заметим, что часто это единственный способ заставить программу дать результат за приемлемое время.

7.3. Структура программы на Ассемблере

При дальнейшем изучения архитектуры компьютера нам придётся писать как фрагменты, так и полные программы на языке Ассемблер. Для написания этих программ мы будем использовать одну из версий языка Ассемблер, так называемый Макроассемблер версии 4.0 (MASM-4.0).[1] 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 и успешной сдачи экзамена. На лекциях мы подробно будем изучать только те осо-

[1] Эта одна из первых версий Ассемблера, в настоящее время существуют и более "продвинутые" версии этого языка. Причина, по которой мы не взяли для изучения более современную версию языка Ассемблер (например, MASM-6.0 или Турбо-Ассемблер) заключается в следующем. В новых версиях этого языка существуют развитые возможности по автоматизации процесса составления программ. Можно сказать, что эти возможности *повышают* уровень языка, их использование скрывает от программиста многие тонкости архитектуры нашего компьютера и позволяет ему не задумываться о некоторых особенностях в использовании машинных команд и сегментной памяти. В нашей же версии MASM-4.0 такие "продвинутые" возможности отсутствуют и студентам приходится всё делать вручную, что благотворно сказывается на усвоении учебного материала.

бенности и тонкие свойства языка Ассемблера, которые недостаточно полно описаны в указанных учебниках, но необходимы для хорошего понимания архитектуры нашей ЭВМ.

Изучение языка Ассемблера начнём с рассмотрения общей структуры программы на этом языке. Полная программа на языке Ассемблера состоит из одного или более *модулей*. Таким образом, Ассемблер принадлежит к классу так называемых модульных языков. В таких языках вся программа может разрабатываться, писаться и отлаживаться как набор относительно независимых программных частей – 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,[1] но позже нами будут рассмотрены и простые 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, иначе доступ к ним невозможен. Именно поэтому описание сегмента является важной синтаксической конструкцией языка Ассемблер.

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 с именами `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 64 Кбайта, и, если сегменты будут перекрываться, то одновременно для работы будет доступно меньшее количество оперативной памяти.[2] Заметим, что пересечение сегментов никак не влияет на логику работы центрального процессора.

В соответствии с принципом Фон Неймана, мы имеем право размещать в любом из сегментов памяти как числа, так и команды. Такой подход, однако, ведёт к плохому стилю программирования, программа перестаёт легко читаться и пониматься программистами, её труднее отлаживать и модифицировать. Кроме того, как мы вскоре узнаем, программа от этого становится ещё и длиннее. В нашей архитектуре программирование становится более лёгким, а сами программы компактнее, если размещать команды программы в одних сегментах, а данные – в других. Весьма редко программисту будет выгодно размещать, например, данные среди команд, один такой случай будет рассмотрен позже в нашем курсе.

На текущий сегмент команд должен указывать регистр `CS`, а на текущий 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* переменные в большинстве форматов команд, если прямо не указано противное, извлекаются из сегмента данных, на который указывает регистр `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 в основном 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

---

[1] Точнее, один модуль мы будем писать сами, а второй модуль, обеспечивающий операции ввода/вывода, будем брать в готовом виде и подключать к своей программе.

[2] Полное перекрытие всех сегментов позволяет использовать при работе программы всего один сегментный регистр, но ограничивает полную длину программы величиной 2^{16} байт. Такие маленькие выполняемые программы имеют на нашем компьютере имена с расширением `.com`, в отличие от "больших" программ, имеющих расширение `.exe`.

лишь отметим, что чаще всего директивы не определяют в программе ни команд, ни переменных (поймите, что именно поэтому они и могут стоять *вне* сегментов).[1]

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.

7.4. Классификация предложений языка Ассемблер

Будем классифицировать предложения Ассемблера по тем функциям, которые они выполняют в программе. Заметим, что эта классификация немного отличается от той, которая приведена в учебнике [5].

• **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 служебное имя **COMMENT** определяет многострочный комментарий, а символ * задаёт его границы, он эквивалентен символам начала и конца комментария { и } в языке Паскаль, этот символ 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 команде 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 также могут располагаться только внутри некоторого сегмента. В том сегменте, где они записаны, резервируются области памяти для хранения переменных. Это некоторый аналог описания переменных языка Паскаль. Заметим, что во многих учебниках такие предложения называют *директивами* резервирования памяти. Полные правила записи этих 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 A, B и C разной длины, которые, как мы уже привыкли в языке Паскаль, не будут иметь конкретных начальных значений, что отмечено символом вопросительного знака в поле параметров. 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 B, а *первый* – в начале переменной C (надо помнить о "перевёрнутом" хранении слов в памяти!). Поэтому следует быть осторожными и не считать A, B и C отдельными, "независимыми" переменными в смысле языка Паскаль, это просто именованные области памяти. Разумеется, в понятно написанной программе эти области лучше использовать так, как они описаны, то есть с помощью присвоенных им имён.

В качестве ещё одного примера резервирования памяти рассмотрим предложение

---

[1] 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.

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

Оно 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.** В некоторых учебниках директивы называют **командами Ассемблеру**, что хорошо отражает их назначение в программе.. 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, задавать синтаксические конструкции и управлять работой Ассемблера при компиляции (переводе) программы на язык машины. В качестве примера рассмотрим директивы для объявления начала и конца описания сегмента с именем A:

```
A segment
   ...
A ends
```

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

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

• **Макрокоманды.** Этот класс предложений Ассемблера относится к *макросредствам* языка, и будет подробно изучаться далее в нашем курсе. Пока лишь скажем, что на место макрокоманды при трансляции 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 в описании синтаксиса 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 обычно это указание на то, что данное предложение может рассматриваться как *команда*, т.е. на него Ассемблеру можно, как говорят, *передавать управление*.[1]

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 предложения, имеющего нестандартную структуру, может служить строка-комментарий, такие строки начинаются с символа точки с запятой, перед которой могут стоять только символы пробелов:

```
; это строка-комментарий
```

7.5. Пример полной программы на Ассемблере

Прежде, чем написать нашу первую полную программу на Ассемблере, нам необходимо научиться выполнять операции ввода/вывода, без которых, естественно, ни одна сколько-нибудь серьёзная программа обойтись не может. В самом языке машины, в отличие от, например, языка нашей

[1] Как мы узнаем при изучении команд переходов, наличие у метки двоеточия заставляет Ассемблер использовать прямой, а не косвенный тип перехода.

учебной машины УМ-3, нет команд ввода/вывода,[1] поэтому для того, чтобы, например, ввести целое число, необходимо выполнить некоторый достаточно сложный фрагмент программы на машинном языке.

Для организации ввода/вывода мы в наших примерах будем использовать набор макрокоманд 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 данной макрокоманды). Таким образом, на первом этапе изучения языка Ассемблера, мы избавляемся от трудностей, связанных с организацией ввода/вывода, используя для этого заранее заготовленные для нас фрагменты программ.

Нам понадобятся следующие макрокоманды ввода/вывода.

- **Макрокоманда вывода символа на экран**
  outch op1

где операнд op1 может иметь формат i8, r8 или m8. Значение операнда трактуется как код (номер) 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, т.е. конструкция 'A' полностью эквивалентна записи ord('A') языка Паскаль. Например, макрокоманда **outch '*'** выведет символ звёздочки на место курсора. Другими словами, эта макрокоманда эквивалентна оператору Паскаля для вывода одного символа Write(op1).

- **Макрокоманда ввода символа с клавиатуры** [2]
  inch op1

где операнд op1 может иметь формат r8 или m8. Код введённого символа записывается в место памяти, определяемое операндом. Эта макрокоманда эквивалентна оператору Паскаля для ввода одного символа Read(op1).

- **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 форматы i16, r16 или m16, а второго – i8, r8 или m8. Действие макрокоманды **outint** op1,op2 эквивалентно выполнению процедуры вывода одного целого значения языка Паскаль write(op1:op2), где второй параметр может задавать ширину поля вывода. Действие же макрокоманды с именем **outword** отличается только тем, что первый операнд при выводе *трактуется* как беззнаковое (неотрицательное) целое число. Заметим, что на месте операнда op1 нельзя использовать данные форматов r8 и m8.

- **Макрокоманда ввода целого числа**
  inint op1

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

- **Макрокоманда без параметров**
  newline

предназначена для перевода курсора к началу следующей строки экрана и эквивалентна вызову стандартной процедуры без параметров Writeln языка Паскаль. Этого же эффекта можно также достичь, если последовательно вывести на экран служебные символы с десятичными кодами 10 и 13, т.е. выполнить, например, две макрокоманды

---

[1] Точнее, в машинном языке есть только команды для обмена одним байтом или одним словом между регистром центрального процессора и заданным в команде периферийным устройством компьютера, с этими командами мы познакомимся позже.

[2] Как Вы должны помнить из курса языка Паскаль, на самом деле ввод производится из стандартного входного потока input, а вывод – в стандарьный выходной поток output. Чаще всего input подключён к клавиатеру, а output – к экрану терминала. Эти же соглашения действуют и в Ассемблере, хотя, в отличие от Паскаля, эти потоки и не называются input и output.

```
outch 10
outch 13
```

- **Макрокоманда без параметров**
  flush

предназначена для очистки буфера ввода и эквивалентна вызову стандартной процедуры без параметров Readln языка Паскаль.

- **Макрокоманда вывода на экран строки текста**
  outstr

Эта макрокоманда не имеет явных операндов, она всегда выводит на экран строку текста из того сегмента, на который указывает сегментный регистр DS, причём адрес начала выводимой строки в сегменте должен находится в регистре DX. Таким образом, физический адрес начала выводимого текста определяется по формуле

$$A_{физ} = (DS*16 + <DX>)_{mod\ 2^{20}}$$

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 символ 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)

$$X := (2*A - 1234 \ \mathbf{div}\ (A+B)^2) \ \mathbf{mod}\ 7$$

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 таким образом:

```
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, который помещается в один байт), однако мы выбрали для хранения X формат слова, т.к. его надо будет выдавать в качестве результата, а макрокоманда **outint** может выводить, как уже говорилось, только *длинные* целые числа.

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

```
include io.asm
; директива include вставляет 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
; Пусть (A+B)^2 помещается в одно слово!
      mov   cx,ax ; cx := младшая часть(A+B)^2
      mov   ax,1234
      cwd         ; <dx,ax> := сверхдлинное 1234
      idiv  cx    ; ax := 1234 div (A+B)^2
                  ; dx := 1234 mod (A+B)^2
      sub   bx,ax ; bx := 2*A - 1234 div (A+B)^2
      mov   ax,bx
      cwd
      mov   bx,7
      idiv  bx    ; dx := (2*A - 1234 div (A+B)^2) mod 7
      mov   X,dx
      outint X
      finish
code  ends
      end start
```

Подробно прокомментируем текст нашей программы. Во-первых, заметим, что сегмент стека с именем `stack` 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 таком переключении обязательно производится запись в сегмент стека, поэтому Ассемблер требует, чтобы такой сегмент был в любой полной программе. Подробно этот вопрос мы рассмотрим далее в нашем курсе при изучении *системы 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. Этой директивой программист уведомляет программу Ассемблер о том, что он (программист) гарантирует следующее: во время счёта программы сегментные регистры будут указывать на описанные в программе сегменты, как это показано на рис. 7.2 (хотя порядок сегментов в памяти и не обязательно будет именно таким). Заметьте, что сама директива **`assume`** не меняет значения ни одного сегментного регистра, подробно про неё необходимо обязательно 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* началом счёта нашей программы. Как мы потом увидим, это будет делать специальная служебная программа, которая называется *загрузчиком*, и нам, таким образом, об этом не стоит беспокоиться.[1]

Первые две команды нашей программы загружают значение сегментного регистра `DS`, иначе невозможно работать с данными их этого сегмента. В младшей модели для этого необходимы именно

[1] Точнее, в тексте программы нам необходимо только как-то задать значания, которые надо загрузить в эти регистры, что для этого надо сделать мы скоро узнаем, а затем ещё раз строго определим это при описании работы загрузчика.

две команды, так как одна команда, которую часто пишут нерадивые студенты, имела бы несуществующий формат:

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

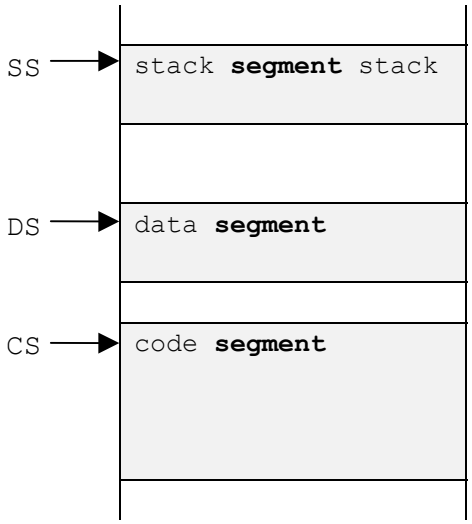


Рис. 7.2. Требуемые значения сегментных регистров
во время счёта нашей программы.

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 ; 6250=100000 div 16; формат r16,i16
```

Следующим предложением нашей программы является макрокоманда

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

которая 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 *знаковое* целое число B, которое мы считали в регистр al, в длинное целое на регистре ax. Далее вычисляется значение выражения $(A+B)^2$ и можно приступать к выполнению деления. Так как делитель является длинным целым числом (мы поместили это число на регистр cx), то необходимо применить операцию *длинного* деления, для чего делимое (число 1234 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 := 1234 div (A+B)^2 , dx := 1234 mod (A+B)^2
```

Далее мы аналогичным образом производим *длинное* деление значения вычисленного выражения 2*A-1234 **div** $(A+B)^2$ на число 7, присваиваем остаток от деления (он в регистре dx) переменной X и выводим на экран значение этой переменной по макрокоманде

```
outint X
```

которая эквивалентна оператору процедуры Write(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.**

Затем следует директива конца сегмента кода

```
code ends
```

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. первую выполняемую команду нашей программы. Явное задание входной точки позволяет начать выполнения сегмента команд с любого места, а не обязательно с первой команды этого сегмента.

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 целые 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 из регистра `ax`, предполагая, что на регистре `dx` находятся только *незначащие* цифры произведения. По-хорошему надо было бы после команды умножения проверить условие, что флаги `OF=CF=0`, это гарантирует, что в `dx` содержаться только нулевые биты, если $ax \geq 0$, и только двоичные "1", если $ax < 0$. Другими словами, все биты в регистре `dx` должны совпадать со старшим битом в регистре `ax`, для *знаковых* чисел это и есть признак того, что в регистре `dx` содержится *незначащая* часть произведения. И, наконец, мы не проверили, что не производим деления на ноль (для нашего случае, что введённое значение `A<>8`). В наших *учебных* программах мы чаще всего 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 байте оперативной памяти с десятичным адресом 234567_{10}.

Мы уже знаем, что запись в любой байт памяти возможна только тогда, когда этот байт располагается в каком-либо сегменте. Сделаем, например, так, чтобы наш байт располагался в том сегменте данных, на который указывает регистр `DS`. 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 их будущего месторасположения неизвестно.[1] Однако ничто не мешает нам в процессе счёта программы любой участок оперативной памяти сделать "временным" сегментом, установив на него какой-либо сегментный регистр. Так мы и сделаем для решения нашей задачи, установив сегментный регистр `DS` на начало ближайшей области памяти, 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:

$DS := 234567_{10}$ **div** $16 = 14660_{10}$.

При этом адрес `A` нашего байта в сегменте (его смещение от начала сегмента) вычисляется по формуле: $A := 234567_{10}$ **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]
```

В последней команде, увеличивающей значение нужного байта на единицу, нам пришлось явно задать размер операнда с помощью операции **`byte ptr`**. Дело в том, что по внешнему виду операнда этой команды `[bx]` программа Ассемблера не в состоянии определить, что подлежит увеличению: байт или слово (а это две *разные* команды машины, отличающиеся битом размера операнда `w`). Для

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

большинства команд, если Ассемблер не может определить в них длину операндов, он фиксирует ошибку в тексте программы и она не будет запускаться на счёт.

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

7.6. Переходы

В большинстве современных компьютеров, в том числе и в нашем, в соответствии с принципом Фон Неймана реализовано *последовательное выполнение команд*. В соответствие с этим принципом после выполнения текущей команды, если она не является командой перехода, счётчик адреса будет указывать на следующую (ближайшую с большим адресом) команду в оперативной памяти.[1] 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 команды в оперативной памяти, а выполнять программу "по кольцу" в большинстве случаев бессмысленно.

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 центрального процессора: сегментного регистра `CS` и регистра-счётчика адреса `IP` и вычисляется по формуле:

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

Следовательно, для осуществления любого перехода необходимо в один или оба эти регистра (т.е. в `CS` и/или `IP`) занести новые значения, соответствующие месторасположению следующей выполняемой команды. Отсюда вытекает первая классификация переходов: будем называть переход **близким** (или внутрисегментным)переходом, если при этом меняется *только* значение регистра `IP`, если же при переходе меняются значения обоих регистров, то такой переход будем называть **дальним** (или межсегментным) переходом.[2] Очевидно, что при близком переходе следующая выполняемая команда будет обязательно располагаться в том же сегменте кода, а при дальнем – уже в любом месте оперативной памяти (отсюда понятны названия этих переходов – близкие и дальние).

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

$$IP := (IP \pm Value)_{mod\ 2^{16}}$$

При **абсолютном** переходе происходит просто присваивание соответствующему регистру нового значения:

```
CS := Value или IP := Value
```

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 (обоснуйте, почему это так сделано!).

---

[1] Однако, как мы уже говорили, при достижении в программе конца оперативной памяти (или конца сегмента максимальной длины при сегментной организации памяти) обычно выполняется команда, расположенная в начале памяти или в начале этого сегмента (память как бы замкнута в кольцо).

[2] В принципе переход возможен и при изменении значения только одного регистра `CS`, но надо заметить, что в реальном программировании такие переходы практически всегда не имеют смысла и не реализуются в языке машины.

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):

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

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

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

Легко понять, что абсолютные переходы делать короткими бессмысленно, так как они могут передавать управление только в самие начало (первые 256 байт) оперативной память или сегмента кода.

Следующей основой для классификации переходов будет месторасположение величины, используемой при *абсолютном* переходе для задания нового значения какого-либо из этих регистров. При **прямом** переходе эта величина является просто числом (в нашей терминологии это *непосредственный* адрес в самой команде i8, i16 или i32). При **косвенном** переходе нужная величина располагается в памяти или на регистре, и в команде перехода задаётся *адрес* той области памяти (или номер того регистра), откуда и будет извлекаться необходимое число, например:

IP := <m16>

Здесь на регистр IP будет заноситься число, содержащееся в двух байтах памяти по адресу m16, т.е. это в нашей классификации *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 архитектуре нашего компьютера, так, мы уже знаем, что короткими или длинными бывают только относительные переходы, кроме того, относительные переходы бывают только прямыми.

7.7. Команды переходов

В первую очередь мы изучим *команды* переходов. Заметим, что эти команды предназначены *только* для передачи управления в другое место программы, они *не меняют* никаких флагов.

7.7.1. 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 операнд op1 может иметь следующие форматы:

op1	Способ выполнения	Вид перехода
i8	$IP:=(IP + i8)_{\mathrm{mod}\ 2^{16}}$	Близкий относительный короткий прямой
i16	$IP:=(IP + i16)_{\mathrm{mod}\ 2^{16}}$	Близкий относительный длинный прямой
r16	IP:=[r16]	Близкий абсолютный длинный косвенный
m16	IP:=[m16]	Близкий абсолютный длинный косвенный
m32	IP:=[m32], CS:=[m32+2]	Дальний абсолютный длинный косвенный
i32 = i16:i16= seg:off	IP:=off, CS:=seg	Дальний абсолютный длинный прямой

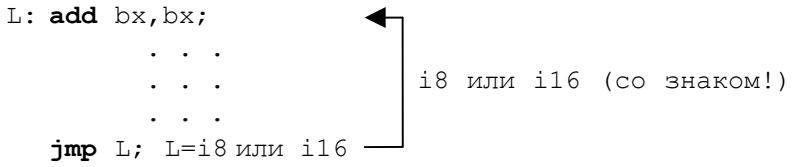
Здесь seg:off – это мнемоническое обозначение на Ассемблере двух операндов в формате i16, разделённых двоеточием, в машинной команде это просто два расположенных подряд двухбайтных поля (т.е. значение формата i32). Как видно из этой таблицы, многие потенциально возможные виды безусловного перехода (например, близкие абсолютные прямые, близкие абсолютные короткие и др.) не реализованы в нашей архитектуре. Это сделано, во-первых, потому, что это пере-

ходы практически бесполезны в реальном программировании, и, во-вторых, 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 задаётся код операции и операнды команд безусловного перехода. Для указания близкого относительного перехода в команде обычно записывается метка (т.е. имя) 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 требуемое смещение `i8` или `i16`, которое необходимо записать на место операнда в команде на машинном языке,[1] например:



Здесь формат для операнда `L` (`i8` или `i16`) выбирается программой Ассемблера автоматически, в зависимости от расстояния в байтах между командой перехода и командой с указанной меткой. Более сложным является случай, когда метка `L` располагается в программе *после* команды перехода. Тогда при первом просмотре текста программы Ассемблер, ещё не зная истинного расстояния до этой метки, "на всякий случай" отводит под поле смещения в команде два байта, т.е. операнд размером `i16`. Поэтому для тех программистов, которые знают, что смещение должно быть коротким (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 в основном при острой нехватке оперативной памяти для программы.[2] Для явного указания дальнего перехода на метку в другом сегменте памяти, программист должен использовать оператор **`far ptr`**, например:

```
jmp far ptr L
```

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

```
data  segment
   . . .
A1    dw L2;       Смещение команды с меткой L2 в своём сегменте
A2    dd Code1:L1; Это m32=seg:off
   . . .
data ends

code1 segment
      . . .
L1:   mov ax,bx
      . . .
code1 ends

code2 segment
      assume cs:code2, ds:data
```

---

[1] Необходимо учитывать, что в момент выполнения команды перехода счётчик адреса `IP` уже указывает на *следующую* команду, что, конечно, существенно при вычислении величины смещения в команде относительного перехода. Однако, так как эту работу выполняет программа Ассемблера, мы на такую особенность не будем пока обращать внимания, она будет существенной далее, при изучении команд вызова процедуры и возврата из процедуры.

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

```
start:mov   ax,data
      mov   ds,ax ;     загрузка сегментного регистра DS
L2:   jmp   far ptr L1; дальний прямой абсолютный переход, op1=i32=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
      jmp   A2;   дальний абсолютный косвенный переход, op1=m32
      jmp   bx;   близкий абсолютный косвенный переход, op1=r16
      jmp   [bx]; ошибка, нет выбора между: op1=m16 или op1=m32
      mov   bx,offset A2
      jmp   dword ptr [bx]; дальний абсолютный косвенный переход op1=m32
      . . .
code2 ends
```

Заметьте, что, если в команде перехода задаётся метка, то при прямом переходе она с двоеточием, а при косвенном – без двоеточия. Отметим здесь также одно важное преимущество относительных переходов перед абсолютными переходами. 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.[1] Как видим, архитектура нашего компьютера обеспечивает большой спектр команд безусловного перехода, вспомним, что в нашей учебной машине УМ-3 была только одна такая команда. На этом мы закончим наше краткое рассмотрение команд безусловного перехода. Напомним, что для усвоения этого материала по курсу Вам необходимо изучить соответствующий раздел учебника по Ассемблеру.

7.7.2. Команды условного перехода

Все команды условного перехода в нашей архитектуре выполняются по схеме, которую на Паскале можно записать как

if <условие перехода> **then goto** L

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

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

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

if rez <отношение> 0 **then goto** L

В нашей архитектуре 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 Ассемблера имеет такой вид:

j<мнемоника перехода> i8; $IP := (IP + i8)_{mod\ 2^{16}}$

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 такие же допустимые форматы операндов, 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

[1] Это полезное свойство относительных переходов позволяет, например, при необходимости достаточно легко *склеивать* два сегмента кода в один, что используется в системной программе редактора внешних связей, эту программу мы будем изучать позже.

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` **меньше** `01h = 1`, а как *беззнаковые* числа `0FFh = 255` **больше** `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 операнда. 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. Можно сказать, что операции отношения (кроме, естественно, операций "равно" и "не равно") как бы раздваиваются: есть "знаковое больше" и "беззнаковое больше" и т.д. Это учитывается в различных мнемониках этих команд.

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 этой таблицы.

В качестве примера рассмотрим, почему условному переходу `jl/jnge` (переход, когда первый операнд меньше или, что то же самое, не больше или равен второго операнда) 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 будет 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 флага не равны друг другу, т.е. должно выполняться условие `SF<>OF`, что и указано в нашей таблице. Для тренировки разберите правила формирования и других условий переходов в зависимости от значения соответствующих флагов.

Как видим, команд условного перехода достаточно много, поэтому понятно, почему для них реализован только один формат – близкий короткий относительный прямой 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 за счёт удлинения команд условного перехода. 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* величин `X,Y`:

```
X  dw   ?
Y  dw   ?
   . . .
   mov  ax,X
   cmp  ax,Y
   jg   L
   . . .
L:
```

может быть неверным, если расстояние в программе между меткой `L` 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 уже с длинным смещением:

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

```
        . . .
L:
```

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

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

Это, конечно, по необходимости, вынуждает нас использовать на Ассемблере плохой стиль программирования.

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 разультата (7-й или 15-ый, в зависимости от размера) равен единице
jns	SF = 0	Знаковый бит разультата (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 использования команд условного перехода рассмотрим программу, которая вводит знаковое число A в формате слова и вычисляет значение X по формуле

$$X:=\begin{cases}(A+1)*(A-1), & \text{при } A>100\\ 4, & \text{при } A=100\\ (A+1)\ \mathbf{mod}\ 7, & \text{при } A<100\end{cases}$$

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

```
include io.asm
; файл с макроопределениями для макрокоманд ввода-вывода
data  segment
A     dw ?
X     dw ?
Diagn db 'Ошибка – большое значение A!$'
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,100; Сравнение A и 100
      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> :=  32-хбитное (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 этой 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. Далее, мы предусмотрели выдачу 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 выполнились такие операции над *беззнаковыми числами*, это необходимо учитывать в надёжном программировании. У нас, однако, числа *знаковые*, поэтому всё в порядке.

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 часто приводить к ошибке. Здесь остаток от деления `(A+1)` на число `7` всегда поместится в регистр `ah`, однако *частное* `(A+1) div 7` может *не поместиться* в регистр `al` (пусть, например, `A=28000`, тогда `(A+1) div 7 = 4000` – не поместится в регистр `al`).

Для выдачи аварийной диагностики мы использовали предложения Ассемблера

```
Error:mov  dx,offset Diagn
      outstr
```

В первой команде применена специальная операция языка Ассемблера **`offset`**. Эта одноместная операция, будучи применена к некоторому имени, вычисляет адрес (т.е. смещение) этого имени от начала сегмента. Можно сказать, что эта операция смены способа адресации:

```
mov   ax,A; ax:=значение переменной A, это формат RX=r16,m16
mov   ax,offset A; ax:=адрес переменной A, это формат RI=r16,i16
```

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 программы между командой условного перехода и соответствующей меткой (включая код, подставляемый вместо макрокоманд **`outint`** и **`finish)`** будет больше `128` байт (обязательно понять это!).

7.7.3. 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`** S_1`;` S_2`; ...` S_k **`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 (и на одну метку) 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 выполнение 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. Особым является случай, когда N=0, в этом случае цикл выполнится 2^{16} раз. К сожалению, никакой другой регистр *нельзя* использовать для организации этого цикла (т.к. это неявный параметр команды цикла). Кроме того, в приведённом выше примере реализации цикла тело этого цикла не может быть слишком большим, иначе команда **loop** L *не сможет* передать управление на метку L.[1]

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. На языке Турбо-Паскаль эта программа могла бы выглядеть следующим образом (предполагая её последующий перенос на Ассемблер):

```
Const N:word=0;
Var S,ax.cx: integer;
Begin
   Write('Введите N<=500 '); Read(N);
   If N>500 then Writeln('Ошибка – большое N!') else
   If N>0 then begin Write('Вводите целые числа');
      For cx:=N downto 1 do begin Read(ax);
          If (-2000<=ax) and (ax<=5000) do S:=S+ax
      End
   End;
   Writeln('S=',S)
End.
```

На Ассемблере 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 $'
T2    db    'Ошибка – большое N!$'
```

[1] В нашей архитектуре для эффективной реализации циклов применяются условные команды близкого перехода, как следствие тело цикла не может быть слишком большим (порядка 30-40 машинных команд), иначе, как уже говорилось, приходится пользоваться вспомогательными командами длинного безусловного перехода. Архитектурное решение о том, что циклы и условные переходы реализуются командами *короткого* перехода, опирается на так называемое свойство **локальности** программ. Это свойство означает, что бóльшую часть времени центральный процессор выполняет команды, расположенные (локализованные) внутри относительно небольших участков программы (это и есть программные циклы), и достаточно редко переходит из одного такого участка в другой. Как мы узнаем позже, на свойстве локальности программ основано и существование в современных компьютерах специальной памяти типа кэш.

```
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 об ошибке
Err:  outstr
      newline
      finish
L1:   mov   cx,N; Счётчик цикла
      jcxz  Pech; На печать результата при N=0
      mov   dx,offset T3; Приглашение к вводу
      outstr
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 сумма всегда имеют формат **dw** (помещаются в слово), иначе выдадим аварийную диагностику. 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 (?); Массив X=2*N байт
data1 ends
```

```
data2 segment
Y     dw    N dup (?); Массив Y=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_my_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 bx на X[1]
      mov   si,offset Y+2*N-2; указатель si 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, т.е. i:=i+1
      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_my_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 далее в программе, где встретится имя N, надо подставить вместо него операнд этой директивы, т.е. число 20000. Таким образом, это почти полный аналог описания константы в языке Пас-

каль.[1] Так как длина каждого элемента массива равна двум байтом, то под каждый из массивов соответствующая директива **dw** зарезервирует `2*N` байт памяти.

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 с именем `data1`, а массив `Y` – в сегменте с именем `data2`. Директива **assume** говорит, что во время счёта на эти сегменты будут соответственно указывать регистры `DS` и `ES`, что мы и обеспечили, загрузив эти регистры нужными значениями 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 пользу от директивы **assume**: при доступе к элементам массива `Y` программа Ассемблера 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[2] 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` (директива **assume** здесь нам, к сожалению, помочь не сможет).

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 значение типа – 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 модуле, что и оператор **type**, или отрицательное число `−2` для меток из других модулей.[3] Все другие имена (в частности, имена регистров и сегментов), а также имена констант, имеют тип ноль. В остальных случаях попытка использовать этот оператор без имени (например, `type [bx]`) либо вызовет в нашем Ассемблере синтаксическую ошибку (например, в макрооператоре[4] присваивания `K=type [bx]` – ошибка), либо оператор **type** будет проигнорирован (например, `mov ax,type [bx]` ≡ `mov ax,[bx]`).

Вы, наверное, уже почувствовали, что программирование на Ассемблере сильно отличается от программирования на языке высокого уровня (например, на Паскале). Чтобы подчеркнуть это разли-

---

[1] Если не принимать во внимание то, что константа в Паскале имеет *тип*, это позволяет контролировать её использование в программе, а директива эквивалентности в Ассемблере – это просто указание о *текстовой подстановке* вместо имени заданного операнда директивы эквивалентности. Иногда перед такой подстановкой производится также некоторое *вычисление* операнда директивы эквивалентности, подробнее об этом необходимо прочитать в учебнике [5].

[2] Мы называем эту пару команд тесносвязанными, потому что при счёте между ними никогда не происходит прерывания выполнении программы, подробнее от этом в разделе, посвящённом системе прерываний.

[3] Мы уже упоминали, что программа на Ассемблере может состоять из отдельных модулей, подробно об этом мы будем говорить в главе "Модульное программирование".

[4] Об этом операторе мы будем говорить в отдельной главе, посвящённой макросредствам Ассемблера.

чие, 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;
Begin
      { 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];
```

Для переноса программы с языка высокого уровня (в нашем случае с Паскаля) на язык низкого уровня (Ассемблер) необходимо выполнить два преобразования, которые в программистской литературе часто называются *отображениями*. Как говорят, надо отобразить с языка высокого уровня на язык низкого уровня **структуру данных** и **структуру управления** программы. В нашем примере главная структура данных в программе на Паскале – это прямоугольная таблица (матрица) целых чисел. При отображении матрица на линейную память некоторого сегмента данных в Ассемблере приходится, как говорят, *линеаризировать* матрицу. Этим мудрёным термином обозначают совсем простой процесс: сначала в сегменте данных размещается первая строка матрицы, сразу вслед на ней – вторая, и т.д. Для нашего примера в некотором сегменте надо каким-то образом зарезервировать 2*N*M байт памяти.[1]

Теперь займёмся отображением структуры управления нашей программы на Паскале (а попросту говоря – её условных операторов и циклов) на язык Ассемблера. Сначала обратим внимание на то, что переменные `i` и `j` несут в нашей 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. Теперь, чтобы по индексам элемента массива вычислить его адрес в сегменте данных, приходится выполнить достаточно сложные действия. Например, адрес элемента `X[i,j]` компилятору с Паскаля приходится вычислять так:

```
Адрес(X[i,j]) = Адрес(X[1,1])+2*M*(i-1)+2*(j-1)
```

Эту формулу легко понять, учитывая, что для Ассемблера матрица хранится в памяти компьютера 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 отображении этих циклов на язык Ассемблера лучше всего **разделить** функции счётчика цикла и индекса элементов. В качестве счётчика будем использовать регистр `cx` (он и специализирован для этой цели), а адреса элементов матрицы 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}[2]
```

[1] Процесс линеаризации усложняется, когда надо отобразить массивы больших размерностей (например, четырёхмерный массив – совсем обычная вещь в задачах из области физики твёрдого тела или механики сплошной среды). Подумайте, как надо сделать отображение такого массива на линейную память. Кроме того, надо сказать, что некоторые языки программирования высокого уровня требуют другого способа линеаризации. Например, для языка Фортран нужно сначала разместить в памяти сегмента первый столбец матрицы, потом второй и т.д.

[2] В языке Турбо-Паскаль вместо недопустимого `bx:=↑X[1,1];` можно использовать оператор присваивания `bx:=@X[1,1]`

```
          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
             . . .
; Здесь 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; На след. строку
L4:    loop  L1
```

Приведённый пример очень хорошо иллюстрирует стиль мышления программиста на Ассемблере. Для доступа к элементам обрабатываемых данных применяются указатели (ссылочные переменные, значениями которых являются адреса), и используются операции над этими адресами (так называемая *адресная арифметика*). Получающиеся программы могут максимально эффективно учитывать все особенности архитектуры используемого компьютера. Заметим, что применение адресов 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 теперь 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 / (слэш).

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.

7.8. Работа со стеком

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

Стеком в архитектуре нашего компьютера 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 далее иметь в виду.

В нашей архитектуре стек хранится в сегменте в перевёрнутом виде: начало сегмента является концом стека, а конец сегмента – началом стека. Кроме начала и конца, у стека есть текущая позиция – *вершина* стека, её смещение от начала сегмента стека всегда записано в регистре `SP` (stack pointer). Следовательно, как мы уже знаем, физический адрес вершины стека можно получить по формуле $A_{физ} = (SS*16 + SP)_{mod\ 2^{20}}$.

Стек есть аппаратная реализация абстрактной структуры данных *стек*, с которой Вы познакомились в прошлом семестре. В вершину стек можно записывать (и, соответственно, читать из неё) только машинные *слова*, команды чтение и запись в стек байтов не предусмотрены 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 вершину стека и чтения из вершины стека данных этих форматов.[1]

В соответствие с определением машинного стека последнее записанное в него слово будет храниться в вершине стека и читаться из стека первым. Это так называемое правило "последний пришёл – первый вышел" (английское сокращение LIFO – Last In First Out).[2] Обычно стек принято графически изображать "растущим" снизу-вверх, от конца к началу сегмента стека. Как следствие получается, что конец стека фиксирован и расположен на рисунках снизу, а вершина двигается вверх (при записи в стек) и вниз (при чтении из стека).

В любой момент времени регистр `SP` указывает на вершину стека – это *последнее слово*, записанное в стек. Таким образом, регистр `SP` является специализированным регистром, на что указывает и его название (`SP` – Stack Pointer), следовательно, хотя на нём и можно выполнять арифметические операции сложения и вычитания, но делать это следует только тогда, когда мы хотим изменить положение вершины стека. Обычно стек изображают, как показано на рис. 7.3.

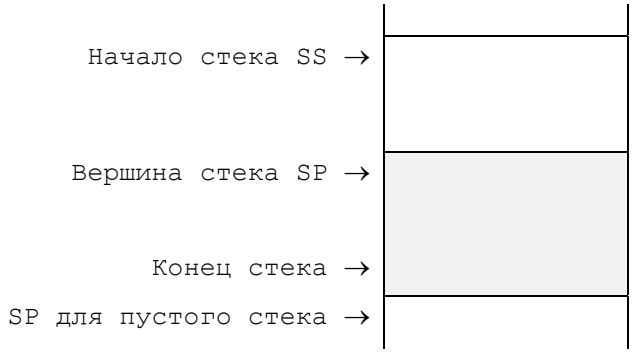


Рис. 7.3. Так мы будем изображать стек.

[1] Конечно, наличие стека не отменяет для нашего компьютера принципа Фон Неймана однородности памяти, поэтому одновременно стек является и просто частью оперативной памяти и, таким образом, возможен обмен данными с этой памятью с помощью обычных команд (например, команд пересылки).

[2] Вообще говоря, это же правило можно записать и как "первый пришёл – последний вышел" (английское сокращение FILO – First In Last Out). В литературе встречаются оба этих правила и их сокращения.

На этом рисунке, как и положено в нашей архитектуре, 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. Это происходит из-за того, что сегмент максимального размера, как мы знаем, как бы склеен в кольцо. Именно поэтому стеки максимального размера использовать в программах 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, стек такого же размера можно описать так:

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

То, что этот сегмент будет при запуске программы на счёт использоваться именно как начальный сегмент стека, указывает не имя стека, а его параметр **stack** в директиве **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* или 128 *байта*, поэтому в начале работы регистр SP будет иметь значение 128, т.е., как мы и говорили ранее, он 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 нужные 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**. Как видим, программа загрузчика выполняет, в частности, те же функции начальной установки значений необходимых регистров, которые в нашей учебной машине выполняло устройство ввода при нажатии кнопки ПУСК. Разумеется, позже при работе программы мы и сами можем загрузить в регистры SS и SP новые значения, это будет *переключением* на другой сегмент стека.

Теперь приступим к изучению команд, которые работают со стеком (т.е. читают из него и пишут в него слова). Рассмотрим сначала те команды работы со стеком, которые *не являются* командами перехода. Команда

push op1

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

$SP := (SP-2)_{mod\ 2^{16}}$; <SS,SP> := op1

Здесь, как и ранее, регистровая пара <SS,SP> обозначает в стеке слово с адресом, вычисляемым по формуле

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

---

[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 [17]) подходят к этому вопросу более строго и не допускают использования служебных слов в качестве имён пользователя. Заметим, что все служебные слова Ассемблера, за исключением имён регистров, мы выделяем в нашей книге жирным шрифтом.

Особым случаем является команда

```
push  SP
```

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

$$\texttt{<SS,SP> := SP; SP := (SP-2)}_{\text{mod } 2^{16}}$$

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 вершины стека слово и записывает его в место памяти, определяемое своим операндом. Это команда выполняется по правилу:

$$\texttt{op1 := <SS,SP>; SP := (SP+2)}_{\text{mod } 2^{16}}$$

Команда без явного параметра

```
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.

Машинные 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` будет равно нулю, как для полного, так и для пустого стека (обязательно понять это!), поэтому не рекомендуется использовать в программе стек максимального размера.

При проверки переполнения стека следует иметь в виде следующее важное обстоятельство. В стеке всегда должен оставаться некоторый "неприкосновенный запас" свободного места, так как в любой момент времени сам компьютер (его центральный процессор) может записать в стек некоторые данные (подробно про это мы будем говорить при изучении системы прерываний). Исходя из этого, в программе следует проверять переполнение стека, например, такими командами:

```
cmp   SP,K
gb    err
```

В наших предыдущих примерах под такой "неприкосновенный запас" `K` мы отводили `64` или `128` байт (посмотрите на описание сегмента стека в этих примерах).

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, равное размеру 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, так как обмен со стеком производится только словами (по два байта), и один байт никогда не будет использоваться. Кроме того, для стеков нечётной длины труднее проверить переполнение и пустоту стека.[1]

[1] Важность контроля правильной работы со стеком можно проиллюстрировать тем фактом, что в программах на Турбо-Паскале **всегда** включён контроль пустоты и переполнения стека, т.е. этот контроль невоз-

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* те из введённых чисел, которые принадлежат диапазону [2..100] (сделаем спецификацию, что таких чисел может быть, например, не более 300). Ниже приведено возможное решение этой задачи.

```
include io.asm
st    segment stack
      db    128 dup (?); это запас 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 [2..100]
      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, на начало которого установили сегментный регистр CS. Можно считать, что сегменты данных и кода в нашей программе в некотором смысле *совмещены*. Мы разместили строковые константы в самом *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 бу-

---

можно выключить никакой директивой транслятору, в отличие от, например, контроля переполнения при работе с целыми числами или выхода индекса за границы массива.

дет рассказано далее) и 300 слов мы отвели для хранения вводимых нами чисел. При реализации этой программы, анализируя переполнение стека, мы использовани команды:

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

Как уже отмечалось выше, это можно сделать и командами

```
cmp   SP,128; только "неприкосновенный запас"?
jb    Err
```

Теперь, после того, как мы научились работать с командами записи слов в стек и чтения слов из стека, вернёмся к дальнейшему рассмотрению команд перехода.

7.9. Команды вызова процедуры и возврата из процедуры

Команды вызова процедуры по-другому называются командами *перехода с запоминанием точки возврата*,[1] что более правильно, так как их можно использовать и вообще 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. Таким образом, обеспечивается возможность возврата в точку программы, следующую непосредственно за командой вызова процедуры. 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 обычного безусловного перехода **jmp** op1 здесь не реализован только близкий короткий относительный переход **call** i8, он практически бесполезен в практике программирования, так как почти всегда процедура находится достаточно далеко от точки её вызова. Следовательно, как и команды безусловного перехода, команды вызова процедуры бывают *близкими* (внутрисегментными, форматы i16, r16, m16) и *дальними* (межсегментными, форматы m32 и i32). Близкий вызов процедуры выполняется по следующей схеме:

```
Встек(IP); jmp op1
```

Здесь запись **Встек**(IP) обозначает действие "записать значение регистра счётчика адреса IP в стек". Заметим, что это часть действия при вызове процедуры, а 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 в Ассемблере 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 *беззнаковый* параметр этой команды i16, то Ассемблер автоматически полагает операнд i16=0.

Команда *близкого* возврата из процедуры выполняется по схеме:

```
Извстека(IP); SP:=(SP+i16) mod 2^16
```

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

Команда *дальнего* возврата из процедуры выполняется по схеме:

```
Извстека(IP); Извстека(CS); SP:=(SP+i16) mod 2^16
```

По нашей классификации переходов команда **ret** осуществляет близкий или дальний абсолютный косвенный переход. Дополнительной действие команды возврата SP:=(SP+i16) mod 2^{16} приводит к тому, что для параметра i16<>0 указатель вершины стека SP устанавливается на некоторое другое место в стеке. В большинстве случаев этот операнд имеет смысл использовать для *чётных* i16>0, и только тогда, когда SP+i16<=K, где K – размер стека. В этом случае из стека *удаляются*

[1] В некоторых книгах такие команды называют "командами перехода с возвратом", что, конечно, не совсем правильно, так как сами эти команды никакого "возврата" не производят.

`i16 div 2` слов, что можно трактовать как *очистку* стека от данного количества слов. Возможность очистки стека при выполнении команды возврата, как мы вскоре увидим, будет весьма полезной при программировании процедур на Ассемблере, она будет аналогом уничтожения локальных переменных Паскаля при выходе из процедуры.

7.10. Программирование процедур на Ассемблере

В языке Ассемблера [1] есть понятие процедуры – это участок программы, который начинается директивой

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

и заканчивается директивой

```
<имя процедуры>  endp
```

Первое предложение будем называть *заголовком* процедуры. Все предложения Ассемблера между директивами заголовка и конца процедуры называются *телом* этой процедуры. Синтаксис Ассемблера допускает, чтобы тело процедуры было пустым, т.е. не содержало ни одного предложения. Вложенность процедур одна в другую, в отличие от языка Паскаль, *не допускается*. Заметьте также, что, в отличие от Паскаля, имена не могут быть локализованы в теле процедуры, т.е. они видны из любой точки программного модуля (мы рассмотрим исключение из этого правила при изучении макросредств Ассемблера). Учтите также, что имя процедуры имеет тип метки, хотя за ним и не стоит двоеточие, как это принято для меток команд. Вызов процедуры обычно производится командой **call**, а возврат из процедуры – командой **ret**.

Спецификация процедуры является необязательным параметром заголовка процедуры, она задаётся служебными именами **near** или **far**. Эти имена являются служебными именами констант, **far**=−2 и **near**=−1.[2] Отметим и другие полезные служебные имена констант в языке Ассемблера: **byte**=1, **word**=2, **dword**=4, **abs**=0. Если спецификация в заголовке процедуры 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 будет близкий возврат, а для дальней – дальний возврат. Отметим, что команду **ret**, вообще говоря, можно использовать и *вне* тела процедуры, в этом случае Ассемблер выбирает команду *близкого* возврата.

Заметим, как при снижении *уровня* языка программирования упрощаются его основные понятия. Так, в Паскале у нас было описание и вызов процедур и функций (да ещё и с развитым механизмом формальных и фактических параметров). На языке Ассемблера у нас остаётся только описание процедуры, а о том, что некоторая процедура "на самом деле" является не процедурой, а функцией, знает тлько сам программист, но не программа Ассемблер. Также в языке Ассемблер нет понятия формальных и фактических параметров процедуры, и нам придётся *моделировать* этот механизм при программировании. Кроме того, по сравнению с Паскалем, само понятие процедуры в Ассемблере резко упрощается. Чтобы продемонстрировать это, рассмотрим такой синтаксически правильный фрагмент программы:

```
        mov   ax,1
 F      proc
        add   ax,ax
 F      endp
        sub ax,1
```

Этот фрагмент полностью эквивалентен таким командам:

```
        mov   ax,1
 F:     add   ax,ax
        sub   ax,1
```

---

[1] Заметьте, что в самом машинном языке понятие процедуры отсутствует, а команда **call**, как уже говорилось, является просто командой безусловного перехода с запоминанием в стеке адреса следующей за ней команды. О том, что эта команда используется для вызова процедуры, знает только программист.

[2] К сожалению, по непонятным причинам Ассемблер MASM 4.0 не позволяет писать заголовок процедуры в виде `proc -1`, а требует писать `proc near`, хотя в других местах программы имена **near** и **far** полностью эквивалентны константам −1 и −2 соответственно.

Другими словами, описание процедуры на Ассемблере может встретиться в любом месте сегмента, это просто некоторый фрагмент кода, заключённый между директивами начала **proc** и конца **endp** описания процедуры.[1]

Аналогично, команда **call** является просто особым видом команды безусловного перехода, и может не иметь никакого отношения к описанию процедуры. Например, рассмотрим следующий синтаксически правильный фрагмент программы:

```
L:      mov   ax,1
        . . .
        call  L
```

Здесь вместо имени процедуры в команде **call** указана обычная метка. И, наконец, отметим, что в языке машины уже нет понятия процедур и функций, а также передачи параметров, и программисту приходится моделировать все эти понятия.

Изучение программирования процедур на Ассемблере начнём со следующей простой задачи, в которой "напрашивается" использование процедур. Пусть надо ввести массивы `X` и `Y` знаковых целых чисел размером в слово, массив `X` содержит `100` чисел, а массив `Y` содержит `200` чисел. Затем необходимо вычислить величину `Sum`:

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

Будем предполагать, что оба массива находятся в одном сегменте данных (на него, как обычно, указывает регистр `DS`), а переполнение результата при сложении элементов массивов будем для простоты игнорировать (т.е. выводить неправильный ответ без выдачи об этом диагностики). Для данной программы естественно сначала реализовать процедуру суммирования элементов массива и затем дважды вызывать эту процедуру соответственно для массивов `X` и `Y`. Текст нашей процедуры мы, как и в Паскале, будем располагать *перед* текстом основной программы (первая выполняемая команда программы, как мы знаем, помечено меткой, указанной в директиве **end** нашего модуля), хотя с таким же успехом процедуру можно было бы располагать и в конце программного сегмента, после макрокоманды **finish**.

Заметим, что, вообще говоря, мы будем писать *функцию*, но в языке Ассемблера, как уже упоминалось, 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 для этой цели обобщённый термин *процедура*.

Перед тем, как писать процедуру, необходимо составить *соглашение о связях* между основной программой и процедурой. Здесь необходимо уточнить, что под *основной программой* мы имеем в виду то место нашей программы, где процедура вызывается по команде **call**, т.е. вполне возможен и случай, когда одна процедура вызывает другую (в том числе и саму себя, используя прямую или косвенную рекурсию).[2] Соглашение о связях между основной программой и процедурой включает в себя место расположение и способ передачи параметров, возврата результата работы (для функции) и некоторую другую информацию. Так, для нашего последнего примера мы "договоримся" с процедурой, что суммируемый массив *слов* будет располагаться в сегменте данных (на него указывает регистр `DS`), адрес начала массива (его первого элемента) перед вызовом процедуры будет записан в регистр `bx`, а количество элементов – в регистр `cx`. Сумма элементов массива при возврате из процедуры должна находится 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    ?
```

[1] Замечание для продвинутых студентов: примерно такие же процедуры были в некоторых первых примитивных языках высокого уровня, например, в начальных версиях языка Basic.

[2] Из материала прошлого семестра Вы должны помнить, что при *прямой* рекурсии процедура вызывает себя сама, а при *косвенной* – через цепочку других процедур.

```
data  ends
stack segment stack
      dw    64 dup (?)
stack ends
code  segment
      assume cs:code,ds:data,ss:stack
Summa proc  near
; 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; на следующий элемент
      loop  L
      ret
Summa endp
start:mov   ax,data
      mov   ds,ax
; здесь надо 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
code  ends
      end start
```

Надеюсь, что Вы легко разберитесь, как работает эта программа. А вот теперь, если попытаться "один к одному" переписать эту Ассемблерную программу на язык Паскаль, 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.
```

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

Как видим, у этой программы очень плохой стиль программирования, так как неявными параметрами процедуры являются глобальные переменные, т.е. полезный механизм передачи параметров Паскаля просто не используется. В то же время именно хорошо продуманный аппарат формальных и фактических параметров делает процедуры и функции таким гибким, эффективным и надёжным механизмом в языках программирования высокого уровня. Если с самого начала решать задачу суммирования массивов на Паскале, а не на Ассемблере, мы бы, конечно, 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)+Summa(Y,200); Writeln(Sum)
End.
```

Это уже достаточно грамотно составленная программа на Паскале. Теперь нам надо научиться так же хорошо писать программы с процедурами и на Ассемблере, однако для этого нам понадобятся другие, несколько более сложные, соглашения о связях между процедурой и основной программой. Если Вы хорошо изучили материал предыдущего курса, то должны знать, что грамотно написанная процедура в языке Паскаль получает все свои аргументы как фактические параметры и не использует внутри себя имён глобальных переменных. Такого же стиля работы с процедурами мы должны придерживаться и при программировании на Ассемблере. При работе с процедурами на языке Ассемблера мы будем использовать так называемые **стандартные соглашения о связях**.

7.10.1. 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 процедуры, реализующие алгоритмы для некоторой предметной области (например, для работы с матрицами), так что программист может внутри своей программы вызывать нужные ему процедуры из такой библиотеки. Библиотеки стандартных процедур обычно поставляются в виде набора так называемых *объектных модулей*, что исключает для пользователя возможность вносить какие-либо изменения в исходный текст этих процедур (с объектными модулями мы познакомимся далее в нашем курсе). Таким образом получается, что взаимодействовать с процедурами из такой библиотеки можно только используя те соглашения о связях, которые были использованы при создании этой библиотеки.

Другим примером является написание частей программы на различных 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 в программе на Паскале 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 этом языке (более точно – в *системе программирования*, включающей в себя этот язык) тоже соблюдаются такие же стандартные соглашения о связях.

Рассмотрим типичные стандартные соглашения о связях между основной программой и процедурой, обычно они в себя включают следующие пункты.

- Фактические параметры перед вызовом процедуры или функции записываются в стек.[1] Как мы помним, в языке Паскаль параметры можно передавать в процедуру по значению и по ссылке, это верно и для многих других языков программирования.[2] Так вот, при передаче параметра *по значению* в стек записывается само это значение, а в случае передачи параметра *по ссылке* в стек записывается адрес начала фактического параметра.[3] 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 C это *обратный* порядок, а в большинстве других языков программирования высокого уровня – *прямой*.[4]
- Если в процедуре или функции необходимы *локальные* переменные (в смысле языка Паскаль), то место им отводится в стеке. Обычно это делается путём записи в стек некоторых величин, или же увеличением значения указателя вершины стека, для чего в нашей архитектуре, 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.[5]
- Если в процедуре или функции изменяются какие-либо регистры, то в начале работы необходимо запомнить значения этих регистров в локальных переменных (т.е. в стеке), 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 принято не запоминать и не восстанавливать 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] Большинство современных компьютеров 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] В некоторых языках программирования одного из этих способов может и не быть. Так, в первой версии языка C (не C++) параметры передаются только по значению, а во многих версиях языка Фортран – только по ссылке.

[3] Является ли этот адрес для нашей архитектуры близким (т.е. *смещением* в сегменте данных) или же дальним (это значения сегментного регистра и смещение) обычно специфицируется при написании процедуры на языке высокого уровня. Ясно, что, скорее всего во внешнюю процедуру будут передаваться дальние адреса.

[4] Обратный порядок позволяет более легко реализовывать процедуры и функции с *переменным* числом параметров (в языке Паскаль, как Вы должны знать, для процедур *пользователя* это запрещено), в то же время обратный порядок менее эффективен, чем прямой, так как труднее удалить из стека фактические параметры после окончания процедуры. В связи с этим, например, в языке C при описании внешней процедуры программист может явно указывать порядок записи фактических параметров в стек.

[5] В различных системах программирования стандартные соглашения о связях могут несколько различаться. Например, результат значения функции может возвращаться не на регистре `ax`, как у нас, а на *вершине стека*. Понятно, что всё это должно учитываться при программировании.

Участок стека, в котором для процедуры или функции размещаются их локальные переменные (в частности, фактические параметры и адрес возврата) называется *стековым кадром* (stack frame). Стековый кадр начинает строить основная программа перед вызовом процедуры или функции, помещая туда (т.е. записывая в стек) фактические параметры. Затем команда **call** помещает в стек адрес возврата (это одно слово для близкой процедуры и два – для дальней) и передаёт управление на начало процедуры. Далее уже сама процедура или функция продолжает построение стекового кадра, размещая в нём свои локальные переменные (в частности, для хранения значений изменяемых регистров).

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

Перепишем теперь нашу последнюю "хорошую" программу с языка Паскаль на Ассемблер с использованием стандартного соглашения о связях. Будем предполагать, что передаваемый по ссылке адрес фактического параметра-массива занимает одно слово (т.е. является близким адресов – смещением в сегменте данных). Для хранения стекового кадра (локальных переменных функции) дополнительно зарезервируем в стеке, например, 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  cx;    запоминание используемых регистров
      sub   sp,2;  порождение локальной переменной
S     equ   word ptr [bp-6]
; S будет именем 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];нельзя add S,[bx]
      add   S,ax   ;так как нет формета память-память
      add   bx,2
      loop  L
      mov   ax,S; результат функции на ax
      add   sp,2; уничтожение локальной переменной S
      pop   cx
      pop   bx
      pop   bp; восстановление регистров cx, bx и bp
      ret   2*2
; возврат с очисткой стека от фактических параметров
```

[1] При обратном порядке записи фактических параметров в стек (как в языке C) удаление из стека фактических параметров обычно делает не процедура, а основная программа, т.к. это легче реализовать для *переменного* числа параметров (основной программе, в отличие от процедуры, легче определить, сколько параметров записано в стек для конкретного вызова). Плохо в этом случае то, что такое удаление приходится выполнять не в одном месте (при выходе из процедуры), а в *каждой* точке возврата из процедуры, что существенно увеличивает размер кода (в программе может быть очень много вызовов одной и той же процедуры).

```
Summa endp
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, именно так мы и записываем эти параметры в стек. После выполнения команды вызова процедуры `call Summa` стековый кадр имеет вид, показанный на рис. 7.4. После полного формирования стековый кадр будет иметь вид, показанный на рис. 7.5. (справа на этом рисунке показаны смещения слов в стеке относительно значения регистра `bp`).

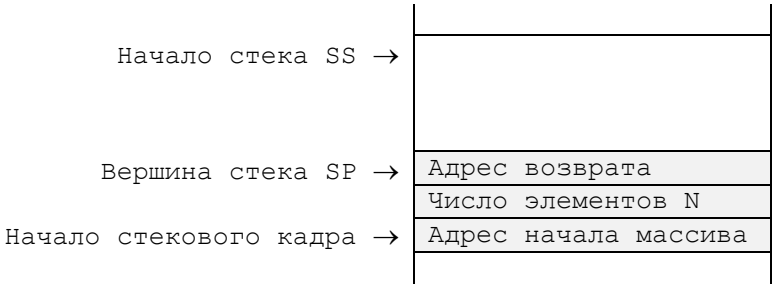


Рис. 7.4. Вид стекового кадра при входе в функцию `Summa`.

Прокомментируем текст нашей программы. 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* для команд формата регистр-память 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

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

а не по адресу

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

как это 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 в качестве базы для доступа к локальным переменным процедуры или функции. Так мы и поступили в нашей программе, поставив регистр `bp` (по сути это ссылочная переменная в смысле Паскаля) примерно на средину стекового кадра. Теперь, отсчитывая смещения от значения регистра `bp` вниз, например `[bp+4]`, мы получаем доступ к фактическим параметрам, а, отсчитывая смещение вверх – доступ к сохранённым значениям регистров и локальным переменным. Например `[bp-6]`является адресом локальной переменной, которую в нашей программе на Ассемблере мы, повторяя программу

на Паскале, назвали именем S (см. рис. 7.5). Теперь понятно, почему регистр bp часто называют *базой* стекового кадра.

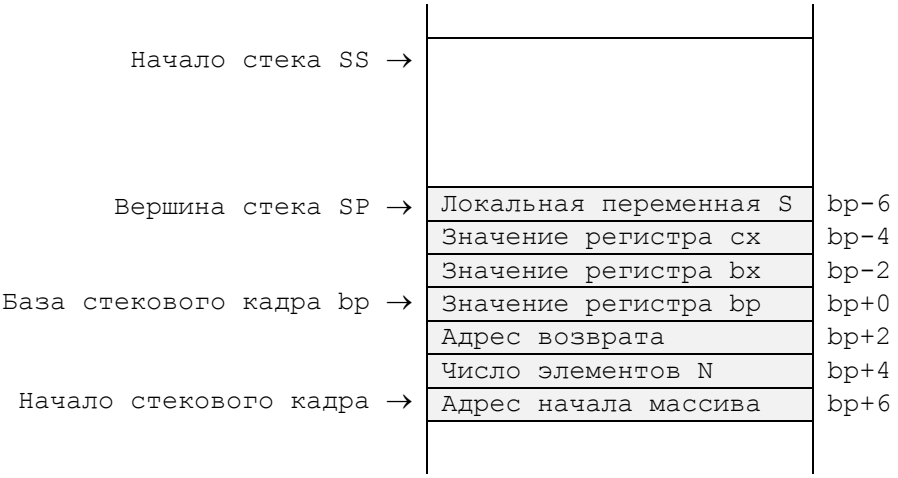


Рис. 7.5. Вид полного стекового кадра (справа показаны смещения слов кадра относительно значения регистра bp).

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

```
S equ word ptr [bp-6]
```

И теперь всюду вместо имени S Ассемблер будет подставлять выражение **word ptr** [bp-6], которое имеет, как нам и нужно, тип слова, расположенного в стеке. Для порождения 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. Эта переменная порождается, как мы и привыкли в языке Паскаль, с *неопределённым* начальным значением. Этой же цели можно было бы достичь, например, более короткой (но менее понятной для читающего программу человека) командой

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

Заметьте, что теперь переменная S порождается уже с начальным значением, равным значению регистра ax, что нам, вообще говоря, было не нужно.

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 (т.е. удалили её из стека), 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 –значений фактических параметров функции. Уничтожение стекового кадра завершено.

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 (т.е. беззнакового) целого числа, при этом будем предполагать, что значение факториала поместится в слово (иначе пусть мы выдадим неправильный результат без диагностики об ошибке). 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;
```

Будем надеяться, что язык Паскаль Вы все хорошо знаете, и без пояснений понимаете, как работает рекурсивная функция ☺. Реализуем теперь эту функцию в виде *близкой* процедуры на Ассемблере. Сначала заметим, что условный оператор Паскаля

```
if N<=1 then Factorial:=1
else Factorial:=N*Factorial(N-1)
```

для программирования на Ассемблере неудобен, так как содержит две ветви (**then** и **else**), которые придётся размещать в линейной структуре машинной программы, что повлечёт использование между этими ветвями команды *безусловного* перехода. Поэтому лучше преобразовать этот оператор в такой эквивалентный вид:

```
Factorial:=1;
if N>1 then Factorial:=N*Factorial(N-1)
```

Теперь приступим к написанию функции `Factorial` 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;    ax*N = Factorial(N-1)*N
Vozv: pop   dx
      pop   bp
      ret   2
Factorial endp
```

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

```
mov   ax,5
push  ax
call  Factorial
outword ax
```

На рис. 7.6 показан вид стека после того, как произведён первый *рекурсивный* вызов функции, заметьте, что 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.

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

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);
```

[1] Это записано на стандарте Паскаля, на языке Турбо-Паскаль так написать нельзя, т.к. массивы `A` и `B` не поместятся в один сегмент данных, а размещать *статические* переменные в разных сегментах Турбо-Паскаль не умеет.

```
   Var i: integer;
Begin Scal:=0; for i:=1 to N do Scal:=Scal+X[i]*Y[i] end;
```

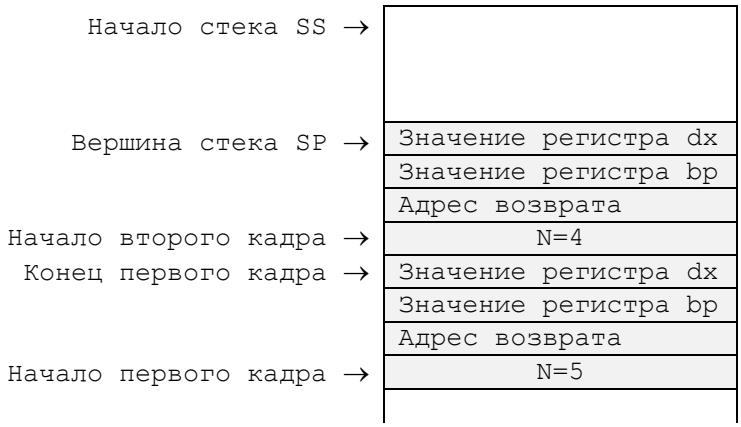


Рис. 7.6. Два стековых кадра функции `Factorial`.

Перед реализацией этой процедуры с именем `SPR` на Ассемблере (со стандартными соглашениями о связях) необходимо решить следующие вопросы. Во-первых, сделаем нашу процедуру *дальней*, чтобы она могла располагаться в любом сегменте памяти нашей программы. Во-вторых, массивы `A` и `B` оба не поместятся в один сегмент данных, поэтому нам придётся описать два сегмента данных и поместить в один из них массив `A`, а в другой сегмент – массив `B`:

```
N     equ   30000
D1    segment
A     dw    N dup (?)
S     dw    ?
D1    ends
D2    segment
B     dw    N dup (?)
D2    ends
```

При передаче таких массивов по ссылке нам придётся заносить в стек *дальний* адрес каждого массива в виде двух чисел `<сегмент,смещение>`. То же самое придётся делать и для передаваемой по ссылке переменной `S`, куда будет помещаться вычисленное значение скалярного произведения. Далее надо решить, как информировать обратившуюся к процедуре основную программу о том, что скалярное произведение не может быть получено правильно, так как не помещается в переменную `S`. Давайте, например, выделим значение $2^{16}-1$ (это знаковое число -1) для случая переполнения результата. Эта проблема является типичной в практике программирования: желательно, чтобы каждая процедура и функция выдавали *код возврата*, который показывает, правильно ли завершилась их работа. Таким образом, значение -1 свидетельствует об ошибке, а все остальные значения переменной `S` будут означать правильное завершение работы нашей процедуры (т.е. *правильное* значение скалярного произведение, равное $2^{16}-1$ мы тоже, к сожалению, объявим *ошибочным*).

Напишем теперь фрагмент программы для вызова процедуры скалярного произведения:

```
mov   ax,D1
push  ax
mov   ax,offset A
push  ax; Полный адрес массива A
mov   ax,D2
push  ax
mov   ax,offset B
push  ax; Полный адрес массива B
mov   ax,N
push  ax; Длина массивов
mov   ax,D1
push  ax
mov   ax,offset S
push  ax; Полный адрес S
```

```
      call  SPR
```

Для такого вызова при входе в процедуру стековый кадр будет иметь вид, показанный на рис. 7.7.

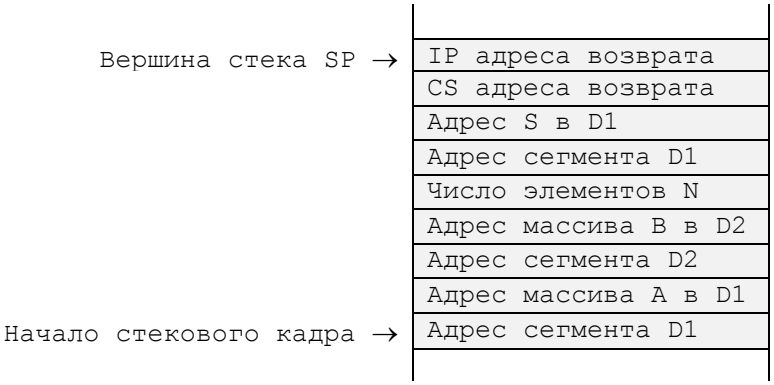


Рис. 7.7. Стековый кадр при входе в процедуру скалярного произведения.

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]; Длина массивов 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 старших моделей нашего семейства ЭВМ, о чём предварительно предупредили Ассемблер директивой **`.186`**. Это сделано только для иллюстрации возможностей старших моделей нашего семейства ЭВМ, в дальнейшем мы не будем использовать такие команды, оставаясь в рамках младшей модели.

На этом мы закончим изучение процедур в языке Ассемблера.

Вопросы и упражнения

1. Напишите *дальнюю* функцию со стандартным соглашением о связях, которая получает как параметр целое беззнаковое число, и вычисляет 32-хбитное значение факториала от этого числа.
2. Обоснуйте, какие из пунктов стандартных соглашений о связях **необходимы** для обеспечения рекурсивных вызовов процедур и функций.

Глава 8. Система прерываний

Далее мы продолжим изучение *переходов*, т.е. нарушений естественного порядка выполнения команд программы. Как мы уже упоминали, некоторые переходы производятся не при выполнении команд программы, а могут делаться центральным процессором *автоматически* при возникновении определённых условий. Если компьютер обладает такими возможностями, то говорят, что в этом компьютере реализована *система прерываний* (иногда эту систему называют ещё *аппаратом* прерываний). Заметим, что самые первые компьютеры такими возможностями не обладали, однако все современные ЭВМ имеют систему прерываний, и сейчас мы начнём изучать, что это такое.

Сначала введём понятие *события* (возникшей ситуации), которое может касаться как выполняемой на ЭВМ программы, так и функционирования компьютера в целом. Эти 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.). Чаше всего бывает так, что при возникновении события продолжать выполнение текущей 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, на сообщение, пришедшее по линии связи от другого компьютера, и т.д.

Не надо думать, что события такого рода в компьютере – это нечто экзотическое, и они происходят редко. Действительно, некоторые из событий (например, деление на ноль) происходят нечасто, в то время как другие (например, события во встроенных в компьютере часах, которые говорит о том, что истёк очередной квант времени) могут происходить несколько сот или даже тысяч раз в секунду.

В архитектуре компьютера предусмотрено, что каждое устройство, в котором произошло событие (центральный процессор, память, устройства ввода/вывода) генерирует при этом особый *сигнал прерывания* – электрический импульс, который приходит на специальную электронную схему центрального процессора. Сигнал прерывания, связанный с каждым из событий, обычно имеет свой *номер*, чтобы центральный процессор мог отличить его от сигналов, связанных с другими событиями.[1] По месту возникновения сигналы прерывания бывают *внутренними* (в центральном процессоре и оперативной памяти) и *внешними* (в периферийных устройствах).

Получив такой сигнал, центральный процессор автоматически (не выполняя никаких команд какой-либо программы!) 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 этого процесса, 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.[2] Здесь, однако, надо рассказать о двух "нетипичных" командах в архитектуре нашего компьютера, это команды с мнемоническими кодами операций **`halt`** и **`wait`**. Команда **`halt`** останавливает выборку команд центральным процессором, и только сигнал прерывания может вывести компьютер из этого "ничегонеделания". Обычно эта команда используется для "выключения" центрального

---

[1] В некоторых компьютерах сигналы о разных событиях приходят на различные входы (линии) электронной схемы анализа прерываний центрального процессора. В этом случае сигнал прерывания дополнительно идентифицируется ещё и номером входной линии, на которую он поступил.

[2] 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.

Необходимо также учитывать, что есть и такие пары команд (мы называли их тесносвязанными) между которыми центральный процессор тоже не "смотрит" на сигналы прерывания. Для нашего компьютера такие тесносвязанные команды порождаются, например, предложением Ассемблера `mov es:[bx],ax`, которое порождает две следующие друг за другом машинные команды `es:` `mov [bx],ax`. (это однобайтная команда смены сегмента и команда пересылки). Скоро мы поймём, почему между такими командами центральному процессору нельзя анализировать сигнал прерывания.

процессора, если ни одна из программ не готова к счёту, это позволяет, в частности, резко снизить энергопотребление центрального процессора, что может оказаться важным, например, при работе компьютера от аккумуляторов. Команда **`wait`** в младшей модели нашего семейства приостанавливает выполнение программы и ждёт окончания команд для операции с вещественными числами, эти команды мы в нашем курсе не рассматриваем.

К описанному выше правилу начала аппаратной реакции на сигнал прерывания по завершению выполнения текущей команды необходимо сделать существенное замечание. Дело в том, что большинство современных ЭВМ отступают от принципа Фон Неймана *последовательного* выполнения команд. Напоминаем, что согласно одному из положений этого принципа, 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). Примерами компьютеров, обладающих такими возможностями, являются так называемые *конвейерными* ЭВМ, они могут одновременно выполнять до восьми и более команд текущей программы. 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 прекращается и в дальнейшем их необходимо повторить *с самого 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. При частом приходе сигналов прерываний это может сильно снизить скорость выполнения программ. 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* времени 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 от внешних устройств по отдельности, для этого надо установить в ноль соответствующий этому прерыванию бит в специальном *регистре маски* прерываний. Таким образом, прерывания с определёнными номерами можно закрывать (маскировать) и открывать (разрешать, снимать с них маску).

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 этот вход центрального процессора ни придёт следующий сигнал прерывания. В последнем случае первый сигнал обычно безвозвратно потеряется, что может быть очень опасно, так как мы не прореагировали должным образом на некоторое событие, и оно прошло для нас незамеченным.[1] 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 пришедший сигнал прерывания не замаскирован, 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 была занята срочной работой, которую нельзя прерывать даже на короткое время. Например, эта программа может обрабатывать некоторое важное событие, 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 неис-

[1] Чтобы снизить такую опасность в архитектуре некоторых ЭВМ, как мы уже упоминали, центральный процессор имеет *несколько* независимых входных линий, на которые могут поступать сигналы прерывания.

правности 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, накопленной на конденсаторах 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, необходимо понять, что бессмысленно маскировать сигналы прерываний, которые выдаются при выполнении некоторых специальных команд, т.к. основным назначением этих команд и является выдача сигнала прерывания (вскоре мы познакомимся с такими командами). 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 запоминанием возврата **`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 находится в двух словах, расположенных в памяти по физическим адресам `4*N` и `4*N+2`. Можно сказать, что для перехода на процедуру-обработчика прерывания с номером `N` необходимо выполнить безусловный переход

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

Таким образом, можно сказать, что прерывание вызывает *дальний прямой косвенный переход*. Заметим, что на самом деле это аналог команды дальнего вызова процедуры, 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 процедура-обработчик *начинает* своё выполнение в режиме запрета таких прерываний. Это гарантирует, что, начав свою работу, процедура-обработчик не будет тут же (после выполнения первой же команды) 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 при работе программ-отладчиков для пошагового

[1] Хотя некоторые программы могут выполняться полностью в режиме закрытых прерываний, но не надо забывать, что существуют прерывания, которые нельзя замаскировать, а также специальные команды, которые *всегда* вызывают прерывания..

выполнения (трассировки) отлаживаемой программы. Надеюсь, что Вы уже имели возможность познакомиться с работой отладчика при выполнении практических заданий на Паскале.[1]

На этом *аппаратная* реакция на незамаскированное прерывание заканчивается.[2] 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 собственно произошло. Так и компьютер по сигналу прерывания может автоматически, "не раздумывая" переключиться на процедуру-обработчика этого сигнала. Отсюда можно сделать вывод, что одно из назначений системы прерываний – обеспечить быструю реакцию компьютера на события, возникающие как в его центральной части, так и на периферии (в устройствах ввода/вывода).

Итак, после завершения аппаратной реакции на незамаскированный сигнал прерывания начинает работать процедура-обработчик прерывания с данным номером, эта процедура выполняет, как говорят, *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 (сигнал), так и компьютер можно "научить", как реагировать на то или иное событие, написав процедуру-обработчика сигнала прерывания с номером, соответствующим этому событию.

Из рассмотренной выше схемы аппаратной реакции на сигнал прерывания можно сделать вывод о том, что наш компьютер будет правильно работать только тогда, когда его вектор прерывания заполнен не случайными числами, а "правильными" адресами процедур-обработчиков прерываний. Об этом должна позаботиться операционная система нашего компьютера в самом начале работы.

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 флага `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 возобновления счёта этой программы с прерванного места. Необходимо запомнить 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* или *контекстом* программы.[3] Заметим также, что обычно процедура-обработчик переключается на свой собственный стек, а не использует далее стек прерванной программы.
- 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 прерывания. Этот сигнал произведёт переключение на какую-то другую процедуру-обработчика, за время работы которой уже может быть нажата

[1] Подробно работа отладчика студенты факультета вычислительной математики и кибернетики МГУ изучают в курсе третьего семестра "Системное программное обеспечение".

[2] Современные компьютеры обычно работают в так называемом мультипрограммном режиме. В конце аппаратной реакции на прерывание такие компьютеры могут дополнительно к вышесказанному переключаться в специальный привилегированный режим работы центрального процессора. Всё это мы будем изучать в конце нашего курса.

[3] Точнее контекстом ***процесса***, процессы студенты факультета вычислительной математики и кибернетики МГУ изучают в курсе третьего семестра "Системное программное обеспечение". Заметим, что в контексте процесса *постоянно* хранится и другая информация, связанная с процессом (объём занимаемой памяти, приоритет, описание открытых файлов и т.п.).

другая клавиша, а информация о нажатой ранее клавише, таким образом, будет безвозвратно потеряна.

После выполнения минимальной программной реакции процедура-обработчик включает (разрешает) внешние прерывания – в нашей архитектуре устанавливает флаг `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 выполнение нашей процедуры-обработчика может быть прервано другим сигналом прерывания.[1]

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 является ещё одним видом команд безусловного перехода и выполняется по схеме:

```
Извстека(IP); Извстека(CS); Извстека(FLAGS)
```

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

Из рассмотренной схемы механизма обработки прерываний можно сделать один важный вывод. Для "обычной" программы прерывание проходит незамеченным, если только эта программа специально не следит за показанием встроенных в компьютер часов (в последнем случае она может заметить, что физическое время её счёта "почему-то" увеличилось). Иногда для подчёркивания этого факта говорят, что механизм прерываний *прозрачен* для выполняемой программы.

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

Таблица 8.1. Начало вектора прерываний

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

Продолжим теперь изучение переходов, вызванных специальными *командами* прерываний, выполнение каждой такой команды в нашей архитектуре *всегда* вызывает прерывание (исключением

---

[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 "принудительный" повторный вход в своё начало (не путать это с рекурсивным вызовом!), называются повторно-входимыми или *реентерабельными*. Для реентерабельных программ при каждом входе в их начало должна резервироваться новая область памяти под хранение контекста этой программы при её прерывании. Программы обработки прерываний для младших моделей нашего семейства могли и не быть реентерабельными, что порождало определённые 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 по времени прерванную программу, а в какую-нибудь другую из этих программ. Поэтому после окончания своей работы процедура-обработчик прерывания обычно передаёт управление специальной системной программе – *диспетчеру*, и уже программа-диспетчер определяет, которая из прерванных программ должна быть продолжена. Обычно каждой выполняемой программе присваивается числовой приоритет, и диспетчер продолжает счёт программы с наибольшим приоритетом. Более подробно всё это Вы будете изучать в следующем семестре.

является команда **into**, которая вызывает прерывание *не всегда*, эту команду мы рассмотрим далее). Каждая команда прерывания реализует *дальний прямой 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 короткая команда **int**, если программа-отладчик реализована в виде обработчика прерывания с номером N=3.

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 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, но выполнение программы продолжается. 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 это 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 от команды **into**. Для простоты эта процедура-обработчик будет *фрагментом* нашей программы и располагаться в её 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, и тем более не выполняет стандартных соглашений о связях.

```
include io.asm
data  segment
A     dw    ?
X     dw    ?
Old   dw    ?
      dw    ?
Diagn db    'Ошибка – большое значение!$'
Data  ends
st    segment stack
      dw    32 dup (?)
st    ends
code  segment
      assume cs:code, ds:data, ss:st
start:mov   ax,data
      mov   ds,ax
; инициализация процедуры-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
      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 обработчика 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 нужных регистров
      push  dx
      sti   ; Открыть прерывания
; --- Полная программная реакция ---
      mov   dx,data
      mov   ds,dx
      mov   dx,offset Diagn
      outstr
      newline
      pop   dx; Восстановление регистров
      pop   ds
      iret  ; Возврат из прерывания
code  ends
      end start
```

Обратите внимание, что при выполнении некоторых групп команд работу нашей программы нельзя прерывать, иначе, например, может возникнуть ситуация, когда в вектор прерывания не занесётся полностью вся необходимая информация (одно машинное слово вместо двух), а уже произойдёт переключение на другую программу по пришедшему сигналу прерывания. В программировании такие группы команд называются **критическими секциями** программы, в показанной выше программе критические секции заключены в рамки. Мы позаботились о том, чтобы при выполнении каждой критической секции работа не была прервана, для этого в начале каждой критической секции стоит команда запрета прерывания от внешних устройств **`cli`**, а в конце секции – команда открытия таких прерываний **`sti`**.

После выдачи диагностики об ошибке наша процедура-обработчик может не продолжать выполнение программы, а, например, завершать выполнение этой 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 языка нашей машины, которая всегда вызывает прерывание с номером N, заданным в качестве её операнда:

```
      int   op1
```

Здесь op1=N имеет формат i8. По своей сути это команда дальнего абсолютного косвенного перехода, которая выполняется так же, как команда `jmp dword ptr [4*op1]`. Пожалуй, единственным отличием здесь является то, что команде **int** для доступа к памяти *не требуется* сегментный регистр данных, так как она всегда обращается в фиксированный сегмент (вектор прерывания), расположенный в самом начале оперативной памяти.

Заметим, что с помощью команды **int** можно вызвать прерывание с любым номером, например прерывание, соответствующее делению на ноль или плохому коду операции. Более того, прерывания с номерами, большими 31_{10}, в нашей архитектуре можно вызвать, *только* выполняя команду `int op1` 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). Естественно желание реализовать такие стандартные для большинства программ действия в виде некоторого *набора процедур*, чтобы не записывать такие действия в каждую программу, а просто вызывать необходимые процедуры. 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 (например, при их модификации), то лучше всего присвоить каждой такой процедуре свой *номер* N и оформлять такие процедуры в виде обработчиков прерываний с этим номером. Это освобождает остальные программы от необходимости знать месторасположение таких процедур в памяти, достаточно знать только их *номера*. 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 соответствующую библиотеку стандартных процедур – Базовой системой процедур ввода/вывода (английское сокращение BIOS – Base Input/Output System). Параметры для таких процедур обычно передаются на регистрах, т.е. для системных вызовов не выполняются стандартные соглашения о связях.[1]

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 (код операции) на регистре ah (для вывода строки ah=9) и адрес начала выводимой строки на регистрах <DS:DX> (выводимая строка должна кончаться символом '$'). Исходя из этого на место нашей *макрокоманды* вывода строки текста

```
outstr
```

можно подставить *команды* языка машины

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

В качестве примера опишем на Ассемблере процедуру без параметров, использующую системный вызов. Эта процедура при её выполнении выдаёт короткий звуковой сигнал на встроенный в компьютер динамик:

```
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
```

[1] Первые BIOS появились, когда ещё не существовало многоязыковых систем программирования, и не было необходимости в стандартных соглашениях о связях. Заметим также, что вызов с нестандартными соглашениями о связях производится быстрее. При этом, однако, многие вещи становятся невозможными, например, рекурсивный вызов таких процедур.

```
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 периода 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.

Процедуры-демоны широко используются в современных вычислительных системах, например, при приходе электронного письма на сервер вызывается системная процедура – почтовый демон и т.д. Заметим, что в некоторые языки высокого уровня включены аналогичные возможности, например, в языке С можно писать так называемые функции-реакции на сигналы.[1]

В заключение нашего по необходимости краткого рассмотрения прерываний заметим, что появление в компьютерах системы прерываний было, несомненно, одним из важнейших событий в развитии архитектуры вычислительных машин. Вспомним, что ничего подобного в нашей учебной машине не было. Там при возникновении аварийных ситуаций в центральной части машины (деление на ноль, команда с несуществующим кодом операции и т.д.) просто происходил останов машины, а в устройствах ввода/вывода не происходило ничего такого, что бы могло "заинтересовать" центральный процессор.[2]

Как мы вскоре узнаем, появление системы прерываний было одним из необходимых условий, обеспечивающим работу ЭВМ в так называемом мультипрограммном режиме. Недаром появившиеся компьютеры с системой прерываний (и некоторыми другими важными аппаратными особенностями, о которых мы будем говорить далее) стали относить к следующему, третьему поколению ЭВМ. Подробнее об этом можно прочитать в книгах [1,3].

---

[1] Функции-реакции на сигналы в операционной системе Unix студенты факультета вычислительной математики и кибернетики МГУ изучают в курсе третьего семестра "Системное программное обеспечение".

[2] Точнее, если в устройствах ввода/вывода ЭВМ такой архитектуры происходила аварийная ситуация, например, сбой в работе, то устройство управления просто останавливало машину, и для возобновления работы требовалось вмешательство человека-оператора.

Глава 9. Дополнительные возможности Ассемблера

9.1. Строковые команды

Сейчас мы рассмотрим последний из ещё нерассмотренных нами и весьма 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. Например, для присваивания переменной Y значения переменной X вместо команды пересылки формата память-память

```
mov Y,X
```

нам приходилось использовать две команды пересылки, например

```
mov ax,X
mov Y,ax
```

Это не всегда практично, 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 команды формата память-память, в некоторых случаях они позволяют получать более компактные и быстрые программы. В архитектуре нашего компьютера такие команды относятся к так называемым строковым или цепочечным командам (мы скоро поймём, почему они получили такое название).

Строковые команды нашего компьютера хорошо использовать для обработки элементов массивов. Массивы коротких беззнаковых целых чисел могут трактоваться как строки (цепочки) символов, отсюда и название – *строковые* или *цепочечные* команды. Можно считать, что каждая строковая команда обрабатывает *один* элемент такого массива. При выполнении строковых команда в цикле получается удобный способ обработки массивов, элементами которых являются короткие или длинные целые числа, расположенные в оперативной памяти.

Знакомство со строковыми командами начнём с команд пересылки байта (**movsb**) или слова (**movsw**) с одного места оперативной памяти в другое. Эти команды различаются только битом размера операнда w в коде операции. С битом w мы уже 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).

Алгоритм выполнение команд **movsb** и **movsw** существенно зависит от так называемого флага направления DF из регистра флагов FLAGS. Для изменения значения этого флага можно использовать команды **cld** (для операции очистки флага DF:=0), и **std** (для операции установки флага DF:=1). Чтобы описания правил выполнения команд **movsb** и **movsw** были более компактными и строгими, введём следующие условные обозначения:

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

Как видим, **δ** может принимать только значения ±1 и ±2, а оператор **φ** меняет величину заданного регистра r16 на значение **δ**. В этих обозначениях команды **movsb** и **movsw** выполняются по следующему правилу:

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

Таким образом, неявный операнд op1 находится в памяти по адресу, заданному регистровой парой <es,di>, а неявный операнд op2 – по адресу <ds,si>, т.е. в общем случае операнды находятся в *разных* сегментах памяти.

---

[1] Точнее, лишним будет только использование *адресуемого* регистра центрального процессора, их в распоряжении программиста и так немного (ax,bx и т.д.). Однако, как мы знаем, двухадресные команды формата память-память КОП op1,op2 обязательно требуют для своего выполнения использования некоторых *служебных* регистров центрального процессора (напомним, что мы обозначали эти регистры как R1, R2 и S), и выполняются по схеме: R1:=op1; R2:=op2; S:=R1 КОП R2; op1:=S

Для того, чтобы лучше понять логику работы описанных выше команд, рассмотрим широко распространённую задачу пересылки массива целых чисел с одного места оперативной памяти в другое. На языке Паскаль такая задача решается совсем 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 в приведённом примере массивы A и B будут располагаться в *одном* сегменте данных. Более общий случай пересылки массива на стандарте Паскале можно реализовать (для массива символов), например, в таком виде:

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

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 A и B:

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

Операция копирования требует *одновременного* доступа к этим двум сегментам памяти, поэтому на начало сегмента D1 установим сегментный регистр ds, а на начало сегмента D2 – сегментный регистр es. Длину массива обозначим N, в частном случае длина массива может быть и нулевой (вспомним, какую важную роль в программах на Турбо-Паскале играли строки символов нулевой длины). Теперь фрагмент программы для реализации оператора присваивания B:=A может, например, иметь на Ассемблере 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
      jcxz  L1; при N=0
L:    mov   al,[si]
      mov   es:[di],al
      inc   si
      inc   di
      loop  L
```

```
L1:   . . .
```

Оценим сложность нашего алгоритма пересылки массива. За единицу измерения сложности примем операцию обмена данными или командами между центральным процессором и оперативной памятью. В нашем случае сложность алгоритма пересылки массива равна `7*N`, где `N` – это длина массива. Действительно, необходимо `N` чтений элементов массива из памяти на регистр `al`, `N` записей из этого регистра в память, и ещё нужно `5*N` раз считать команды тела цикла из памяти на регистр команд в устройстве 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 предыдущей программы можно более компактно записать в виде:

```
        jcxz  L1; при N=0
        cld
 L:     movsb
        loop  L
```

Теперь в нашем цикле пересылки массива осталось всего две команды, следовательно, сложность нашего алгоритма снизилась до `4*N` операций обмена с оперативной памятью. Для дальнейшего ускорения выполнения таких циклов в язык машины была включена специальная команда цикла с кодом операции **`rep`**, которая называется *префиксом повторения*. Она похожа на команду цикла **`loop`**, но не имеет явного операнда – метки перехода на начало тела 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 [1]

```
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 в программах его часто пишут на месте метки (в качестве первого поля предложения Ассемблера), так как служебное слово нельзя спутать с именем метки, заданным пользователем. Пара команд `rep` `movsb` является тесно связанной, они *вместе* выбираются на регистр команд центрального процессора, так что теперь в цикле пересылки массива нет необходимости обращаться в память *за командами*.[2] Теперь сложность нашего алгоритма снизилась до теоретического минимума в `2*N` операций, т.е. это позво-

---

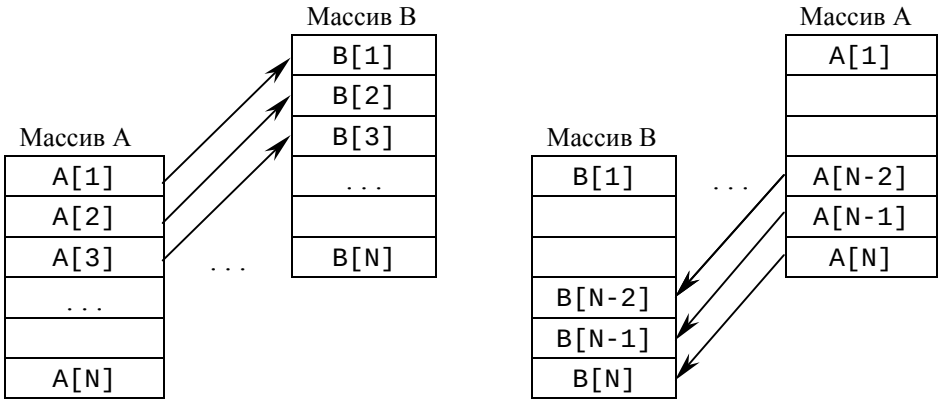
[1] Если за командой **`rep`** следует не строковая команда, а какая-нибудь другая, то команда **`rep`** просто игнорируется (выполняется так же, как и пустая команда с кодом операции **`nop`**, которая в нашей машине эквивалентна пустому оператору языка Паскаль).

[2] Для продвинутых студентов: так как эти команды являются тесно связанными, то между ними запрещены прерывания. Следовательно, при возникновении прерывания регистр `IP` (счётчик адреса) должен указывать на команду **`rep`**, так как на каждом шаге цикла она всегда будет выполняться *следующей*. Таким образом, если происходит прерывание, выполнение цикла пересылки массива прерывается, а при возврате из прерывания возобновляется с прерванного места (с команды **`rep`**). Это следует из того, что при прерывании запоминаются, а потом при возврате в программу восстанавливаются значения всех регистров, используемых для правильной работы этого цикла (перечислите все эти регистры, их 8 штук!).

лило значительно поднять эффективность пересылки массива.[1] Теперь Вам должно быть понятно, для чего цепочечные команды введены в язык нашего компьютера, хотя они и являются избыточными (как мы видели, пересылку массива можно делать и без использования этих команд).

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 – Direction Flag (флаг направления пересылки массива).

Пересылка массива в обратном направлении – не прихоть программиста, а часто единственный способ *правильного* присвоения значения одного массива другому. Правда следует сразу сказать, что флаг направления влияет на правильность присваивания 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 `A` и `B` в памяти при выполнении пересылок. 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 только при *обратном* направлении пересылки элементов массива с флагом `DF=1`. Обязательно поймите, что пересылка перекрывающихся массивов не в том направлении приведёт к ошибке.



а). Должно быть `DF=0` б). Должно быть `DF=1`

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

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

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)`

[1] Такое значительное уменьшение сложности алгоритма пересылки массива при использовании цепочечных команд 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 (команды цикла будут читаться из кэш-памяти), и сложность алгоритма также приближается к `2*N` обменов. Более того, современные компьютеры за одно обращение к памяти читают на один байт, а сразу несколько (обычно 8 или 16) подряд расположенных байт. Таким образом, получается, что сложность нашего алгоритма на современных компьютерах падает уже до `N/4` обменов. Память типа кэш студенты факультета вычислительной математики и кибернетики МГУ изучают в курсе третьего семестра "Системное программное обеспечение".

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

Как мы знаем, команды сравнения обычно необходимы для правильной работы команд условного перехода. С командами сравнения **cmpsb** и **cmpsw** удобно использовать команды-префиксы повторения **repe/repz** (выполнять цикл, пока сравниваемые аргументы равны, т.е. флаг ZF=1) и **repne/repnz** (выполнять цикл, пока сравниваемые аргументы *не* равны, т.е. ZF=0). Эти команды похожи на рассмотренную ранее 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 во многих языках программирования высокого уровня 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 реализовать на Ассемблере таким образом (правда, здесь строки должны иметь одинаковую длину, сравнение строк разной длины реализуется несколько 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; Надо N>0 !
      cld
      lea   si,X
      lea   di,Y
repe  cmpsb
      je    EQ; Строки X и Y равны
      jb    LT; Строка X меньше Y
      ja    GT; Срока X больше Y
```

В нашем примере сравниваемые строки для простоты расположены в одном сегменте (сегменте данных), однако, как мы знаем, для строковых команд это не принципиально. 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**.[1]

Остальные строковые команды имеют формат регистр-память, но они тоже ориентированы на задачи обработки элементов строк (массивов) коротких и длинных целых чисел. Команды *сканирования 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 имеют 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 в оперативной памяти. Если для краткости обозначить буквой r регистр al или ax, то схему выполнения команд сканирования строки можно записать так:

```
      cmp r,<es,di>; φ(di)
```

[1] В алгоритме работы этой программы есть небольшая тонкость, она правильно работает только при длине строк N>0. Дело в том, что команда **repe** определяет цикл с предусловием, поэтому при N=0 команда сравнения **cmpsb** не выполняется ни разу, поэтому значение флага нуля ZF не определено и алгоритм может и не перейти на метку EQ. В то же время часто бывает полезно сравнивать между собой и пустые строки, как в типе String языка Турбо-Паскаль (две пустые строки, конечно же, равны между собой). Для обеспечения такой возможности можно перед командой **cld** поставить, например, команду **cmp cx,cx**, которая установит в единицу флаг ZF.

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

```
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; Адрес последнего элемента
      std   ; Обратный просмотр эдементов
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 (**`lodsb`**) или длинных (**`lodsw`**) 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 в оперативной памяти. Если обозначить буквой `r` регистр `al` или `ax`, то схему выполнения этих команд можно записать так:

`mov` `r,<ds,si>; φ(si)`

Ясно, что эту команду имеет смысл использовать только совместно с командами сравнения и условного перехода. В качестве примера использования команды загрузки напишем 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. Для упрощения алгоритма переполнение самого регистра dx не проверяется.

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 (команда **stosb**) или длинного (команда **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 в оперативной памяти. Если обозначить буквой r регистр al или ax, то схему выполнения команд можно записать так:

```
      mov <es,di>,r; φ(di)
```

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

```
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 массива
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:=abs(X)(учтите, что подобный оператор присваивания – это только иллюстрация условия задачи, так в Паскале для масссивов 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, напомним, что для хорошего освоения этой темы необходимо использовать рекомендованные Вам учебник и задачник по языку Ассемблера. А мы теперь перейдём к следующему классу команд – логическим командам.

Упражнение. Определите, какие значения должен иметь флаг направления DF при операции *вставки* и при операции *удаления* участка редактируемого текста в некотором текстовом редакторе, который использует для этих целей строковые команды.

9.2. Логические команды

Все логические команды нашего компьютера 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:

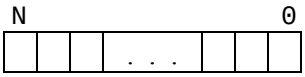
КОП op1,op2

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, справа-налево от нуля до некоторого числа N. Число N равно 7 для коротких (байтовых) операндов и равно 15 для длинных операндов (размером в слово):



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

[1] Для двух команд из этого класса (для двух команд сдвига) в этом наборе будет 9 или 17 бит, это мы далее отметим особо.

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

Здесь мы не принимаем во внимание, что номера битов в логическом операнде, в отличие от индексов элементов массива в Паскале, идут в обратном порядке. Конечно, учитывая нумерацию битов в словах и байтах справа-налево, более правильным было бы описать эти массивы на Паскале как

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

Однако на Паскале так записать невозможно, а для нашего дальнейшего рассмотрения это отличие в нумерации не принципиально.

Рассмотрим теперь, команды с какими кодами операций являются логическими командами. Схему выполнения команды *логического умножения*

```
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 побитового сложения по модулю 2, что соответствует алгоритму выполнения этой команды)

```
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.

И, наконец, рассмотрим одноадресную команду логического отрицания

```
not op1
```

где op1 имеет такие же допустимые форматы r8,m8,r16 и m16, как и первый операнд всех рассмотренных выше логических команд. Схема выполнения этой команды на Паскале можно записать как

```
for i:=0 to N do op1[i]:=not op1[i]
```

Заметим, что команда логического отрицания `not op1` даёт абсолютно тот же результат, что и команда `xor op1,-1`, то есть, является *избыточной* в языке машины, однако её достоинством является то, что она примерно в два раза короче.

Все перечисленные выше логические команды устанавливают флаги так же, как команда вычитания, например, флаги CF и OF будут всегда равны нулю (никакого переноса и переполнения, конечно же 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 **for** языка Паскаль только для иллюстрации и более строгого описания работы логических команд. Не следует понимать это слишком буквально: на самом деле логические команды на современных ЭВМ выполняют операции над всеми битами своих операндов одновременно (чаще всего очень быстро – за один или два такта работы центрального процессора), а совсем не в последовательном цикле.[1]

---

[1] Для любознательных студентов отметим, что для описания работы логических команд более подходит оператор цикла какого-нибудь диалекта параллельного Фортрана, например команды логического умножения:

```
for all i in [1..N] do op1[i]:=op1[i] and op2[i]
```

Этот оператор предписывает выполнить тело цикла *одновременно* (параллельно) для *всех* значений параметра цикла i.

9.3. Команды сдвига

Команды логические 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 до этого логических команд. Команды сдвига имеют формат

```
кол op1,op2
```

где op1 может быть r8,m8,r16 или m16, а op2 – иметь значение единицы или быть коротким регистром cl.[1] Мы рассмотрим сначала команды сдвига вида `кол op1,1`, а затем обобщим их на случай команд вида `кол op1,cl`.

Команда сдвига операнда на один бит *влево* имеет показанный ниже вид

```
shl op1,1
```

и её выполнение можно так описать в виде операторов на Паскале:

```
CF:=op1[N];
for i:=N downto 1 do op1[i]:=op1[i-1];
op1[0]:=0
```

Ниже на рис. 9.2 показана схема выполнения этой команды.

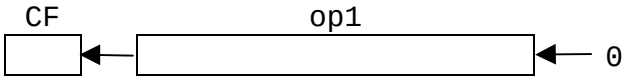


Рис. 9.2. Схема выполнения команды **shl** op1,1.

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 присваивается 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 флаг нуля ZF (он устанавливается в единицу, если, как обычно, получается полностью нулевой вектор-результат).

У названия кода операции сдвига влево **shl** (она называется *логическим* сдвигом влево), 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] (как мы знаем, в этом случае флаг переполнения OF=0). Таким образом, правильность умножения на два с помощью команды сдвига влево можно контролировать, как обычно, анализируя флаги переноса и переполнения для беззнаковых и знаковых чисел соответственно.

Заметим, что беззнаковое и знаковое умножение в нашем компьютере делается *разными* командами **mul** и **imul**, однако умножении на 2 с помощью команды сдвига влево делается по одному алгоритму как для знаковых, так и для беззнаковых чисел. Именно поэтому одна команда логического сдвига влево имеет два имени для своего кода операции: логический (т.е. беззнаковый) сдвиг **shl** и арифметический (т.е. знаковый) сдвиг **sal**.

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

---

[1] В старших моделях нашего семейства у команд сдвига дополнительно допускается также второй операнд формата i8. Мы в своих примерах этой возможностью, как и договаривались, пользоваться не будем.

```
shr op1,1
```

выполняется по правилу, которое можно так описать на Паскале:

```
CF:=op1[0];
for i:=0 to N-1 do op1[i]:=op1[i+1];
op1[N]:=0
```

На рис. 9.3 показана схема выполнения этой команды.

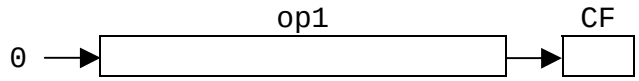


Рис. 9.3. Схема выполнения команды **shr** op1,1.

Команда *логического* сдвига на один разряд вправо эквивалентна делению на два *беззнакового* целого числа, результат при этом всегда получается правильным. Делению на два *знакового* целого числа в определённом смысле 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]
```

Ниже на рис. 9.4 показана схема выполнения этой команды.

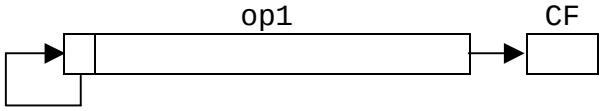


Рис. 9.4. Схема выполнения команды **sar** op1,1.

Как видим, крайний левый бит аргумента при арифметическом сдвиге вправо *не меняется*. При трактовке операнда сдвига как знакового числа простой анализ показывает, что для неотрицательных и отрицательных чётных значений арифметический сдвиг вправо всегда эквивалентен операции деления этого аргумента на 2. В то же время для отрицательных нечётных значений это неверно, например, (-5) div 2 = (-2), а sar (-5),1 = (-3), т.е. на единицу меньше. Таким образом, правильное деление на 2 сдвигом любой знаковой величины op1 реализует, например, такой фрагмент программы:

```
    sar op1,1
    jns L
    adc op1,0
L:
```

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.[1]
```

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 такими 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 сохраня-

[1] Как видим, есть только небольшое различие от команды знакового деления на два с кодом операции **idiv**, так как остаток от деления равных по модулю делимых не различается.

ет все свои биты, меняются только номера этих битов внутри операнда. Команда циклического сдвига *влево*

```
rol op1,1
```

выполняется по правилу

```
shl op1,1; op1[0]:=CF
```

Другими словами, сначала выполняется команда логического сдвига влево, а потом в результате корректируется значение нулевого бита. Ниже на рис. 9.5 показана схема выполнения этой команды.

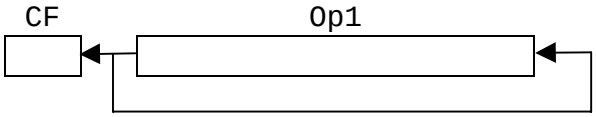


Рис. 9.5. Схема выполнения команды **rol** op1,1.

Команда циклического сдвига *вправо*

```
ror op1,1
```

выполняется по правилу

```
shr op1,1; op1[N]:=CF
```

Ниже на рис 9.6 показана схема выполнения этой команды.

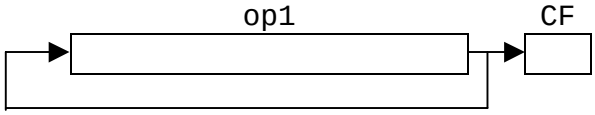


Рис. 9.6. Схема выполнения команды **ror** op1,1.

Команды 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

```
temp:=CF; shl op1,1; op1[0]:=temp
```

Здесь t – некоторая вспомогательная (вре́менная) переменная. На рис. 9.7 показана схема выполнения этой команды.

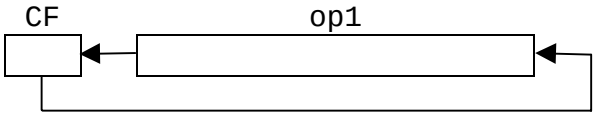


Рис. 9.7. Схема выполнения команды **rcl** op1,1.

Команда циклического сдвига *вправо через флаг переноса*

```
rcr op1,1
```

выполняется по правилу

```
temp:=CF; shr op1,1; op1[N]:=temp
```

Ниже на рис. 9.8 показана схема выполнения этой команды.

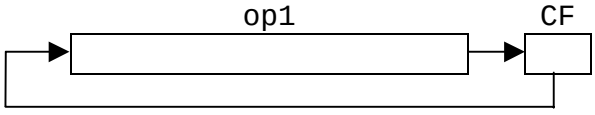


Рис. 9.8. Схема выполнения команды **rcr** op1,1.

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 cl. Каждая такая команда (**КОП** – любой из кодов операций команд сдвигов)

```
КОП op1,cl
```

выполняется по правилу

```
temp:=cx;
```

```
      for cl:=cl downto 1 do кол op1,1;
      cx:=temp
```

Таким образом, значение регистра `cl` задаёт число разрядов, на которые *в цикле* происходит сдвиг операнда, при этом, как видно из описания работы, сам регистр `cx` такими командами изменить нельзя. Кроме того, ясно, что задавать сдвиги операнда более чем на `N` разрядов (длину операнда в битах) не имеет большого смысла.

Главное назначение логических команд – обрабатывать отдельные биты и группы битов в байтах и словах. Как мы знаем, минимальной порцией данных, отрабатываемой в командах, являются в нашей архитектуре байт, поэтому в языке машины и предусмотрены логические команды.[1]

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. Без использования логических команд это можно сделать, например, с помощью такого фрагмента программы на Ассемблере:

```
X     dw    ?
      . . .
      mov   ax,X
      sub   cx,cx; число "1"
      mov   bx,2
L:    cmp   ax,0
      jz    Pech
      mov   dx,0
      div   bx
      adc   cx,0; cx:=cx+CF
      jmp   L
Pech: outint cx
```

А теперь решим эту же задачу с использованием логических команд:

```
X     dw    ?
      . . .
      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
```

Этот алгоритм будет работать быстрее за счёт того, что медленная каманда деления заменяется на быструю команду сдвига, кроме того, теперь мы использовали на два регистра меньше. Заметим, что операция подсчёта числа битов машинного слова со значением `"1"` являлась 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 номеру сигнала прерывания). В этой архитектуре существовала специальная машинная команда для подсчёта количества `"1"` в своём аргументе, что позволяло быстро определить число ещё необработанных сигналов прерывания.

---

[1] По большому счёту все логические команды тоже являются *избыточными* в языке нашей машины, действия логических команд всегда можно заменить (довольно большими) фрагментами программ на Ассемблере, где будут использоваться команды целочисленного деления и умножения. В качестве хорошего упражнения реализуйте без использования логических команд фрагмент программы, эквивалентный, например, оператору `z:=x and y`, где `x`,`y` и `z` – переменные длиной в слово в сегменте данных.

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-1] 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 рекомендациям (игнорировать их), в частности, так поступает и наш Турбо-Паскаль.

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
      . . .
; реализация оператора X:=A[i]
      mov   bx,i;
      xor   ax,ax; ax:=0
      shr   bx,1; беззнаковое деление на 2
      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
```

Прокомментируем эту программу. Сначала мы командой сдвига разделили значение индекса i (это беззнакровое число!) 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-ми, 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), называ*ется 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. Заметим также, что самый большой выигрыш по памяти получается при обработке упакованных массивов логических величин, например:

```
var Z: packed array[1..100000] of boolean;
```

В качестве ещё одного 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 кодом (номером) i присутствует во множестве, иначе этот бит имеет значение 0 (заметим, что язык Турбо-Паскаль реализует такое же представление для переменных типа `set of char`). 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
```

Обратите внимание на такое обстоятельство. Несмотря на то, что множества X и Y из последнего примера являются *упакованными* структурами данных (упакованными логическими векторами из 256 элементов), при работе с ними происходит не замедление, а *увеличение* (в нашем случае в 16 раз) скорости работы, по сравнению с неупакованными данными. Как мы понимает, такое увеличение скорости происходит из-за параллельной обработки битов своих операндов логической командой **and**.

В заключение рассмотрим условный оператор языка Паскаля, включающий знакомую нам по прошлому курсу операцию отношения **in** (символ Sym входит во множество X):

```
if Sym in X then goto L;
```

На Ассемблере этот оператор можно, например, реализовать с использованием логических команд 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 битом символа 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
```

Разберём, как работает эта программа. Сначала, используя команду логического сдвига на `3` (это, как мы знаем, аналогично делению беззнакового числа на 8), на регистр `bx` заносим индекс того байта в массиве `X`, в котором находится бит, соответствующий символу `Sym` во 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 (биты в байте при этом нумеруются, как нам и нужно, *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 **`jc`** по значению этого флага.

На этом мы закончим рассмотрение логических команд.

Упражнение. Для описания упакованного массива данных

```
N     equ   10000
Data  segment
; var A: packed array[1..N] of 0..15;
A     db    N/2 dup (?); N/2 ≡ N div 2
X     db    ?
```

Реализуйте оператор присваивания `A[i]:=X`.

9.4. Упакованные битовые поля

Одним из применений логических команд и команд сдвигов, как мы выяснили, является работа с частями байтов и слов (т.е. составляющими их битами и наборами битов). Для автоматизации работы с такими данными Ассемблер предоставляет в распоряжение программиста определённые языковые средства. К таким средствам, в частности, относятся так называемые упакованные битовые поля (**`record`**). На примере упакованных битовых полей мы познакомимся со способами описания *типов* в языке Ассемблера.

До сих пор для работы с переменными в Ассемблере мы использовали только предложения резервирования памяти, которые являются аналогами описания переменных в языках высокого уровня. Теперь познакомимся с Ассемблерными аналогами описания *типов* данных. Использование типов данных в Ассемблере, как и во всяком алгоритмическом языке, повышает надёжность программирования и делает программу лучше для понимания и модификации.

Битовым полем в Ассемблере называется последовательный набор битов в байте или слове, причём каждому такому битовому полю пользователь присваивает имя. В одном байте или слове можно определить несколько битовых полей, если их общая длина не превышает, соответственно, размера байта или слова. Так организованные данные и называются *упакованными битовыми полями* (имеется в виду, что эти поля плотно упакованы, т.е. примыкают друг к другу, внутри байта или машинного слова).

В Ассемблере для упакованных битовых полей предусмотрено описание типа. В качестве примера рассмотрим описание трёх битовых полей, упакованных в одно машинное слово. Присвоим этому типу данных имя `date`, что будет отражать суть трёх входящих в этот тип битовых полей, предназначенных для хранения некоторой даты (числа, месяца и двух последних цифр года):

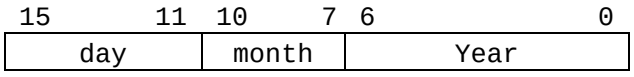
```
date  record day:5,month:4,year:7=4
```

В этом описании типа указано, что переменная с именем типа `date` для своего хранения потребует одно машинное слово (два байта), в котором будут содержаться битовые поля с именами `day` (длиной 5 бит), `month` (4 бита) и `year` (7 битов). Заметим, что некоторым аналогом этого описания в языке Паскаль будет, например, такое описание типа (при условии, что Паскаль-машина "прислушается" к нашей рекомендации использовать именно упакованную структуру дпнных):

```
type date = packed record
     day: 0..31; month: 0..15; year: 0..127
end;
```

Упакованные битовые поля располагаются в машинном слове в порядке их описания, но выравниваются по *правому* краю машинного слова или байта, если сумма длин битовых полей меньше

длины слова (или байта). Ниже показан вид переменной типа `date` с указанием номеров битов машинного слова, занимаемыми битовыми полями:



Подчеркнём, что имя `date` является именем *типа,* и показанное выше предложение Ассемблера не резервирует в памяти никакой области данных (никакой переменной), то есть, является с точки зрения языка Ассемблер *директивой*. Само же резервирование областей памяти для хранения переменных типа `date` производится в некотором сегменте по предложениям резервирования памяти. Ниже показаны примеры таких предложений резервирования памяти для хранения переменных типа `date`, в комментариях показаны начальные значения полей.

			Day	**month**	**year**
dt1	date	<,8>;	0	8	4
dt2	date	<21,4,96>;	21	4	96
dt3	date	<>;	0	0	4
dt4	date	<13,5,?>;	13	5	0

Обратите внимание, что переменные типа упакованных битовых полей *всегда* порождаются с начальными значениями, которые либо задаются программистом в параметрах предложения резервирования памяти, либо берутся из описания данного типа, в остальных случаях они по умолчанию считаются равными нулю. В нашем примере описания типа `date` мы указали, что битовое поле с именем `year` должно иметь начальное значение по умолчанию `4`, если при порождении такой переменной этому полю *явно* не будет задано другое значение. В директивах резервирования памяти начальные значения битовых полей перечисляются через запятые в угловых скобках. Заметьте также, что символ `?` здесь при резервировании памяти задаёт *не неопределённое*, как в других случаях, а *нулевое* начальное значение соответствующего упакованного битового поля.

Языковые средства, которые Ассемблер предоставляет для работы с упакованными битовыми полями, разберём на следующем примере. Напишем фрагмент программы, в котором берётся значение переменной с именем `X` типа `date`, и хранящаяся в этой переменной дата выводится на экран в привычном виде, например, как `день/месяц/год`. Так как для значения года хранятся только две последние десятичные цифры, то здесь у нас тоже возникает своя "проблема 2000 года". Мы сделаем следующую спецификацию нашей задачи: если значение года больше 50, то будем считать дату принадлежащей *прошлому* веку, иначе – *текущему* веку.[1] Ниже приведён фрагмент программы, решающий эту задачу.

```
mov   ax,X
mov   cl,day; cl=11 (№ позиции поля day)
shr   ax,cl; ax:=Только поле день
outword ax
outch '/'
mov   ax,X
and   ax,mask month; mask month=0000011110000000b
mov   cl,month; cl=7 (№ позиции поля month)
shr   ax,cl; ax:=Только поле месяц
outword ax
outch '/'
mov   ax,X
and   ax,mask year; mask year=0000000001111111b
mov   bx,2000; нынешний век
cmp   ax,50
jbe   L
```

[1] Такая спецификация сделана исключительно для учебных целей, она проста для понимания и использования, но допускает хранение некоторых дат в двух форматах. Например, даты, записанные как `01/01/110` и `01/01/10` будет соответствовать одной "настоящей" дате `01/01/2010`. Это происходит из-за того, что из всего возможного диапазана хранимых в таком виде 128 лет (`0–127`) мы использовали только диапазон в 100 лет.

```
      mov   bx,1900; прошлый век
L:    add   ax,bx; ax:=Год из 4-х цифр
      outword ax
      newline
```

Прокомментируем этот фрагмент программы. Как мы знаем, почти всем именам, которые программист записывает в предложениях Ассемблера, приписываются определённые *числовые* значения, которые Ассемблер и *подставляет* на место этого имени в программу. Так, значением имён переменных и имён меток являются их адреса – смещения относительно начала сегмента,[1] значением имён целочисленных констант, определяемых директивами эквивалентности – сами эти константы, значениями имён сегментов являются адреса начал этих сегментов в оперативной памяти, делённые на 16 и т.д. Отметим, что некоторое имена могут вооще не иметь *числовых* значений, например, это, например, вот такое имя S, заданное директивой

```
S equ [bp+6]
```

Имя упакованного битового поля тоже имеет значение – это количество разрядов, на которое надо сдвинуть это поле, чтобы оно попало в самую правую позицию той области памяти (байта или слова) в которой хранится данное поле. Например, имя `month` имеет значение `7`. Таким образом, имена упакованных битовых полей в Ассемблере предназначены в программе для задания числа сдвигов слова или байта, что мы и использовали в нашем примере.

Другое языковое средство, предоставляемое Ассемблером для работы с упакованными битовыми полями – это одноместный оператор, который задаётся служебным именем **`mask`**. Операндом этого оператора должно являться имя упакованного битового поля, а значением данного оператора будет байт или слово, содержащее биты `"1"` только в позициях, занимаемых указанным полем, в остальных позициях будут находиться нулевые биты. Как видим, оператор **`mask`** удобно использовать для задания констант (масок) в командах логического умножения для выделения нужного битового поля, это и было продемонстрировано в нашем примере.

И, наконец, рассмотрим ещё один оператор Ассемблера, предназначенный для работы с именами упакованных битовых полей. Значением оператора

`width` <имя битового поля>

является ширина этого поля, т.е. количество составляющих его бит. Например, для нашего описания типа `date` выражение **`width`** `month` равно `4`. Заметим, что если применить этот оператор к имени *типа*, задающего упакованные битовые поля, то значением оператора **`width`** будет сумма длин всех полей в этом типе. Например, **`width`** `date` равно `16`. Оператор **`width`** достаточно редко встречается в практике программирования, и мы не будем приводить примеры его использования.

Итак, из рассмотренного выше примера видно, что упакованные битовые поля, как и другие упакованные структуры данных, которые мы рассматривали ранее, позволяют экономить место в памяти для размещения данных, но требуют значительно больше команд для манипулирования такими упакованными данными. На этом закончим наше краткое знакомство с упакованными битовыми полями в языке Ассемблера.

9.5. Структуры на Ассемблере

Другим примером описания *типов* в языке Ассемблера являются структуры. В языке Паскаль аналогом структур Ассемблера являются записи, которые позволяют описать совокупность именованных переменных (вообще говоря, разных типов) как новый самостоятельный тип данных. Описания типа структуры задаются в языке Ассемблера в виде

```
<имя типа структуры> struc
    Предложения резервирования
    памяти под поля структуры
<имя типа структуры> ends
```

[1] Как исключение метки, указанные как операнды в командах *относительного* перехода, как мы уже знаем, имеют значения констант, задающих знаковое расстояние до точки перехода в байтах.

В качестве примера рассмотрим описание структуры, предназначенной для хранения информации о студенте**:** текстовой строки, в которой будет храниться фамилия студента (не более 20 символов), а также трёх полей для хранения даты рождения (дня, месяца и года).

```
Stud  struc
Fio   db    "********************"; 20 '*'
      db    '$'
Day   db    ?
Month db    ?
Year  dw    1986
Stud  ends
```

В нашей структуре описано 5 полей. Первое поле с именем `Fio` длиной 20 байт предназначено для хранения фамилии студента, у этого поля есть значение по умолчанию – это строка из 20 символов `'*'`. Обратите внимание на следующую тонкость**:** это предложение нельзя записать в виде

```
Fio   db    20 dup ('*'); 20 '*'
```

так как в этом случае будет определено 20 полей длиной по одному байту каждое. Соответственно, если в первом случае имя `Fio` является именем всей *строки символов* (массива коротких целых чисел), то во втором – будет только именем первого символа в массиве из 20 символов.

Второе поле нашей структуры длиной в один байт не имеет имени, но имеет значение по умолчанию `'$'`. Как видим, мы подготавливаем удобный способ вывода фамилии студента с использованием уже знакомой нам макрокоманды **`outstr`**. Три последних поля нашей структуры определяют переменные для хранения числовых значений дня, месяца и года рождения студента, причём у года рождения мы предусмотрели значение по умолчанию `1986`.

Каждое имя поля имеет целочисленное значение, которое равно смещению в байтах начала этого поля от начала структуры. Так, например, имя поля `Month` в описанной выше структуре `Stud` имеет значение `22`.

После описания структуры можно резервировать в некотором сегменте переменные описанного типа с помощью предложений резервирования памяти, например:

```
St1   Stud  <'Иванов И.И.',,21,4>
St2   Stud  <'Петров П.П.',,31,5,1985>
Grup  Stud  30 dup (<>); Студенческая группа
```

Строки, задающие начальные значения фамилий, дополняются пробелами *справа*, если их длина менее длины поля (в нашем примере менее 20 символов). Обратите внимание, что мы выделили двумя запятыми подряд второе *безымянное* поля, которому при описании переменной, таким образом, не присваивается *нового* значения, и это поле сохраняет начальное значение по умолчанию `'$'`, определённое при описании структуры `Stud`. Для студента с фамилией `Иванов` мы не задали начального значения поля, в котором хранится год рождения, таким образом, это поле будет иметь начальное значение `1986`, заданное в описании типа `Stud`.

К сожалению, поля структуры могут принадлежать только к стандартным типам Ассемблера (**`db`**, **`dw`**, **`dd`** и т.д.). Нельзя, например, задавать в виде полей структуры определённые пользователем упакованные битовые поля и другие структуры.

В качестве небольшого примера рассмотрим фрагмент программы на Ассемблере, в котором выводится информация о студенте, хранящаяся в переменной с именем `Z` типа `Stud`.

```
mov   dx,offset Z.Fio
outstr
newline
xor   ax,ax
mov   al,Z.Day
outword ax
outch '/'
mov   al,Z.Month
outword ax
outch '/'
outword Z.Year
```

```
        newline
```

В этом примере мы встретились с новым *двуместным* оператором языка Ассемблер, который обозначается символом `'.'`. Этот оператор выполняется Ассемблером так же, как и двуместный оператор `'+'`, однако тип (размер) этого операнда задаётся тем именем, которое располагается после точки. Таким образом, команда

```
        outword Z.Year
```

эквивалента команде

```
        outword word ptr Z+Year
```

Это показывает, как различаются правила вычисления типа выражения, в которое входит поле структуры. Так, хорошо изучив рекомендованный Вам учебник по языку Ассемблера, постарайтесь понять, что `type Z.Year=2`, `type (Z+Year)=35`, а `type Z+Year=68`. Другим отличием (для нашего примера, впрочем, несущественным), является различный *приоритет* операций Ассемблере `'.'` и `'+'`. Операция `'.'` имеет более высокий приоритет, чем операция `'+'`, что может оказаться полезным для записи некоторых выражений без использования круглых скобок. Приоритеты всех операций языка Ассемблер и правила вычисления типов Вам необходимо обязательно выучить по учебнику [5].

В качестве ещё одного примера напишем на Ассемблере функцию со стандартным соглашением о связях, которая получает в качестве параметров массив KURS структур с данными о студентах, длину этого массива N и беззнаковое целочисленное значение Y. Функция будет вырабатывать в качестве своего значения количество студентов, родившихся в году Y. Соответствующие данные в Ассемблере можно, например, описать так (предполагаем, что структура Stud уже описана):

```
N     equ   500
KURS  Stud  N dup (<>)
Y     dw    ?
```

Тогда вызов нашей процедуры (дадим ей, не долго думая, имя P) будет производиться, например, командами:

```
        mov   ax,offset KURS
        push  ax
        mov   ax,N
        push  ax
        push  Y
        call  P
```

Ниже приведён возможный текст этой процедуры

```
P       proc  near
        push  bp
        mov   bp,sp
        push  bx
        push  dx
        push  cx
        mov   dx,[bp+4]; Год Y
        mov   cx,[bp+6]; Длина массива N
        mov   bx,[bp+8]; Начало массива
        xor   ax,ax; Число студентов с годом рождения Y
        jcxz  Vozv; Не всех ли уже отчислили? ☺
L:      cmp   [bx].Year,dx
        jne   L1
        inc   ax
L1:     add   bx,type Stud; На след. студента в массиве X
        loop  L
Vozv:   pop   cx
        pop   dx
        pop   bx
        pop   bp
```

```
        ret   6
P       endp
```

Из этого примера видно, что значением одноместного оператора **type**, применённого к имени типа структуры (как, впрочем, и к имени самой переменной этого типа) является длина структуры в байтах. Для нашего примера **type** Stud = **type** KURS = **type** St1 = 25. Здесь следует подчеркнуть, что наша функция, конечно, не знает, что переданный ей массив программист назвал именем KURS. В то же время для правильной работы функция обязана знать имя типа Stud, который имеют элементы массива, а также должна знать имена всех полей структуры, с которыми она работает. Стоит отметить, что это же будет справедливо и для аналогичной функции, написанной на Паскале.

На этом мы завершим знакомство с дополнительными возможностями языка Ассемблер, более подробно эту тему необходимо изучить по учебнику [5].

Глава 10. Модульное программирование

Архитектура ЭВМ тесно связана со способами выполнения программ, поэтому сейчас мы переходим к изучению большого раздела нашего курса под названием "Введение в системы программирования". Мы рассмотрим весь путь, который проходит новая программа – от написания текста на некотором языке программирования, до этапа счёта этой программы. Первая тема в этом разделе и называется "Модульное программирование".

Модульное программирование предполагает особый способ разработки программы, которая при этом строится из нескольких относительно независимых друг от друга частей – *модулей*. Понятие модуля является одним из центральных при разработке программного обеспечения, и сейчас мы приступим к изучению этого понятия.

Известно, что при разработке программ в основном используется метод, который называется "программирование сверху-вниз" или "пошаговая детализация". Суть этого метода совсем проста: исходная задача сначала разбивается на относительно независимые друг от друга подзадачи. В том случае, когда полученные подзадачи относительно просты, то для каждой из них разрабатываетсясоответствующий алгоритм, иначе каждая такая всё ещё сложная подзадача снова разбивается на более простые и т.д. Далее мы приступаем к реализации каждой из полученных таким образом относительно простых подзадач на некотором языке программирования, и каждая такая реализация подзадачи и называется чаще всего (программным) модулем. Таким образом, использование модулей является естественным способом разработки и построения сложных программ.

Модули задачи 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 более строго определим несколько позже, а пока изучим общее понятие модульного программирования и *программного модуля*.

Мы уже знаем одно из полезных свойств программы, отдельные части (модули) которой написаны на разных языках программирования – это позволяет нам из программ на языках высокого уровня вызывать процедуры на Ассемблере. Далее мы будем знакомиться со свойствами модульной программы, написанной целиком 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. А вот в разных модулях Ассемблера имена могут совпадать, так как имена, как и в блоке программы на языке Паскаль, *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.

Следует отметить и такое хорошее свойство модульного программирования, как возможность повторного использования (reuse) разработанных модулей в других программах. Очевидно, что для повторного использования программ их следует оформлять в виде процедур и функций со стандартными соглашениями о связях.

---

[1] Из этого правила совсем немного исключений, это, например, повторение имён сегментов и процедур в начале и в конце их описания. С другими исключениями уникальности имён в ассемблерном моделе мы познакомимся при изучении *макросредств* языка Ассемблера.

[2] Точнее на *объектный* язык, о чём мы будем говорить далее.

Разумеется, за всё надо платить, у модульного программирования есть и свои слабые стороны, перечислим основные из них.

- Во-первых, модули не являются совсем уж 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, получая от других модулей входные данные и передавая им результаты своей работы. Заметим, что все такие связи между модулями на Ассемблере должны быть явно заданы при описании этих модулей.[1]
- Во-вторых, теперь перед счётом программы необходим особый этап *сборки* программы из составляющих её модулей. Этот процесс достаточно сложен, так как кроме собственно объединения всех модулей в одну программу, необходимо проконтролировать и установить все связи между этими модулями.[2] Сборка программы из модулей производится специальной системной программой, которая называется *редактором внешних связей* между модулями (по-английски связь называется *link*, поэтому жаргонное название этой программы – линкер или линковщик).
- 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.

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 Вашем учебнике 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 (главным) *кодовым* 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 (т.е. адрес первой 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 управления можно использовать одну из команд перехода.

---

[1] Точнее, явно задаются только так называемые статические связи между модулями, о чём мы будем подробно говорить далее.

[2] Возможно выполнение программы без её сборки из модулей, при этом установление связей между модулями будет отложено на этап счёта программы, о чём вы будем говорить позже при изучении схемы работы так называемого динамического загрузчика.

Кроме связей по управлению, между модулями могут существовать и связи по *данным*. Связи по данным предполагают, что один модуль может иметь доступ по чтению и/или записи к областям памяти (переменным) в другом модуле. Частным случаем связи по данным является и использование одним модулем именованной 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, во-первых, компилятор в "видит" только один этот модуль, и, во-вторых, будущее расположение модулей (их сегментов) в памяти во время счёта на этом этапе, как правило, неизвестно.

Связи между модулями будем называть *статическими*, если численные значения этих связей (т.е. адреса) известны сразу после размещения программы в памяти компьютера, но *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 статические связи между модулями задаются с помощью специальных директив. Директива

public <список имён модуля>

объявляет перечисленные в этой директиве имена *общедоступными* (**public**), т.е. разрешает использование этих имён в других модулях. Общедоступными можно сделать только имена переменных, меток и целочисленных констант. 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 тип равен `-1` для близкой метки и `-2` для дальней. Остальные имена (имена сегментов, имена констант в директивах числовой 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).[3]

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 каких-то 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 на Ассемблере первый модуль должен *разрешить* использовать некоторые из своих имён в других модулях, а второй модуль – яв-

---

[1] Это верно и для случая, когда один модуль использует константу, определённую в другом модуле, так как константа – это тоже целое число (непосредственное значение формата `i8` или `i16`). Кроме того, мы не будем рассматривать связи между модулями с помощью внешних файлов, в этом случае связи задаются не целыми числами, а текстовыми строками – именами таких файлов.

[2] В некоторых языках имена одного модуля по умолчанию считаются доступными из других модулей, если они описаны или объявлены в определённой части первого модуля. Например, в языке С, который Вы будете изучать в следующем семестре, в модуле общедоступны все имена, описанные *вне* функций, если про них явно не сказано, что это локальные имена модуля.

[3] Как мы узнаем позже, имена сегментов можно сделать видимыми из других модулей, но не с помощью директивы **public**, а другим, более сложным, способом.

но *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 целочисленные 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.asm` 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, а третьим, как и рагьше, будет модуль с именем `ioproc.asm` (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. Сделаем следующее разбиение нашей задачи на подзадачи-модули. Ввод массива и вывод результатов, а также выдачу диагностики об ошибках 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 *внешние* имена для получения своих параметров, выдачи результата работы и диагностики об ошибке.

Текстовый файл, содержащий первый (головной) модуль нашей программы на Ассемблере, мы, не долго думая, назовём `p1.asm`, а файл с текстом второго модуля, содержащим процедуру суммирования массива, назовём `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 заведомой ошибкой
      mov   A,7FFFh; Maxint
      mov   A+2,1; Чтобы было  переполнение
      call  Sum
      outint Summa; Сюда возврата не будет!
      newline
      finish ; Вообще-то не нужен
      public Error; Входная точка
```

```
Error:lea   dx,T; Диагностика
      outstr
      finish
Code  ends
      end   Start; Это 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, директива
   include io.asm не нужна, т.к. 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; Внешняя переменная
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
      add   SP,4; удаление возврата
      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 конечной 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 по директиве **include**. Оба наших модуля используют общий стек объёмом 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=seg:off на месте операнда Sum команды **call** Sum = **call** seg:off

Между основной программой и процедурой установлены следующие (нестандартные) соглашения о связях. Суммируемый массив *знаковых* чисел расположен в сегменте данных головного модуля и имеет общедоступное имя A. Длина массива является общедоступной константой с именем N, также описанной в головном модуле. Вычисленная сумма массива помещается в общедоступную переменную с именем Summa, описанную во втором модуле. Всё это примеры *статических* связей между модулями 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 переданной по ссылке переменной становится известным вызванной процедуре только во время счёта программы, когда он передан ей основной 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 используются две метки с одинаковым 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; Внешняя переменная
```

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 здесь снова пригодилась.

Продолжим рассмотрение работы нашей модульной программы. Получив управление, процедура Sum сохраняет в стеке используемые регистры (эта часть соглашения о связях у нас выполняется), и накапливает сумму всех элементов массива A в регистре ax. При ошибке переполнения процедура восстанавливает запомненные значения регистров, удаляет из стека дальний адрес возврата (4 байта) и выполняет команду безусловного перехода 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 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

[1] Иногда в таких случаях говорят, что имя A *объявлено* в сегменте Data (правда термин "объявить имя" используется в основном в языках высокого уровня). Объявление имени переменной в некотором сегменте модуля, в отличие от *описания* этого имени, не требует выделения для переменной памяти в данном сегменте этого модуля. Для Ассемблера такое объявление переменной является также и указанием (директивой) о том, что во время счёта программы данная переменная будет находится в памяти именно этого сегмента (сегмента с этим именем), как это обеспечивается мы узнаем несколько позже.

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 переводится 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, поэтому 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 (области памяти) 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, заданных директивами **`public`**, 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 не адрес, а просто целое число – значение этой константы).
- Сведения об именах и типах всех внешних адресов модуля, заданных в директивах **`extrn`**.
- Сведения о местоположении всех остальных незаполненных полей в сегментах модуля, для каждого такого поля задано его месторасположение в сегменте и 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`, независимо полученных программой Ассемблера, для каждого модуля изображены его сегменты, входные точки и внешние адреса. Вся эта информация содержится в паспортах объектных модулей (напомним, что для простоты изложения мы не принимаем во внимание третий модуль нашей программы с именем `ioproc`).[1]

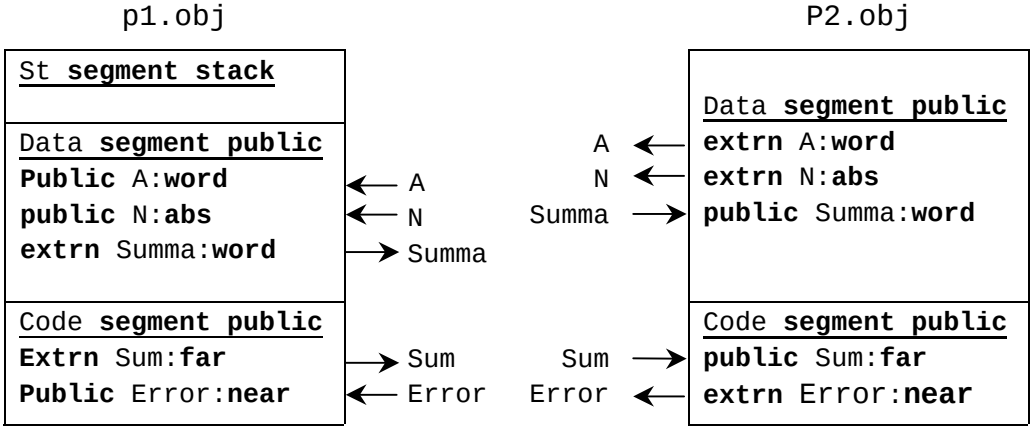


Рис. 10.1. Схематический вид двух объектных модулей с внешними адресами и входными точками.

Обратимся теперь к проблеме сборки полной программы из модулей. Как мы уже упоминали, эту работу выполняет специальная системная программа, которая называется *редактором внешних*

[1] Все имена исходного модуля, кроме внешних имён и имён входных точек, обычно заменяются в объектном модуле их численными значениями. Например, уже на важно, какое имя имела та или иная (локальная) переменная программы, эта информация больше никому не понадобится (исключением являются так называемые "отладчики в терминах входного языка", но для Ассемблеров они практически не используются и мы не будем касаться этой темы).

связей. Из этого названия хорошо видно одно из назначений этой программы – редактировать (в смысле устанавливать, настраивать) связи между внешними адресами и входными точками модулей. Рассмотрим схему работы редактора внешних связей на нашем предыдущем примере.

10.2. Схема работы редактора внешних связей

Целью работы редактора внешних связей является построение из объектных модулей почти готового к счёту программы, которая 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 первая команда в головном модуле

```
mov  ax,Data
```

всё ещё будет иметь незаполненное поле `Data` формата `i16` на месте второго операнда, так как конкретное значение этого поля (адрес сегмента данных) 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 (в нашей системе программирования 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* (или *блоки*) данных).

Как именно будут обрабатываться сегменты при сборке загрузочного модуля из объектных модулей, определяет сам программист, задавая определённые *параметры* в директивах **`segment`**. Существуют следующие параметры, управляющие обработкой сегментов редактором внешних связей.

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 одноимённые сегменты из других объектных модулей. В том случае, если одноимённые сегменты с параметром **`public`** встре-

---

[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, в нашем языке Ассемблера склеиваемые сегменты должны ещё принадлежать к одному и тому же *классу* сегментов. В рассмотренных нами примерах это требование выполняется, а для более полного изучения понятия класса сегмента необходимо обратиться, например, к учебнику [5].

чаются не в головном модуле, то их порядок при склейке определяется конкретным редактором внешних связей (надо читать документацию к нему).[1]

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 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 (информация о необходимости какой коррекции начального значения, естественно, тоже содержится в паспорте второго объектного 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 и 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 суммирования массива, при этом сегменты данных двух наших модулей будут *накладываться* друг на друга. В то же время *кодовые* сегменты будут по-прежнему склеиваться. 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 common
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
```

[1] Заметим, что одноимённые сегменты (как с параметром **`public`**, так и без него) могут встречаться и внутри *одного* модуля. В этом случае такие сегменты тоже склеиваются, но эту работу проводит не редактор внешних связей, а сама программа Ассемблера.

```
      extrn Sum:far; Внешнее имя
      call  Sum; Процедура суммирования
      outint S; синоним имени Summa в p2.asm
      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
      add   SP,4; удаление возврата
      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 с именем Data 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 – начала нашего суммируемого 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 сегменты команд почти всегда бессмысленно). Вследствие этого можно (как в нашем примере) резко сократить число *явных* связей по данным (то есть имён 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 местами массив и переменную для хранения суммы этого массива (при компиляции никакой диагностики об этой *семантической* ошибке при этом, естественно, не будет).

Итак, явные статические связи по данным требуют согласования имён и типов входных точек и внешних адресов в разных модулях, а неявные статические связи (при наложении сегментов данных) требуют согласования взаимного расположения и типов переменных в сегменте (имена теперь являются локальными и о них не надо договариваться). Оба способа связи модулей по данным имеют свои достоинства и недостатки, однако, современные способы программирования рекомендуют как можно большее число связей между модулями по данным вообще делать не статическиеми, а динамическими. Другими словами, модули (кроме, естественно, головного) следует оформлять в виде наборов процедур и функций со стандартными соглашениями о связях. В этом случае, как мы знаем, связь между такими модулями по данным реализуется посредством механизма фактических и формальных параметров, при этом отпадает необходимость "договариваться" между модулями об именах или взаимном расположении параметров в сегментах.

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 во время счёта 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 значения переменных этого сегмента будут обозначать конкретные адреса процедур обработки прерываний. Так заданный сегмент данных может облегчить для программиста 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 внешнее имя" (термин "неразрешённое" имеет здесь тот же смысл, что и в выражении "неразрешённая, т.е. нерешённая проблема").

Для некоторого внешнего имени могут существовать и несколько входных точек в разных модулях. При этом многие редакторы внешних связей такую ситуацию не считают ошибкой, и берут при просмотре паспортов модулей первую встреченную входную точку с таким именем, так что про-

[1] Стиль программирования с общими (накладываемыми друг на друга) сегментами данных широко используется, например, в языке Фортран, там такие сегменты называются общими (**common**) блоками данных для нескольких модулей. При этом часто случаются ошибки, вызванные плохим семантическим согласованием переменных в таких общих блоках памяти.

граммисту надо быть осторожным и обеспечить уникальность входных имён у всех модулей. Можно сказать, что такие редакторы внешних связей предполагают, что многомодульная программа разрабатывается сравнительно небольшим коллективом программистов, которые всегда смогут договориться между собой об уникальности всех внешних имён модулей (это часть спецификации разрабатываемой программы).[1]

К большому сожалению, некоторые редакторы внешних связей (в том числе и в нашей системе программирования с Ассемблером 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 будет трудно найти при отладке программы. Заметим, что в тех системах, которые уделяют большое внимание вопросам надёжности программирования, такие проверки всегда делаются. Например, редактор внешних связей Турбо-Паскаля считает несоответствие типов внешнего имени и входной точки фатальной ошибкой. Скажем здесь, что в системе программирования с языком С, который Вы будете изучать в следующем семестре, также принимаются специальные меры по контролю соответствия типов внешних имён и входных точек модулей.

Вернёмся к нашему редактору внешних связей. 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, в поле с именем `N` второго операнда команды

```
mov   cx,N
```

запишется значение `1000`:

```
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,[2] а в паспорте собраны необходимые для дальнейшей работы данные:

- 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). Заметьте, что параметры сегментов **`public`** и **`common`** уже не нужны. Внутри сегмента кода показаны незаполненные поля (они подчёркнуты). Метку `Start` можно рассматривать как единственную *входную точку* загрузочного модуля.

---

[1] В некоторых технологиях программирования программные комплексы разрабатываются большими коллективами программистов, которые, вообще говоря, могут быть разбросаны по всему миру, и общаться между собой только по сети. В таких системах предпринимаются специальные меры, обеспечивающие уникальность всех внешних имён (в частности, и имён самих модулей). См. например [19].

[2] 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 "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).

p.exe

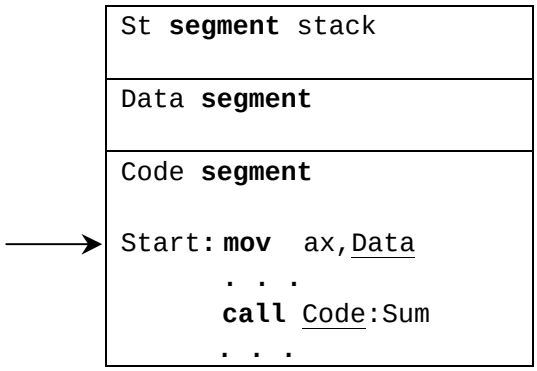


Рис. 10.2. Схематический вид загрузочного модуля, незаполненные поля в сегментах подчёркнуты.

Вот теперь всё готово для запуска программы на счёт. Осталось только поместить нашу программу в оперативную память, задать значения незаполненным полям и передать управление на начало программы (в нашем примере – на метку с именем `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, в котором хранится загрузочный модуль. 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 и 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. Иногда 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 о нехватке памяти (unsufficient memory) 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.

Затем загрузчик просматривает паспорт загрузочного модуля и заполняет в его сегментах все поля, которые ещё не имели необходимых значений. В нашем предыдущем 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.[3] 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 теперь готов к работе.[4]

---

[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 не читаются из файла, а просто размещаются на свободных местах памяти и, естественно, области памяти в них не будут иметь конкретных начальных значений.

[3] Наш компилятор с Ассемблера те поля, которые заполняются редактором внешних связей, помечает в листинге программы как `0000 E`, а те поля, которые заполняет загрузчик – как `---- R`.

[4] 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*).

И, наконец, последним действием загрузчик производит дальний абсолютный переход на начало загруженной программы, например, по команде

```
jmp Code:Start
```

Здесь `Code` – адрес начала головного кодового сегмента, а `Start` – адрес (смещение) входной точки 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, как-то возвратить 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.[1] Очевидно, что это должна делать процедура-обработчик соответствующего прерывания, которое происходит при выполнении макрокоманды **`finish`**. 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* разработки и выполнения модульной программы, эта схема включает в себя следующие этапы:

- Разбиение задачи на подзадачи.
- Реализация каждой такой подзадачи в виде модуля на некотором языке программрования, такой модуль принято называть исходным модулем.
- Синтаксическую отладку каждого модуля путём его компиляции в объектный модуль.
- Сборка из объектных модулей загрузочного модуля с помощью с помощью редактора внешних связей.
- Запуск загрузочного модуля на счёт с помощью служебной программы-загрузчика.

Два последних этапа определяют схему счёта, которая носит название "статическая загрузка и статическое связывание модулей", здесь имеется в виду, что до начала счёта программы вся она располагается в оперативной памяти компьютера и все статические связи между модулями уже установлены. 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, так как все статические внешние связи уже установлены, а соответствующие внешние адреса известны и записаны в сегментах программы.

Поймём теперь, что эта схема выполнения модульной программы со статической загрузкой и статическим связыванием модулей имеет два очень серьёзных недостатка. Чтобы понять, в чём они заключаются, предположим сначала, что наша достаточно сложная программа состоит из 100 модулей (для простоты будем считать, что в каждом модуле содержится только *одна* процедура или функция нашей 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 запуске программы на счёт, 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 данны-

[1] Для младших моделей нашего семейства компьютеров загрузчик входил в состав так называемой управляющей программы, которая выполняла команды (директивы) пользователя. Таким образом, загруженная на счёт программа пользователя после своего успешного или аварийного завершения должна была возвратиться в эту управляющую программу, которая обычно имела имя `COMMAND.COM`.

ми) могут понадобиться другие 10 процедур из 100, но в целом эта безрадостная картина не меняется: каждый раз во время счёта программы около 90% оперативной 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. 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 (с этой схемой мы будем знакомиться далее). 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, как более простая.

Далее отметим, что в настоящее время счёт модульных программ (а подавляющее большинство "больших" программ только такие теперь и есть), 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 решаемых задач и число реализующих их модулей растёт *быстрее*, что объём памяти.

Второй серьёзный недостаток схемы счёта модульной программы со статическоим связыванием и статической загрузкой проявляется при так называемом мультипрограммном режиме работы ЭВМ (отметим, что в настоящее время большинство ЭВМ работают именно в этом реэиме). С мультипрограммным режимом работы мы будем детально знакомиться далее, пока лишь отметим, что в этом режиме в памяти ЭВМ могут одновременно находиться несколько независимых друг от друга и готовых к счёту программ разных пользователей. И вот оказывается, что что во многих программах часто приходится использовать *одинаковые* программные модули. Эти модули выполняют функции, без которых не может обойтись большинство программ: это организация диалогового интерфейта с пользователями (всемозможные окна, меню, формы для ввода данных и т.д.), работа с файлами, работа с динамическими переменными и многое другое.[2] При схеме счёта со статическим связыванием и статической загрузкой приходится все эти общие для многих программ модули включать на этапе редактирования связей в состав *каждой* такой программы и хранить в *каждом* загрузочном модуле. Схема счёта модульной программы с динамической загрузкой и динамическим связыванием модулей призвана решить эту проблему.

Перейдём теперь непосредственно к изучению схемы счёта модульной программы с использованием динамического связывания и динамической загрузки модулей. Как уже говорилось, обычно та системная программа, которая занимается динамическим связыванием и динамической загрузкой модулей, называется *динамическим загрузчиком*.

10.4. Схема работы динамического загрузчика

Итак, пусть наша программа состоит из головного модуля (основной программы) и какого-то числа процедур и функций, располагающихся в остальных модулях.

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, содержащие процедуры и функции, загружаются в оперативную память по мере необходимости, когда к этим процедурам и функциям 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 могут вызываться в процессе работы программы, *на самом деле существуют* (на это остаётся

---

[1] Виртуальная память позволяет, в частности, использовать в программе объём памяти, превышающий физический объём памяти компьютера. Например, при физической памяти 2^{20} байт (как в нашей младшей модели) можно использовать под (одновременное!) размешение сегментов программы, наприер, 2^{24} байт. Виртуальную память студенты факультета вычислительной математики и кибернетики МГУ изучают в курсе третьего семестра "Системное программное обеспечение".

[2] Это так называемая проблема повторного использования (reuse) программного обеспечения: разработанные для одной задачи программные модули часто можно использовать и в других задачах.

только надеяться изо всех сил ☺).[1] Естественно, что, если какой-нибудь модуль не будет найден, когда понадобится его вызвать, то будет зафиксирована ошибка времени выполнения программы. Отметим, что это один из *недостатков* такой схемы счёта модульной программы. Впрочем, если посмотреть с другой стороны, то возможность выполнять программы, не все модули которых ещё на самом деле написаны, позволяет начать отладку программной системы на самых ранних стадиях её реализации, что, конечно, является одним из *достоинств* этой схемы выполнения.

Надо отметить, что динамические загрузчики в современных ЭВМ достаточно сложны и, к тому же, сильно различаются в разных системах программирования, поэтому мы рассмотрим только упрощённую схему работы такого загрузчика. Эта схема будет похожа на схему работы динамических загрузчиков в ЭВМ второго-третьего поколений 70-х годов 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:

```
; Фрагмент головного модуля с точкой входа
Code  segment
      assume cs:Code,ds:Data,ss:Stack
Start:mov   ax,Data
      mov   ds,ax
      . . .
      extrn A:far
      call  A
      . . .
      extrn Beta:far
      call  Beta
      . . .
      extrn C12:far
      call  C12
      . . .
      finish
Code  ends
      end   Start; головной модуль
```

Пусть для простоты внешние процедуры с именами `A`, `Beta` и `C12` расположены каждая в своём отдельном модуле, где эти имена, естественно, объявлены общедоступными (**`public`**) и имеют тип дальних меток (**`far`**). 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), пусть каждая из этих таблиц располагается в своём отдельном сегменте памяти.

В таблицу внешних имён заносятся все внешние имена выполняемой программы (в начале работы у нас 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 С). 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 приве-

[1] 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, находится где угодно, например, в сетевой (удалённой) библиотеке объектных модулей на другом конце Земли. В качестве примера сошлёмся на так называемую COM технологию программирования [19].

дён вид ТВИ после обработки головного модуля, сейчас в этой таблице три имени: A, Beta и C12 (в каждой строке таблицы, кроме заголовка, мы для удобства чтения разместили по четыре символа).

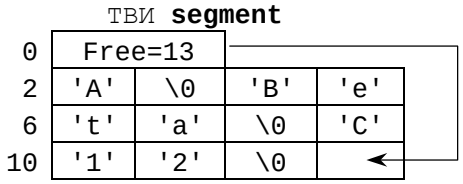


Рис. 10.3. Вид таблицы внешних имён после загрузки головного модуля.

Другая таблица динамического загрузчика, таблица внешних адресов, состоит из строк, каждая строка содержит пять полей. Первое поле имеет длину четыре байта, в нём динамический загрузчик размещает команду *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 в оперативную память и перейти на выполнение этой процедуры. Отметим, что сама процедура LoadGo загружается в память вместе с головным модулем до начала счёта и *статически* связана с этим модулем.

Во втором поле (назовём его именем Offset) длиной 2 байта находится адрес (смещение) загруженной 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=-1, что является признаком *отсутствия* данной процедуры на рабочем поле. В третьем поле тоже длиной в два байта расположена ссылка на имя этой внешней процедуры в таблице внешних имён. В четвёртом поле длиной в два байта с именем Length будет храниться длина внешней процедуры (пока там значение ноль). И, наконец, пятое 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 12 байт. В заголовке (первых двух байтах) ТВА содержится ссылку на начало свободного места в этой таблице (эта таблица будет только расти сверзувниз, строки из неё никогда не будут удаляться). Таким образом, перед началом счёта программы ТВА будет иметь вид, показанный на рис. 10.4.

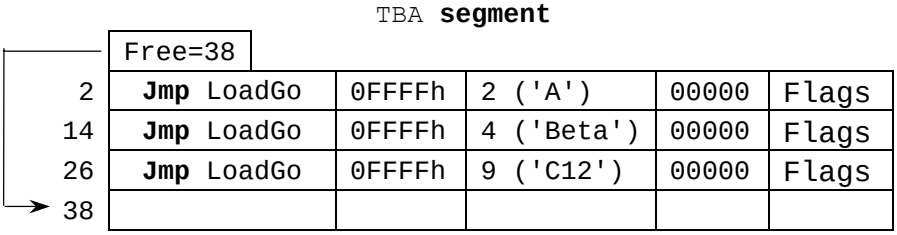


Рис. 10.4. Вид таблицы внешних адресов после загрузки головного модуля.

Каждая команда *дальнего* вызова внешней процедуры в головном модуле заменяется динамическим загрузчиком на команду дальнего перехода с возвратом на соответствующую строку ТВА. Например, команда `call Beta` заменяется на команду `call TBA:14`, а команда в головном модуле `call C12` заменяется на команду `call TBA:26`.

Проследим работу нашей программы. Пусть головная программа в начале своего выполнения вызывает, например, внешнюю процедуру с именем Beta, которая имеет следующий вид:

```
Beta  proc  far
      . . .
      extrn Delta:far
      call  Delta
      . . .
      ret
Beta  endp
```

Как видим, в процедуре Beta возможен вызов другой внешней процедуры с именем Delta. Естественно, что при первой попытке основной программы вызвать процедуру Beta, управление пере-

даётся на 14-ую строку TBA и вызывается служебная процедура LoadGo динамического загрузчика. Получив управление, процедура LoadGo последовательно выполняет следующие действия.

- Сначала вычисляется величина TBA_proc, равная адресу строки вызываемой процедуры Beta в таблице TBA. Например, для случая, как у нас, вызова процедуры Beta величина TBA_proc=14.
- Затем анализируется поле Offset в строке таблицы с адресом TBA_proc. Значение Offset=-1 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 должна быть описана входная точка с именем Beta и типом дальней метки **far**). Если такой объектный модуль не найден, то фиксируется фатальная ошибка времени выполнения и наша программа завершается, иначе объектный модуль с требуемой процедурой Beta загружается в оперативную память и динамически связывается с основной программой. Для этого корректируются таблица внешних адресов TBA и, возможно, таблица внешних имён ТВИ (если, как в нашем примере, в загружаемой процедуре есть и свои внешние имена). Для загрузки процедур в оперативной памяти компьютера выделяется специальная область, она часто называется *рабочим полем процедур*. Мы отведём под рабочее поле отдельный сегмент с именем Work, занимающий, например, 50000 байт (заметим, что рабочее поле в нашем простом примере должно помещаться в один сегмент):

```
Work    segment
        db    50000 dup (?)
Work    ends
```

  Рабочее поле размещается в оперативной памяти одновременно с сегментами головного модуля, ТВИ и TBA. После загрузки процедуры Beta поле Offset в строке TBA_proc принимает значение адреса начала этой процедуры на рабочем поле, а поле Length будет равно длине загруженной процедуры. Пусть длина процедуры Beta будет, например, 30000 байт. 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 00000.
- 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 поля сегмента в адресе возврата (обязательно поймите, как это сделать).
- Если, как в нашем примере, вызов внешней процедуры Beta 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 по команде **ret** будет производиться в головной модуль нашей программы (дальний адрес возврата, как обычно, находится на вершине стека). В нашем примере динамический загрузчик перейдёт на выполнение процедуры Beta с помощью команды дальнего абсолютного перехода `jmp Work:0`.

На рис. 10.5 показан вид ТВИ, TBA и рабочего поля после загрузки процедуры Beta.

Упражнение. Объясните, каким образом служебная процедура LoadGo, получив управление по команде `jmp LoadGo`, вычислит величину TBA_proc, то есть определит, что надо загружать, например, именно процедуру с именем Beta, а не какую-нибудь другую внешнюю процедуру из TBA.

Теперь, после динамической загрузки процедуры Beta на рабочее поле и связывания 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.

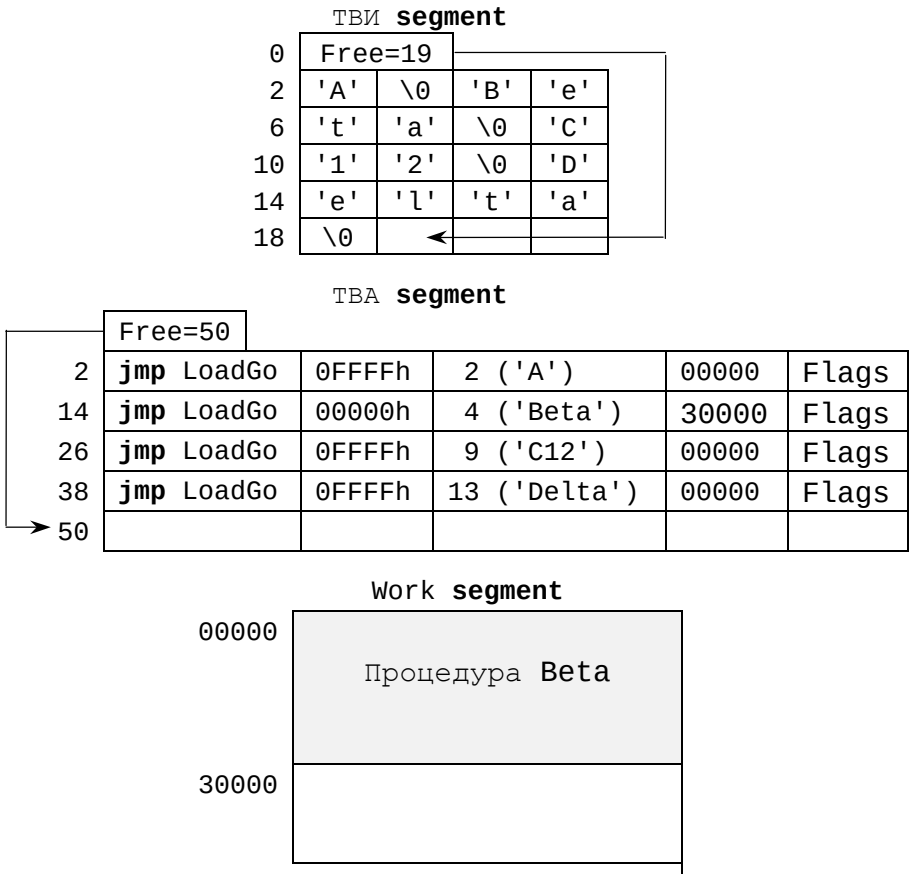


Рис. 10.5. Вид ТВИ, TBA и рабочего поля после загрузки процедуры Beta.

Продолжим анализ работы динамического загрузчика на нашем примере. Пусть загруженная на рабочее поле процедура Beta, проработав некоторое время, в свою очередь вызывает свою внешнюю процедуру с именем Delta, которая имеет, например, 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:38`, то управление опять получает служебная процедура LoadGo. Она находит процедуру Delta [1] 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.

Далее процедура LoadGo определила, что вызов процедуры TBA_proc(Delta)=38 производится не из основной программы, а из процедуры TBA_proc(Beta)=14, расположенной на рабочем поле. В этом случае LoadGo производит следующие действия, при выполнении которых используется ещё один служебный сегмент динамического загрузчика, описанный, например, так:

```
My_Stack    segment
Free        dw    0
            dw 4000 dup (?)
My_Stack    ends
```

Этот сегмент используется как вспомогательный программный (не аппаратный) стек динамического загрузчика. Наш стек с именем My_Stack, в отличие от машинного стека, будет "расти" свер-

[1] Точнее, как мы уже говорили, ищется объектный модуль, в котором расположена эта общедоступная (**public**) процедура.

ху-вниз, при этом переменная Free играет роль указателя вершины программного стека, то есть регистра SP.

TBA **segment**

	Free=50				
2	**jmp** LoadGo	0FFFFh	2 ('A')	00000	Flags
14	**jmp** LoadGo	00000	4 ('Beta')	30000	Flags
26	**jmp** LoadGo	0FFFFh	9 ('C12')	00000	Flags
38	**jmp** LoadGo	30000	13 ('Delta')	15000	Flags
50					

Work **segment**

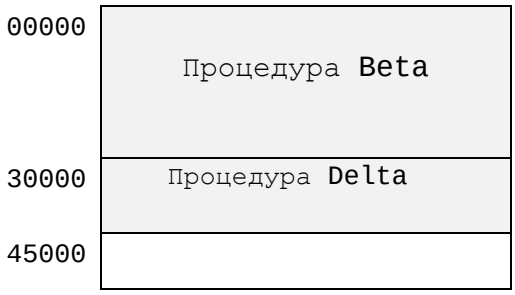


Рис. 10.6. Вид TBA и рабочего поля после загрузки процедуры Delta.

Сначала LoadGo извлекает из аппаратного стека (на него, как обычно, указывает регистровая пара <SS,SP>) адрес дальнего возврата (два слова). Затем в программный стек My_Stack сначала записывается одно слово (назовём его именем RETURN), которое имеет значение разности IP-TBA_proc(Beta).offset, где TBA_proc(Beta).offset – адрес начала процедуры Beta на рабочем поле, т.е. значение поля offset строки TBA для процедуры Beta. Как легко понять, величина RETURN равна смещению точки возврата относительно начала процедуры, из которой производится вызов.

Затем в программный стек записывается значение TBA_proc(Beta) (одно машинное слово), и таким способом запоминается, из какой процедуры с рабочего поля произошёл вызов. Таким образом, во вспомогательном стеке запоминается, из какой процедуры и из какого места этой процедуоы производится вызов. И, наконец, LoadGo производит вызов необходимой внешней процедуры Delta, расположенной на рабочем поле со смещением Offset=TBA_proc(Delta).offset от его начала, командой дальнего вызова процедуры **call** Work:Offset. Очевидно, что в этом случае и возврат из процедуры Delta по команде **ret** будет производиться не в вызвавшую её процедуру Beta, а в нашу служебную процедуру LoadGo.

После возврата из внешней процедуры в LoadGo, она производит следующие действия (напомним, что в этот момент ей уже известно о том, что вызов внешней процедуры был осуществлён из некоторой процедуры, расположенной на рабочем поле, иначе возврат из процедуры производится сразу в основную программу, миную LoadGo).

- 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 этой строке TBA_proc. Если величина Offset<>-1 , то это означает, что наша процедура всё ещё присутствует на рабочем поле. В этом случае из вспомогательного стека My_Stack извлекается значение RETURN смещения точки возврата относительно начала процедуры, после чего динамический загрузчик производит возврат в процедуру на рабочем поле по команде дальнего безусловного перехода **jmp** Work:Offset+RETURN.
- Разберём теперь случай, когда после возврата величина Offset=-1, 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`. Затем, как и в случае с `Offset<>-1`, из вспомогательного стека `My_Stack` извлекается значение `RETURN` смещения точки возврата относительно начала процедуры, после чего динамический загрузчик производит возврат в процедуру, которая теперь уже снова находится на рабочем поле, по команде дальнего безусловного перехода `jmp Work:Offset+RETURN`.

На этом `LoadGo` завершает обработку вызова внешней процедуры. Как видим, эта процедура играет роль своеобразного буфера, располагаясь между вызывающей программой и вызываемой внешней процедурой. На рис. 10.7 показана схема, поясняющая роль процедуры `LoadGo` на примере, когда одна процедура с именем `X` (это может быть и основная программа) вызывает внешнюю процедуру с именем `Y`.

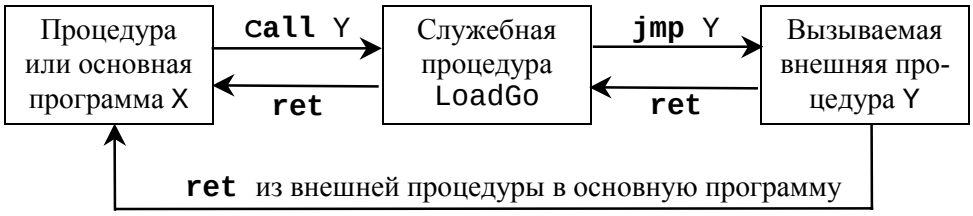


Рис. 10.7. Вызов внешней процедуры и возврат из неё при схеме счёта с динамической загрузкой и динамическим связыванием.

Как следует из описания алгоритма работы программы `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. Это позволяет не накладывать никаких ограничений на порядок вызова процедур, в частности, как мы и привыкли, рекурсивный вызов процедуры для динамического загрузчика ничем не отличается от не рекурсивного вызова.

Продолжим исследование работы динамического загрузчика на нашем примере. Предположим теперь, что произошёл возврат из процедур `Delta` и `Beta` в основную программы, которая после этого вызвала процедуру `A` длиной, скажем, в `25000` байт. Процедура `A` отсутствует на рабочем поле, поэтому её надо загрузить, однако процедура `LoadGo` определяет, что на рабочем поле нет достаточного 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`.[1] В нашем случае достаточно, например, удалить с рабочего поля процедуру `Beta`.

Итак, служебная процедура `LoadGo` считает память, занимаемую процедурой `Beta` на рабочем поле свободной, загружает на это место процедуру `A` и корректирует соответствующим образом строки `TBA.` Будем говорить, что процедура `Beta` *удаляется* с рабочего поля, а на её место загружается процедура `A`. На рис. 10.8 показан вид `TBA` 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, программист нуждается в некотором *управлении* динамической загрузкой модулей на рабочее поле. Например, пусть на рабочем поле уже находятся процедуры с именами `X` и `Z`, а в основной программ программист начинает в цикле попеременно вызывать процедуры с именами `X` и `Y`, например, так:

```
L:   Call  X
     Call  Y
     Loop  L
```

Тогда может случиться так, что при вызове и загрузке на рабочее поле процедуры `Y` она будет удалять с рабочего поля процедуру `X`, а процедура `X`, в свою очередь, при загрузки на рабочее поле

[1] Разумеется, мы предполагаем, что длина рабочего поля достаточна для загрузке на него любого объектного модуля, входящего в состав модульной программы.

будет удалять с него процедуру Y. Другими словами, на рабочем поле не хватает места, чтобы одновременно разместить на нём все три процедуры X,Y и Z. При возникновении такой ситуации выполнение программы резко замедлиться, так как динамический загрузчик большую часть времени будет занят чтением процедур из медленной внешней памяти на рабочее поле. Такого рода неприятные ситуации имеют в программистской литературе специальное название *трэшинг* (trashing – дрожание, мельтешение) памяти. Скорость выполнения всей программы при этом может упасть в несколько десятков и даже сотен раз.

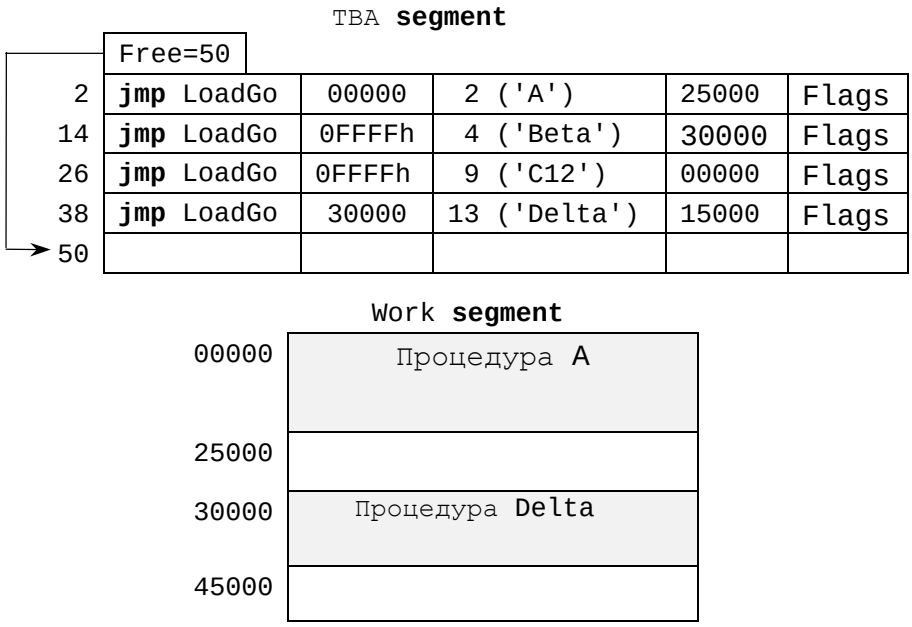


Рис. 10.8. Вид TBA и рабочего поля после загрузки процедуры A.

Для того чтобы избежать такой ситуации, программист может потребовать у динамического загрузчика, чтобы во время работы этого цикла 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 (в нашем примере процедуру Z), 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 специально определить способ передачи этого строкового параметра в служебную процедуру, а на языке Паскаль это можно было бы записать, например, так

```
Lock('X');
```

Процедура Lock находит в TBA строку, соответствующую указанной процедуре, и ставит в этой строке в поле флагов признак о том, что она зафиксирована на рабочем поле. Ясно, что для нашего примера с тремя процедурами с рабочего поля будет теперь удалена процедура Z, а процедуры X и Y будут всё время работы цикла находится на рабочем поле (мы, естественно, предполагаем, что процедуры X и Y вместе поместятся на рабочее поле, иначе трэшинга не миновать!).

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 было бы 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, если при размещении в оперативной памяти головного модуля программы он увидит вызов этих *внешних* процедур.

Теперь, когда мы познакомились со схемой выполнения программы в режиме динамической загрузки и связывания, обсудим те ограничения, которые необходимо наложить на процедуры и функ-

ции, выполняющиеся на рабочем поле. Во-первых, заметим, что во время счёта такие процедуры и функции могут быть *без их ведома* перезагружены с одного места рабочего поля на другое.[1] Это свойство схемы счёта программы с динамической загрузкой и связыванием налагает достаточно жёсткие ограничения на внешние процедуры и функции, которые будут выполняться на рабочем поле. Образно говоря, такие процедуры и функции не должны "помнить", на каком месте памяти они располагаются. Вам необходимо понять, что в архитектуре нашего компьютера такие процедуры и функции при этом должны обладать следующими свойствами.

- У них может быть только один кодовый сегмент, что налагает ограничение на максимальную длину процедур и функций (т.е. предполагает разбиение алгоритма на достаточно небольшие шаги). Кроме того, кодовый сегмент не должен изменяться (модифицироваться) в процессе счёта, так как при удалении модуля с рабочего поля он может быть вновь загружен из объектного модуля, где этих изменений уже не будет.[2]
- Они не имеют *собственных* сегментов данных, все обрабатываемые данные передаются им в качестве фактических параметров (можно сказать, что все сегменты данных принадлежат *основной* программе или являются *динамическими* переменными. Это ограничение следует из того, что при загрузке процедуры на новое место рабочего поля никто не меняет значения сегментных регистров, кроме кодового сегментного регистра.
- Все вызываемые из них процедуры и функции должны быть *дальними* и *внешними*, т.е. вызываться через служебную процедуру `LoadGo`. Действительно, возврат из близкой процедуры запоминается в стеке, а перед возвратом процедура может быть перемещена в другое место рабочего поля или просто удалена.[3]

Обязательно разберите и поймите, почему должны выполняться такие свойства внешних процедур. Заметим, что, хотя эти ограничения и могут показаться на первый взгляд обременительными, но они не создают принципиальных трудностей при программировании. Так, все приводимые нами в этой книге примеры процедур и функций (если сделать их *дальними*) удовлетворяют этим ограничениям.

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. Время, затраченное при счёте программы на выполнение служебных процедур, обычно принято называть, следуя бухгалтерской терминологии, *накладными расходами*. Обычно это расходы не превышают нескольких процентов от времени счёта программы.
- Во-вторых, может достаточно существенно замедлиться выполнения всей программы, так как теперь во время счёта понадобится периодически загружать внешние модули на рабочее поле, т.е. использовать обмен с относительно медленной внешней памятью. С этим недостатком, конечно, можно бороться, увеличивая размер рабочего поля и используя так называемую виртуальную память, что и делают динамические загрузчики на современных ЭВМ.

Итак, программист должен оценить эти дополнительные затраты, связанные со счётом по схеме с динамической загрузкой и связыванием, по сравнению со статической загрузкой и статическим связыванием. В том случае, если такие затраты допустимы, то схеме счёта с динамической загрузкой и связыванием следует отдать предпочтение.[4] В то же время следует иметь в виду, что, если общий объём всех модулей программы достаточно велик, то у схемы счёта с динамической загрузкой и связыванием вообще нет альтернативы, так как все модули просто не поместяться одновременно даже в большой памяти современных компьютеров.

---

[1] Забегая вперёд отметим, что далее, при классификации выполняемых модулей в следующей главе, мы скажем, что такие процедуры и функции относятся к классу *перемещаемых* программ.

[2] В большинстве случаев на современных ЭВМ во время выполнения программ кодовые сегменты закрыты на запись, т.е. попытка их изменения вызывает сигнал прерывания по защите от записи.

[3] Как следствие, запрещены *вложенные* (локальные) процедуры и функции, так как их наличие предполагает *близкий* вызов. Это свойство выполняется в нашем языке Ассемблера и в некоторых языках высокого уровня, где вложенные процедуры и функции тоже запрещены (например, в Фортране и С).

[4] 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.).

В заключение рассмотрения схемы работы динамического загрузчика отметим одно обстоятельство. Алгоритм работы рассмотренного нами динамического загрузчика можно значительно упростить, если запретить процедурам на рабочем поле вызывать друг друга, т.е. эти процедуры вызываются только из головной программы, и туда же производится возврат. Именно так и обстояло дело на ЭВМ первого и, частично, второго поколения, поэтому там динамический загрузчик был совсем простой программой, и его применение не слишком увеличивало время счёта. В качестве упражнения можете сами разработать схему такого упрощённого динамического загрузчика.

В современных ЭВМ наборы динамически загружаемых модулей одной тематики обычно объединяют в один файл – библиотеку динамически загружаемых и динамически связываемых модулей (в большинстве операционных систем они называются по-английски Dynamic Link Library – DLL). Таким образом, модульная программа состоит из головного модуля и одной или нескольких DLL библиотек. Заметим, что объединение близких по тематике процедур в один файл облегчает динамическому загрузчику поиск необходимых внешних процедур, так как оглавления (паспорта) таких библиотек можно всё время держать в памяти динамического загрузчика. Разумеется, при использовании таких библиотек необходима и специальная служебная программа *Библиотекарь*. С помощью Библиотекаря пользователь может включать в заданную библиотеку новые модули и исключать устаревшие, а также заменять модули, в которых найдены ошибки, на исправленные версии этих модулей.

Заметим далее, что в принципе один и тот же набор объектных модулей можно использовать как для статической схемы счёта (предварительно вызвав редактор внешних связей и статический загрузчик), так и для динамической схемы счёта (вызвав динамический загрузчик с параметром – головным объектным модулем). Таким образом, компиляторы могут получать объектные модули, пригодные для использования в любой их этих двух схем счёта.[1]

Сделаем теперь замечание к использованию одних и тех же программных модулей в *разных* программах при мультипрограммной работе ЭВМ. Из рассмотренной схемы работы динамического загрузчика ясно, что для решения этой проблемы достаточно сделать рабочее поле и таблицы динамического загрузчика *общими* для всех находящихся в памяти программ. При этом, правда, все такие модули, использующиеся в разных программах, должны обладать особым свойством: быть так называемыми реентарабельными (или параллельно используемыми). Что это такое, мы будем изучать в следующей главе.

На этом мы завершим наше по необходимости краткое знакомство со схемами выполнения модульных программ.

[1] Некоторые операционные системы (например, Windows) допускают и комбинированную схему счёта модульной программы, при этом часть модулей собирается редактором внешних связей в загрузочный модуль, а некоторые процедуры могут вызываться программистом при помощи динамического загрузчика (такой вызов делается не по обычной команде **`CALL`**, а обращением к специальной служебной процедуре операционной системы, которая является аналогом рассмотренной нами ранее служебной динамического загрузчика процедуры `LoadGo`).

Глава 11. Понятие о системе программирования

Как мы уже упоминали в начале нашего курса, все программы, которые выполняются на компьютере, можно разделить на две части – *прикладные* и *системные*. Вообще говоря, компьютеры существуют в основном для того, чтобы выполнять прикладные программы, однако понятно, что в данной книге нас то в первую очередь будут интересовать не прикладное, именно системное программирование.

Все системные программы можно, в свою очередь, тоже 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. Кроме того, обычно сюда же включают и служебные программы для управления обрабатываемыми данными (так называемую файловую систему). Программы этого класса входят в большой комплекс системных программ, который называется *операционной системой* ЭВМ.[1]

В другой класс входят системные программы, предназначенные для автоматизации процесса разработки и эксплуатации *новых* программ. Программы этого класса входят в состав *системы программирования*. Не надо, однако, думать, что система программирования состоит только из таких системных программ, которые помогают писать новые программы. Система программирования является *комплексом*, в состав которого входят языковые, программные и информационные компоненты. Ниже перечислены все компоненты, входящие в систему программирования.

11.1. Компоненты системы программирования

2. 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. В качестве такого командного файла можно привети файл с именем `ma.bat`, который Вы сейчас используете для разработки программ на Ассемблере (каждое предложение в этом файле является некоторой командой). В следующем семестре Вы познакомитесь езё с одним командным языком системы Unix – языком C-Shell. Другие входящие в систему программирования языки могут предназначаться для автоматизации разработки больших программ (например, так называемый язык спецификации программ). Вы не должны здесь путать три разных понятия: язык (например, Ассемблер), программу на этом языке и компилятор, который переводит Ассемблерные программы (на объектный язык).
3. 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 (обычно это исходные модули).
   - Трансляторы (компиляторы) для перевода с одного языка на другой (например, программа Ассемблера транслирует исходный модуль с языка Ассемблер на язык объектных модулей).[2]
   - Редакторы внешних связей, собирающие загрузочный модуль из объектных модулей в схеме счёта со статической загрузкой и статическим связыванием.
   - Статические и динамические загрузчики, запускающие задачи на счёт.

---

[1] Назначение и принципы работы операционных систем студенты факультета вычислительной математики и кибернетики МГУ подробно изучают в курсе третьего семестра "Системное программное обеспечение" на примере операционной системы Unix.

[2] Для продвитутых студентов заметим, что если исходные модули (жаргонное название – "исходники") программист может легко изменять с помощью текстового редактора, то объектные модули изменить практически нельзя (их можно только получить заново из изменённых исходных модулей). Более точно, попытка изменения объектного модуля выливается в задачу программирования на языке машины (не на Ассемблере!), и что бы там не говорили "крутые" программисты, никакие Дизассемблеры здесь существенно помочь не могут. Именно поэтому фирмы, продающие программное обеспечение, поставляют его пользователям чаще всего в виде загрузочных и объектных модулей, и пуще глаза берегут исходные тексты программ.

- Отладчики, помогающие пользователям в диалоговом режиме искать и исправлять ошибки в своих программах.[1]
- Оптимизаторы, позволяющие автоматически улучшать программу, написанную на определённом языке. Бывают оптимизаторы программ как на исходном языке программирования (например, на Фортране), так и на машинном языке (оптимизация загрузочных модулей).
- 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 эти фрагменты в виде процедур на языке Ассемблера).
- Библиотекари, которые позволяют создавать и изменять файлы-библиотеки процедур (например, библиотеки динамически загружаемых процедур DLL), файлы-библиотеки макроопределений, с которыми мы вскоре познакомимся, 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 программы).
- И другие служебные программы.

4. Информационное обеспечение системы программирования. Сюда относятся различные структурированные описания языков, служебных программ, библиотек модулей и т.п. Без хорошего информационного обеспечения современные системы программирования эффективно 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 (служебные) программы – в прямоугольники с закруглёнными углами. На этой схеме можно проследить весь путь, по которому проходит программа от написания её текста на некотором языке программирования, до этапа счёта.

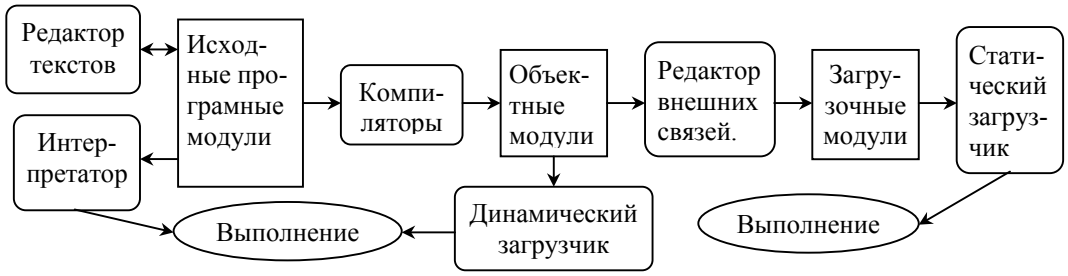


Рис. 11.1. Общая схема прохождения программы через систему программирования.

Заметим, что сейчас для многих языков программирования созданы так называемые *интегрированные среды*, включающие в себя работающие под общим управлением почти все компоненты системы программирования. Примером такой интегрированной среды разработки программного обеспечения является знакомая большинству программистов система Турбо-Паскаль.

На этом мы закончим описание состава системы программирования и перейдём к описанию характеристик исполняемых модулей.

[1] Для продвинутых студентов: обычно различают отладчики программ на языке машины (в кодах машины) и так называемые отладчики в терминах исходного языка программирования. Для отладчиков последнего вида при компиляции модулей с исходного языка в паспортах объектных модулей сохраняется достаточно полная информация об именах, использованных пользователем (это имена меток, процедур, переменных, констант и т.д.). В дальнейшем эта информация переносится редактором внешних связей в паспорт загрузочного модуля, и может быть использована таким отладчиком при выдаче соотбщений об ошибках (например, "При выполнении третьего оператора в процедуре `Summa` деление переменной `X` на ноль").

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 (без её ведома) перенесены в другое место оперативной памяти так, что это не повлияет на правильную работу этих модулей. Разбирая работу динамического загрузчика, мы выяснили, что во время счёта программы модуль может быть удалён с рабочего поля, а потом вновь загружен, вообще говоря, на другое место этого поля, т.е. все эти модули были перемещаемыми. Ясно, что динамический загрузчик может также и просто *переместить* модуль с одного места рабочего поля в другое, без его новой загрузки.

Рассмотрим пример, когда такое перемещение исполняемых модулей без их новой загрузки может оказаться полезным. На рис. 10.8 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 поместится процедура `C12`.

В том случае, если, как в нашем случае, модуль `Delta` является перемещаемым, его можно сдвинуть вверх так, чтобы объединить свободные участки рабочего поля и разместить на нём новую процедуру `C12`. Сдвиг модуля в оперативной памяти является по существу хорошо знакомой нам операцией пересылки массива с помощью цикла со строковой командой, и это заведомо более быстрая операция, чем удаление модуля с рабочего поля (ведь его потом, скорее всего, придётся вернуть обратно из медленной внешней памяти). Ранее, при изучении динамического загрузчика, мы уже рассмотрели ограничения, накладываемые на перемещаемые процедуры и функции.[1]

На первых ЭВМ перемещаемость была очень полезным свойством выполняемого модуля, так как позволяла уменьшить операции обмена с очень медленной внешней памятью. На современных ЭВМ, однако, появился механизм *виртуальной памяти*, который обеспечивает большое логическое адресное пространство (сейчас – порядка 2^{32} байт, а в дальнейшем будет ещё больше). Это позволило работать с очень большим (виртуальным) рабочим полем и назначать каждой процедуре свой диапазон адресов, несовпадающий с диапазонами адресов других процедур, таким образом, перемещаемость перестала быть важной характеристикой исполняемых модулей.[2]

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. Какими свойствами должна обладать программа, чтобы после, например, окончания счёта по макрокоманде **`finish`** было возможно снова войти в начало программы по метке `Start`, не загружая эту программу заново в оперативную память?

Естественно, что, прежде всего, необходимо восстановить для программы первоначальный (пустой) стек, однако этого может оказаться недостаточно. Легко понять, что для головной программы на Ассемблере должно выполняться следующее условие: программа не должна менять свои кодовые сегменты и переменные с начальными значениями в сегментах данных, или, в крайнем случае, вос-

---

[1] Для продвинутых студентов заметим, что такой модуль можно перемещать с одного места памяти в другое не в любой момент, а только тогда, когда он не считается (например, если он вызвал внешнюю процедуру). В частности заметим, что нельзя перемещать модуль в другое место памяти во время выполнения им системного вызова по команде **`int i8`**, так как возврат из процедуры-обработчика прерывания производится по команде **`iret`**, которая, как мы знаем, в частности восстанавливает и значение регистра счётчика адреса `IP`.

[2] Подробную организацию виртуальной памяти студенты факультета вычислительной математики и кибернетики МГУ изучают в курсе третьего семестра "Системное программное обеспечение".

станавливать 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 (в этом случае говорят о вызове модуля), 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, а также регистры для работы с вещественными числами).

Главное отличие реентерабельных программ от обычных *рекурсивных* процедур заключается именно в том, что, в отличие от вызова программы, при *каждом* входе в реентерабельную программу порождается новая текущая точка её выполнения и порождается новое поле сохранения. Это позволяет продолжить выполнение реентерабельной программы с *любой* из этих нескольких текущих точек выполнения программы, восстановив значения всех её регистров из поля сохранения этой точки программы.[1]

На рис. 11.2 показан пример реентерабельной программы с тремя точками выполнения, отмеченными значениями регистра счётчика адреса IP. Какая-нибудь из этих точек в данный момент может быть активной, т.е. в ней находится центральный процессор, выполняя команды программы. Возможен, однако, и случай, когда все эти три вычислительных процесса находятся в состоянии *ожидания*, а центральный процессор занят выполнением какой-либо другой программы.

Принцип выполнения реентерабельных программ можно пояснить на таком шутливо-детективном примере. Вернувшись из очередного отпуска, следователь по особо важным делам майор Пронин принял к производству дело опасного преступника Х. Произведя первые допросы свидетелей и приобщив их к материалам дела, Пронин был вынужден прервать данное расследование, так как его попросили срочно заняться новым преступлением, которое предположительно совершил вор-рециди-

[1] Вообще говоря, здесь ситуация сложнее, чем мы её описываем. Для возобновления счёта программы с прерванного места необходимо, кроме регистров, восстановить из контекста процесса ещё значения специальных служебных переменных (их часто называют *переменными окружения* программы). В этих переменных, например, хранится информация обо всех файлах, с которыми работает программа. Ясно, что в разных точках модуля может производиться и обработка разных файлов. Таким образом, под контекст процесса необходимо отводить достаточно большую область памяти (порядка килобайта).

вист Y.[1] Найдя в новом деле первые улики и отправив их на экспертизу, которая должна была занять несколько дней, Пронин вернулся к делу рецидивиста X, но вскоре после этого ему позвонил его начальник полковник Петренко и приказал немедленно заняться делом серийного убийцы Z… Стоит пожалеть майора Пронина и разрешить ему хотя бы в воскресенье поехать на дачу и совсем не заниматься расследованиями ☺.

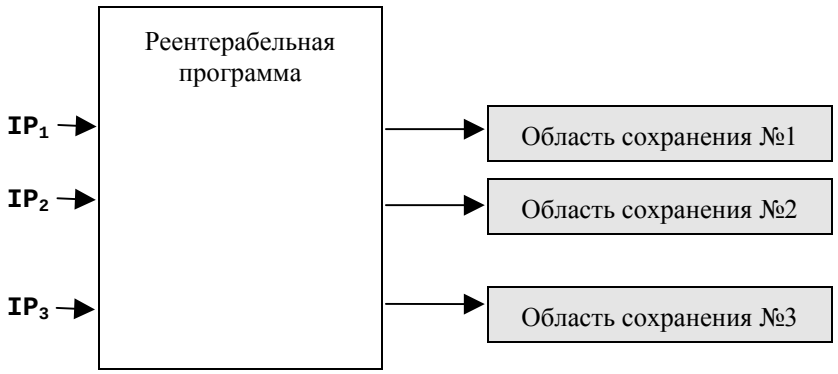


Рис. 11.2. Реентерабельная программа с тремя точками выполнения.

Как видим, следователь Пронин может возобновить расследование любого порученного ему дела, освежив свою память материалами из соответствующей папки (или из компьтерного файла, если майор Пронин шагает в ногу со временем). Точно так же и выполнение реентерабельной программы можно продолжить с любого прерванного места, так как вся необходимая для продолжения работы информация находится в соответствующей области сохранения. Как видим, в нашей аналогии майор Пронин играет роль центрального процессора, а дела, которые он ведёт – это процессы одной общей реентерабельной "программы расследования преступлений".

Упражнение. Что должен делать центральный процессор, когда его аналог майор Пронин отдыхает на своей даче и не занят расследованием преспуплений? Подсказка: как и в детективных фильмах отдых майора Пронина (а что это в нашем компьютере?) в любой момент может быть нарушен неожиданным звонком об очередном преступлении.

Таким образом, одна реентерабельная программа, находясь в памяти ЭВМ, может породить не один, а несколько самостоятельных (независимых друг от друга) *вычислительных процессов* (обычно слово "вычислительных" опускают, и называют их просто *процессами*). В нашем примере на рис. 11.2 для служебной программы, управляющей счётом задач (эта программа операционной системы чаще всего называется *диспетчером* процессов), для одной копии реентерабельной программы в памяти будут существовать три независимых единицы работы – три процесса.[2]

Ниже перечислены основные свойства, которыми должен обладать модуль на Ассемблере, чтобы быть реентерабельным.

- Модуль не изменяет свои сегменты кода.
- Модуль либо совсем не имеет собственных сегментов данных (т.е. все данные передаются ему в качестве параметров), 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 надёжно *защищённым* от изменения со стороны программ обычных пользователей.

[1] Мы предполагаем, что, подобно большинству людей, майор Пронин не может вести два дела по настоящему одновременно, подобно тому, как большинство ЭВМ, обладая только одним центральным процессором, в каждый момент времени может считать только одну программу. Людей, которые могут на самом деле одновременно заниматься несколькими разними делами так же мало, как среди всех ЭВМ очень мало многопроцессорных компьютеров (по преданию, одним таких людей был Юлий Цезарь, который мог слушать доклад одного чиновника, отдавать приказания другому и одновременно писать письмо по совершенно иной теме).

[2] Информация к размышлению: должен ли сам диспетчер тоже выполняется в виде процесса и, следовательно, управлять самим собой?

Таким образом, получается, что порождаемые из реентерабельной программы процессы имеют общие сегменты кода, но раздельные сегменты данных и стека.[1]

Реентерабельность является особенно важной при написании программ, входящих в состав операционных систем и систем программирования. Это следует из того, что если некоторая системная 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 программ – параллельно используемые).[2]

В современных ЭВМ большинство системных программ являются реентерабельными.

[1] Для совсем продвинутых студентов информация на будущее. На современных ЭВМ сейчас распространяется технология программирования, для которой при порождении из программы нового процесса он получает только свой собственный сегмент стека, а сегменты кода и данных будут общие для всех этих процессов. Такие процессы получили название легковесных процессов или *нитей* (threads). Так что некоторым из Вас придётся заниматься "многонитевым программированием" – звучит весьма необычно.

[2] Как мы узнаем позже, на однопроцессорной ЭВМ в каждый момент времени может компилироваться только одна программа, но, по сигналам прерывания от внутренних часов компьютера, можно производить быстрое переключение с одного вычислительного процесса на другой, так что создаётся впечатление, что компилируются (хотя и более медленно) сразу несколько программ.

Глава 12. Макросредства языка Ассемблер

Сейчас мы переходим к изучению очень важной и сложной темы – *макросредств* в языках программирования. С этим понятием мы будем знакомиться постепенно, используя примеры из макросредств нашего языка Ассемблера. Здесь следует подчеркнуть, что для полного понимания макросредств Ассемблера в нашем курсе обязательно требуется изучение учебника [5].

Вообще говоря, макросредства включены не только в язык Ассемблера, но и во многие языки *высокого* уровня. Заметим, однако, что мы не случайно будем изучать именно макросредства языка Ассемблера, а не макросредства каких-нибудь 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 всех тех возможностей, которые мы должны изучить в макросредствах. Именно поэтому для уровня университетского образования приходится рассматривать достаточно развитые макросредства в нашем языке Ассемблер.

Макросредства по своей сути являются *алгоритмическим языком*, встраиваемый в некоторый другой язык программирования,[1] который в этом случае будет называться *макроязыком*. Например, изучаемые нами макросредства встроены в язык Ассемблера, который поэтому называется Макроассемблером. Таким образом, программа на Макроассемблере в общем случае содержит в себе запись *двух* алгоритмов: один на макроязыке, а второй – собственно на языке Ассемблера.

Как мы знаем, у каждого алгоритма обязвтельно должен быть свой *исполнитель*. Например, исполнитель апгоритма на языке Паскаль в научной литературе называют Паскаль-машиной, это компьютер вместе с набором служебных программ (вспомните предыдущую тему "Понятие о системе программирования"). Исполнитель алгоритма на макроязыке называется *Макропроцессором*, а исполнителем алгоритма на Ассемблере является, в конечном счете, компьютер. Здесь важно, чтобы Вы не путали Макропроцессор с процессором компьютера: макропроцессор – это специальная программа, а не часть аппаратуры ЭВМ.

Результатом работы Макропроцессора (будем пока считать, что этот исполнитель работает *первым*, а компилятор с Ассемблера – вторым) является программный модуль на "чистом" языке Ассемблера, уже *без* макросредств. В связи с тим заметим, что в некоторых языках программирования (например, в языке С), макропроцессор так и называется – *препроцессор*, чтобы подчеркнуть, что при обработке программы он выполняется первым.[2] Иногда говорят, что Макропроцессор, получив входной модуль на языке Макроассемблера, *генерирует* модуль на Ассемблере, что хорошо отражает суть дела.[3]

На рис. 12.1 показана упрощенная схема работы Макропроцессора и компилятора с Ассемблера (как мы уже сказали, их часто называют общим именем – Макроассемблер). Программные модули пользователя на этом рисунке заключены в прямоугольники, а системные программы – в прямоугольники с закруглёнными углами.

Работа Макропроцессора по обработке макросредств исходного программного модуля называется *макропроцессированием*. Изучение макросредств языка Ассемблера мы начнём с уже знакомых

---

[1] Макросредства могут использоваться также и в формальных языках, не являющихся языками программирования, но этот вопрос далеко выходит за рамки нашей книги.

[2] В отличие от препроцессора языка С, который действительно полностью отрабатывает перед компилятором, взаимодействие Макропроцессора и Ассемблера при обработке программного модуля более сложное. На самом деле Макропроцессор и Ассемблер обрабатывают программу одновременно, предложение за предложением. Однако конечный этап компиляции – генерация объектного модуля – выполняется Ассемблером уже *после* полного завершения работы Макропроцессора. Мы пока не будем принимать это во внимание, будем считать, что сейчас мы рассматриваем работу Макроассемблера на *внешнем* уровне. Несколько более полно взаимодействие Макропроцессора и Ассемблера (уже на *концептуальном* уровне) мы рассмотрим в главе, посвящённой схеме работы компилятора Ассемблера. Здесь только отметим, что такая совместная работа Макропроцессора и компилятора Ассемблера и порождают те развитые возможности макросредств, которых нет в языках высокого уровня, где Макропроцессор работает именно как препроцессор (до начала работы самого компилятора).

[3] Подчеркнём, что программа Ассемблера относится к трансляторам (переводчикам) с языка Ассемблер на язык объектных модулей. В то же время Макропроцессор не стоит называть переводчиком с языка Макроассемблера на язык Ассемблера, так как текст на языке Ассемблера получается не в процессе переводя, а именно путем *генерации* этого текста как выходных данных алгоритма, заданного макросредствами.

нам *макрокоманд*. До сих пор мы говорили, что при обработке программы на Ассемблере на место макрокоманды по определённым правилам подставляется некоторый набор предложений языка Ас-

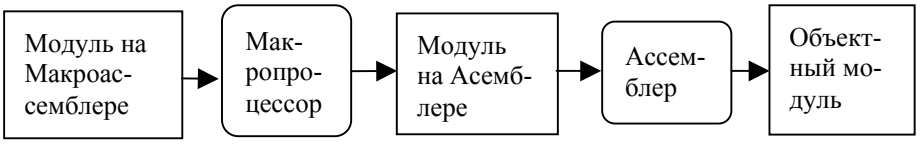


Рис. 12.1. Упрощённая схема работы Макроассемблера.

семблер. 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 Ассемблера. Коды операций макрокоманд являются *именами пользователя*, а все остальные коды операций – *служебными именами*.[1] Кроме того, у макрокоманды есть (возможно, пустой) список фактических параметров. Ниже приведён общий вид макрокоманды:

```
[<метка>:] <имя> [<список фактических параметров>]
```

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> [<список фактических параметров>]
```

эквивалентна двум таким предложениям Ассемблера:

```
<метка>:
        <имя> [<список фактических параметров>]
```

Обратите внимание, что метка не может быть без двоеточия, которое, как мы знаем, задаёт в нашем Ассемблере имя *команды*. Это совсем не означает, что макрокоманды не могут располагаться и, например, в сегменте данных, просто в этом случае в нашем Макроассемблере у них не может быть метки. В случае если макрокоманда располагается в сегменте данных, то на её место будет, вероятно, подставляться набор предложений Ассемблера, не являющихся командами (например, это могут быть предложения резервирования памяти или директивы). Один пример такой макрокоманды мы рассмотрим в конце этой главы.

Итак, каждая макрокоманда обязательно имеет имя. Обработка макрокоманды Макропроцессором начинается с того, что он, просматривая текст модуля от данной макрокоманды *снизу 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 снизу вверх от данной макрокоманды. Если в программе есть несколько макроопределений с одинаковыми именами, то говорят, что новое макроопределение *переопределяет* одноимённое макроопределение, описанное ранее.[2]

Макроопределение в нашем Макроассемблере имеет следующий синтаксис:

```
<имя> macro [<список формальных параметров>]
        Тело макро-
        определения
        endm
```

[1] В наших программах мы для удобства чтения выделяли имена макрокоманд жирным шрифтом (**finish**, **inint** и т.д.), хотя это, конечно, не совсем правильно, так как это не служебные слова Макроассемблера, а имена пользователя.

[2] Этот удобный механизм позволяет, например, заменять некоторое макроопределение, вставляемое в нашу программу по директиве `inclule io.asm`. Так, если программисту не понравится, как работает, например, макроопределение **inint**, он может написать своё собственное макроопределение с таким же именем и вставить его в текст своей программы за директивой `inclule io.asm`. Заметим также, что это один из немногих случаев, когда в одном ассемблерном модуле могут встречаться описания одинаковых имён.

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, в свою очередь, могут быть и предложения, относящиеся к макросредствам языка). Заканчивается макроопределение директивой **endm** (обратите внимание, что у этой директивы нет метки, как, скажем, у директивы конца описания процедуры или конца сегмента).

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. Более того, макроопределения, посвященные одной тематике можно объединить в один текстовый файл и хранить его *отдельно* от программных модулей. По аналогии с библиотеками объектных модулей, такие файлы часто называются библиотеками макроопределений. Например, используемые нами макроопределения для организации ввода/вывода объединены в библиотеку макроопределений, которая хранится в текстовом файле с именем `io.asm`. Как мы знаем, для того, чтобы вставить текст этого файла в программным модуль (и, таким образом, сделать содержащиеся там макроопределения доступными для макрокоманд ввода/вывода), используется директива Ассемблеру с именем **include**.

Итак, каждой макрокоманде должно быть поставлено в соответствие макроопределение с таким же именем, иначе в программе фиксируется синтаксическая ошибка (неописанной имя). Макроассемблер допускает *вложенность* одного макроопределения внутрь другого (в отличие от процедур, вложенность которых на Ассемблере, как мы уже знаем, не допускается), однако это обычно не нужно и редко используется в практике программирования.

Далее, как мы знаем, в макрокоманде на месте поля операндов может задаваться список *фактических параметров*. Например, мы можем написать макрокоманду `outint X,10` с двумя фактическими параметрами. 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 внешнее сходство между процедурами в Паскале и макроопределениями в Макроассемблере заканчивается, и начинаются различия в виде, способах передачи и обработки параметров.

Каждый фактический параметр макрокоманды является *строкой символов* (возможно пустой). Хорошим аналогом являются строки типа **String** в Турбо-Паскале, однако, фактические параметры макрокоманд, как правило, *не заключаются в апострофы*. Фактические параметры, если их более одного, разделяются запятыми или *пробелами*.[1] Если фактический параметр расположен не в конце списка параметров и является пустой строкой, то его позиция просто выделяется запятой, например:

```
Mymacro X,,Y; Три фактических параметра, второй пустой
Mymacro ,A B; Три фактических параметра, первый пустой
```

Как видим, в отличие от Паскаля, все параметры макроопределения одного типа – это (возможно пустые) строки символов, другими словами, всегда есть *соответствие по типу* между фактическими и формальными параметрами. Далее, в Макроассемблере, в противоположность языку Паскаль, 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 не используются (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 (третий) способ передачи фактических параметров макрокоманды в макроопределение, его нет в Паскале. Этот способ называется передачей *по написанию* (иногда

[1] Как видим, запятая, пробел (и некоторые другие символы) являются служебными и не могут просто так встречаться внутри строки-параметра. При необходимости вставить служебный символ внутрь параметра макрокоманды используются особые приёмы, о которых мы будем говорить далее.

– передачей по имени). При таком способе передачи параметров все *имена* формальных параметров в теле макроопределения заменяются соответствующими им фактическими параметрами (строками символов).[1] Возможна замена формального параметра на фактический даже внутри текстовых строк, правда, для этого имя формального параметра необходимо выделить специальным макроограничителем &, например:

```
P     macro X
      db    "Параметр X=&X"
```

После передачи параметров начинается просмотр тела макроопределения и поиск в нём предложений, содержащих макросредства, например, макрокоманд. Все предложения в макроопределении, содержащие макросредства, обрабатываются Макропроцессором так, что в результате получается набор предложений на "чистом" языке Ассемблера (уже без макросредств), такой набор предложений 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.

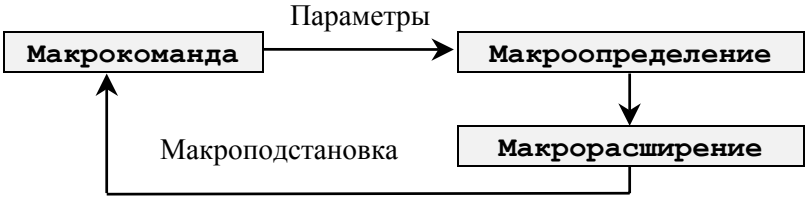


Рис. 12.2. Схема обработки макрокоманды.

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 обычных команд Ассемблера, но, в отличие от них, 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. Таким образом, уровень языка Ассемблера, по существу, приближается к языкам высокого уровня.

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 (при произвольном задании операндов `z,x` и `y` – в памяти, на регистрах или в виде констант) 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 кодом операции `Sum`:

[1] Это несколько упрощённое описание действий Макропроцессора при передаче параметров, позже мы сделаем в этом месте существенные уточнения.

```
Sum Z,X,Y; Z:=X+Y
```

Уточним синтаксис нашей новой команды, т.е. зафиксируем допустимые форматы её операндов. Потребуем, чтобы первый операнд этой команды мог иметь форматы `r16` и `m16`, а второй и третий операнды – форматы `i16,m16` и `r16` (в любой комбинации). Такой команды, как мы знаем, в нашем компьютере нет, но можно создать новую *макрокоманду*, которая работала бы так, как нам надо.[1] 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, скажем, описания переменных

```
A     dw    ?
B     dw    ?
C     dw    ?
```

и надо выполнить присваивание `C:=A+B`, то программист может записать это действие в виде одного предложения Ассемблера – макрокоманды

```
Sum   C,A,B
```

Увидев такую макрокоманду, Макропроцессор, просматривая текст программы снизу-вверх, найдёт соответствующее макроопределение с именем `Sum`, и, после замены формальных параметров на фактические, построит следующее макрорасширение:

```
mov   ax,A
add   ax,B
mov   C,ax
```

Это макрорасширение и будет подставлено в текст нашей программы вместо макрокоманды **`Sum C,A,B`** (произойдёт *макроподстановка* полученного макрорасширения на место макрокоманды).

Программист доволен: теперь *исходный* 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. Кроме того, можно заметить, что с помощью макрокоманд мы как бы меняем *архитектуру* нашего компьютера, например, у нас появилась трёхадресная команда сложения, которой нет в языке машины.

Далее, однако, программист может заметить, что некоторые макрокоманды работают не совсем хорошо. Например, на место макрокоманды с *допустимым* форматом параметров

```
Sum   C,ax,B; формат m16,r16,m16
```

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 (так как в программах ценится и их красота ☺), 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*.

---

[1] Любопытно отметить, что такой команды не было и в нашей учебной трёхадресной машине УМ-3, где допускался только прямой способ пдресации, т.е. операнды команды могли быть только адресами (в нащей теперешней терминологии `m16`). В то же вреемя наша новая макрокоманды допускает смешанную адресацию, когда второй и третий операнды могут быть прямыми или непосредственными (`m16` или `i16`).

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 (или не 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 фактический 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 с именем **`ifdif`** (if different). 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
```

Проверьте, что теперь Макропроцессор не будет вставлять в макрорасширение ненужной команды **`mov ax,ax`**.

Изучая дальше наше макроопределение можно заметить, что и на место, например, макрокоманды

```
Sum   ax,ax,13
```

с допустимыми форматами операндов `r16,r16,i16` подставится макрорасширение

```
add   ax,13
mov   ax,ax
```

с лишней последней строкой, что тоже некрасиво. 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-о-очень хорошее макрорасширение

```
add   ax,13
```

Дальнейшее изучение нашего макроопределения, однако, выявит новую неприятность: макрокоманда

```
Sum   ax,Y,ax
```

с допустимыми форматами операндов `r16,m16,r16` порождает *неправильное* макрорасширение

```
mov   ax,Y
```

```
add   ax,ax
```

Легко понять, что это, конечно, не то же самое, что `ax:=Y+ax`. Как можно заметить, источник неприятности здесь состоит в том, что третий параметр нашей макрокоманды испортился до того, как мы его использовали по назначению. С этой проблемой тоже можно справиться, 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** (if ifentical) 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 нашего последнего примера вызова 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 и неравенство 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 текста <AX> и <ax> *не идентичными*, так как полагает внутри строки текста большие и маленькие буквы различными, со всеми вытекающими отсюда последствиями. В то же время служебные имена регистров AX и ax в языке Ассемблера, как мы знаем, считаются *идентичными*, и у нас нет причин ограничивать пользователя в написании таких имён, как ему захочется. С этой проблемой нам придётся что-то делать…

Другая трудность подстерегает нас, если мы попытаемся использовать в нашей программе, например, такую макрокоманду

```
Sum   ax,bx,dl
```

После обработки 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 следующем этапе, когда Ассемблер будет анализировать синтаксическую правильность этих предложений Ассемблера, для команды `add ax,dl` зафиксируется ошибка – несоответствие типов операндов. Это очень важный момент – ошибка зафиксирована не при обработке Макропроцессором *неправильной* макрокоманды `Sum ax,bx,dl`, как происходит при проверке синтаксической правильности обычных команд Ассемблера, а позже, и уже при анализе не макрокоманды, а макрорасширения. В этом отношении наша макрокоманда *уступает* обычным командам Ассемблера, так как по аварийной диагностике труднее определить место и характер ошибки. Нам бы, конечно, хотелось, чтобы диагностика об ошибке (и лучше на русском языке) выдавалась

уже на этапе обработки макрокоманды Макропроцессором. Например, для неправильной макрокоманды

```
Sum   ax,bx,dl
```

Для программиста много лучше получить диагностику

```
*** У Sum плохой тип третьего параметра,
```

чем такую диагностику:

```
add   ax,dl
ASM(329): error 31: Operand types must match
```

Итак, мы обратили внимание на две трудности, которые, как Вы догадываетесь, можно преодолеть, снова усложнив наше макроопределение. Мы, однако, сделаем это не на примере простой макрокоманды **Sum** для суммирования двух чисел, а на примерах других, более сложных, задач.

Рассмотрим теперь типичный ход мыслей программиста. Когда у него может возникнуть необходимость в написании новой макрокоманды? Например, во многих программах необходимо выдавать многочисленные диагностические сообщения. Как мы знаем, для того, чтобы в нашем 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 много удобнее выводить строку текста, например, так (текстовые строки в Ассемблере можно заключать как в двойные кавычки, так и в апострофы):

```
outtxt "Строка для вывода"
```

Осознав такую потребность, программист решает написать новое, своё собственное, 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
      outstr
      pop   dx; восстановление dx
      pop   ds; восстановление ds
      endm
```

В этом макроопределении мы использовали новое для нас 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, которое в этом смысле играет роль блока, следовательно, в других частях этого программного модуля также могут использоваться эти имена. Директива **local** 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, хотя программисту и не рекомендуется использовать такие имена, чтобы не конфликтовать с именами, автоматически порождаемыми Макропроцессором.

Например, для макрокоманды

```
      outtxt 'Привет!'
```

будет построено макрорасширение

```
            jmp   ??0001
    ??0002  db    'Привет!'
            db    '$'
    ??0001: push  ds; запоминание ds
            push  cs
            pop   ds; ds:=cs
            push  dx; сохранение dx
            mov   dx,offset ??0002
            outstr
            pop   dx; восстановление dx
            pop   ds; восстановление ds
```

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. Обязательно поймите, что при использовании директивы **local** внутри программного модуля на Ассемблере теперь не стало *одинаковых* имён.

Обратите внимание, что второй фактический параметр (строку символов) – наше макроопределение располагает внутри макрорасширения (т.е. в сегменте кода). А так как макрокоманда **outstr** "думает", что выводит текст из сегмента данных, то мы *временно* совместили сегменты данных и кода, загрузив в регистр ds значение регистра cs.

После изучения написанной нами макрокоманды **outtxt** у некоторых студентов может сложиться *ошибочное* впечатление, что теперь *во время счёта* программы *изменяется* её сегмент кода (т.е. наша программа стала самомодифицирующейся). Обязательно поймите, что это не так!

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, например с именем **oint**, 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! [1]
      .err
      exitm
endif
      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
```

[1] Будьте очень осторожны в употреблении *русских* букв в текстовых строках для вывода и даже в комментариях, так как наш Ассемблер MASM 4.0 относится к таким буквам очень "болезненно", иногда принимает за какие-то служебные символы и часто просто зацикливается при компиляции. В связи с этим внутри макроопределений рекомендуется использовать только латинские буквы, в наших примерах мы не делали этого из соображений наглядности. Ну, не предназначен (или как теперь модно говорить, не *локализован*) этот Ассемблер для российских программистов ☺.

```
      exitm
  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 очень 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** (if blank), который вырабатывает значение **true**, если в угловых скобках 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**.

После того, как макроопределение обнаружит ошибку в своих параметрах, у программиста есть две возможности дальнейших действий. 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, в этом случае, естественно, будет невозможен и выход программы на счёт. Заметим, однако, что при возникновении фатальной ошибки Ассемблер, тем не менее, будет продолжать проверку остальной части входного модуля 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 выдаст на экран и 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).[1]

После возникновения фатальной ошибки и выдачи директивы **.err** дальнейшая обработка макроопределения не имеет уже никакого смысла, и мы запретили эту обработку, выдав Макропроцессору директиву **exitm**. Эта директива в нашем случае прекращает процесс построения макрорасширения,[2] 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:

```
      %out Нет аргумента в oint!
      .err
```

[1] В следующей главе мы кратко рассмотрим схему работы Ассемблера и узнаем, что наш Ассемблер при компиляции дважды просматривает текст исходного модуля. Директивы **%out** и **.err** выполняются при каждом просмотре Ассемблером текста программы, то есть определяемая ими диагностика выводится дважды. Этого можно избежать, используя специальные условные макрооператоры с именами **if1** и **if2** и директивы **err1** и **err2** (см. учебник [5]).

[2] Точнее, директива **exitm** производит выход вниз за ближайшую директиву **endm**, которая, как мы вскоре узнаем, может задавать конец не только макроопределения, но и макроциклов. Таким образом, **exitm** прекращает обработку той части макроопределения, которая ограничена ближайшей при просмотре вниз директивой **endm**.

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. Когда Ассемблер завершает компиляцию программного модуля, переменные периода генерации уничтожаются, так как заканчивается выполнение самой программы Ассемблера.

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 – после выхода из того макросредства, в котором они порождены. В нашем Макроассемблере различают локальные переменные периода генерации *макроопределения* (они порождаются при входе в макроопределение по директиве **`local`**, и уничтожаются после построения макрорасширения), и локальные переменные – параметры макроциклов 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, а параметры макроциклов – ещё и 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, он очень похож на цикл с параметром языка Паскаль 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, в отличие от переменных периода генерации, определяемых пользователем как показано выше, принимает *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

[1] Заметим, что в языках высокого уровня обычно есть далеко не все из рассматриваемых нами макросредств.

```
<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>
```

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 наших примерах мы для удобства изобразили его в две строки. Выше, при написании текста макроопределения **oint** мы записали 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 (первая строка текста), после чего выполняется тело цикла, при этом все вхождения в это тело параметра цикла заменяются на текущее значение этой переменой периода генерации. После этого параметру цикла присваивается следующее значение из списка цикла и т.д. Ясно, что для пустого списка тело цикла не будет выполняться ни одного раза. В нашем примере тело цикла будет выполняться 32 раза, при этом переменная `i` будет последовательно принимать значения двухсимвольных строк текста `al,ah,bl` и т.д.

Как можно заметить, целью выполнения макроцикла в нашем примере является присваивание переменной периода генерации `K` значения единицы, если параметр макрокоманды совпадает по написанию с именем одного из коротких регистров нашего 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.[1] После присваивания переменной периода генерации значения `K=1` дальнейшее выполнение цикла бессмысленно, и мы выходим из него по директиве **exitm** за ближайшую вниз директиву **endm.**

Обратите внимание, что макроцикл завершается такой же директивой **endm**, как и всё макроопределение. Это позволяет рассматривать макроцикл как некоторое *стандартное* макроопределение, только без заголовка (т.е. без имени и без параметров).

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
```

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 используются, как и в некоторых других языках (например, в Фортране), мнемонические двухбуквенные имена, которые перечислены ниже:[2]

EQ	вместо	=
NE	вместо	<>
LT	вместо	<
LE	вместо	<=
GT	вместо	>

[1] Для удобства программирования в некоторые Макроассемблеры введены условные макрооператоры, которые сравнивают две строки на равенство и неравенство без учёта регистра (при этом большие и маленькие латинские буквы не различаются). Например, такие условные макрооператоры есть в старших версиях нашего Макроассемблера MASM, их использование, конечно, существенно облегчает программирование макроопределений.

[2] В отличие от Паскаля, эти операции отношения можно применять только к целочисленным операндам. Другими словами нельзя, например, написать сравнение на равенство текстовых строк

```
if <X> EQ <ax>
```

Для вычисления логического выражения от строковых значений, как мы уже знаем, используются специальные условные макрооператоры (**ifb**, **ifidn**, **ifdif** и другие).

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 для этого примера 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 задан выражением, которое имеет формат `m8`). В остальных случаях (для параметра форматов `r16,m16,i16`) 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 правило, рассматривают свои операнды как беззнаковые значения. Исключением является случай, когда Макропроцессор "видит", что некоторой переменной периода генерации *явно* присвоено отрицательное значение (т.е. в присваиваемой константе есть знак минус). Например, рассмотрим следующий фрагмент программы:

```
L:    mov ax,ax; Чтобы была метка, type L=-1
      K = type L
; Макропроцессор "видит" беззнаковое L=0FFFFh
if K LT 0; Берётся K=0FFFFh > 0 ==> false !
            . . .
      K = -1
; Макропроцессор "видит" знаковую константу -1
if K LT 0; Берётся K=-1 < 0 ==> true !
            . . .
```

Как видим, этот вопрос в нашем Макроассемблере 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

```
mov  ax,--8
outint ax
```

[1] Для простоты наше макроопределение никогда не задаёт второй параметр макрокоманды **outint** – ширину поля для вывода целого числа.

(т.к. **type** --8 = 0). К сожалению, наш Макропроцессор не предоставляет хороших средств, позволяющих выявить синтаксические ошибки такого рода в макрокомандах.[1] Показанные выше ошибки будут выявлены уже компилятором с Ассемблера при анализе полученного макрорасширения.

Далее, Вам важно понять принципиальное отличие переменных языка Ассемблера и переменных периода генерации. Так, например, переменная Ассемблера с именем X может, например, определяться предложением резервирования памяти в сегменте данных

```
X      dw     13
```

В то время как переменная периода генерации с именем Y может порождаться макрооператором присваивания

```
Y = 13
```

Главное – это уяснить, что эти переменные имеют разные и непересекающиеся времена́ существования. Переменные периода генерации существуют только во время компиляции исходного модуля с языка Ассемблер на объектный язык (т.е во время генерации объектного модуля, отсюда и их название), и заведомо уничтожаются до начала счёта. В то же время статические переменные Ассемблера, наоборот, существуют только во время счёта программы (от начало счёта до выполнения макрокоманды **finish)**. Некоторые студенты не понимают этого и пытаются использовать значение переменной 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. Необходимо понять, что на этапе компиляции значение есть не у *переменной* с именем X, а только у *имени* переменной X (значение имени X равно адресу этой переменной в соответствующем сегменте). Кроме того, как мы знаем, на этапе компиляции любое имя имеет *тип*, например, наше имя X имеет тип **word**=2).

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 в нашем Макроассемблере можно написать макроопределение, у которого будет только *один* формальный параметр, на место которого будет, однко, 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.

Сделаем спецификацию нашей макрокоманды. Мы будем допускать, чтобы некоторые параметры из списка опускались (т.е. задавались пустыми строками). При поиске максимума такие пустые параметры будем просто игнорировать, и не выдавать при этом никакой диагностики. Далее необходимо договориться, что будет делать макрокоманда, 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 **cmpax**. Это вспомогательное макроопределение загружает на регистр ax максимальное из двух величин: регистра ax и своего единственного параметра X.

---

[1] Любознательные студенты могут попробовать диагностировать такого рода ошибки при помощи макроцикла **irpc**, который мы вскоре рассмотрим.

```
; Вспомогательное макроопределение
cmpax macro X
      local L
      cmp   ax,X
      jge   L
      mov   ax,X
L:
      endm
; Основное макроопределение
; Максимум переменного числа аргументов
maxn  macro X
      mov   ax,8000h; MinInt
irp i,<X>
   ifnb <i>
      cmpax i
   endif
endm
      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 Макроассемблере *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. Обратите внимание, что отбрасывается только одна пара угловых скобок, если внутри фактического параметра есть ещё угловые скобки, то они сохраняются.
- 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 фактическом параметре сами служебные символы.
- В том случае, если комментарий начинается с двух символов `;;`, то это *макрокомментарий,* такой комментарий, в отличие от обычного комментария (начинающегося одним символом `;`) *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,<1,h>
        K=K+1
        mov   a&i,X&K&i
endm
```

  После обработки этого макроцикла Макропроцессор подставит на его место в текст программы следующие строки:

```
        mov   al,X21
        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 поиска максимума от нескольких аргументов **maxn** на примере макрокоманды

```
maxn  <-13,,bx,Z>
```

При передаче фактического параметра-строки `<-13,,bx,Z>` крайние угловые скобки будут отброшены, поэтому в макроцикле **irp** на место формального параметра `X` (внутрь угловых скобок) 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** (if not blank), 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 **cmpax**. Единственным назначением этого вспомогательного макроопределения является локализация в нём имени метки `L`, которая таким образом получает уникальное значение для каждого параметра из списка цикла.

Обратите внимание, что наше макроопределение будет *неправильно* работать для макрокоманд, содержащих в списке параметров регистр `ax`, например

```
maxn  <-13,ax,bx,Z>
```

Упражнение. Используя средства условной генерации (условные макрооператоры), исправьте макроопределение **maxn** так, чтобы оно правильно обрабатывало регистр `ax` в списке своих параметров.

Упражнение. Напишите макроопределение **maxn** *без* использования вспомогательного макроопределения **cmpax**. 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, чем в макросредствах Ассемблера) есть, например, в языке высокого уровня С, который допускает написание функций с переменным числом фактических параметров. Правда, для функций языка С фактических параметров должно быть не менее одного, и значение этого *первого* параметра обязано каким-то образом однозначно задавать для функции *общее* число фактических параметров.

Напомним, что в языке Паскаль программист не может писать свои собственные процедуры и функции с переменным числом параметров, так как, по мнению автора этого языка Н. Вирта, это снижает надёжность программирования. В то же время совсем обойтись без такого очень удобного средства Паскаль не мог: в нём есть *стандартные* процедуры с переменным числом параметров, например, это процедуры ввода/вывода `Read` и `Write`.

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 самого Ассемблера, если параметр содержит некоторое *имя*, то у этого параметра может быть *тип*. Например, фактическим параметром может быть *имя* длинной или короткой целой переменной, константы и т.д. и ясно, что для обработки операндов разных типов в Ассемблере требуются и различные форматы

команд. Другими словами, программисту может понадобится внутри макроопределения выбрать (средствами условной генерации) один из нескольких форматов команды (например, команды сложения) для обработки операндов разных типов, переданных в это макроопределение в качестве фактических параметров.

Как мы помним, для стандарта языка Паскаль у процедур и функций должно соблюдаться строгое соответствие между типами формальных и фактических параметров.[1] 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.[2]

Сейчас мы увидим, что макросредства предоставляют программисту простое и элегантное решение, позволяющее *настраивать* макроопределение на тип фактических параметров. Рассмотрим следующую задачу. 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 `X` этого макроопределения можно задавать массивы коротких или длинных *знаковых* целых чисел, количество элементов в этом массиве указывается во втором параметре `N`. Другими словами, первый операнд макрокоманды **SumMas** может быть формата `m8` или `m16`, а второй – формата `m16,r16` 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 будем считать, что в нашем массиве `100` элементов (т.е. предположим, что массивы такой длины наиболее часто встречаются в нашей программе, и нам просто лень задавать в этом случае второй параметр макрокоманды). Для простоты изложения 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 (Вы должны уже представлять, как можно сделать такую проверку на допустимость параметра). Ниже показан возможный вариант такого макроопределения.

```
SumMas macro X,N
      Local K,L
ifb <X>
      %out Нет массива!
      .err
      exitm
endif
ifb <N>
      %out Берём длину=100
```

---

[1] Исключение составляют некоторые *стандартные* процедуры и функции Паскаля. Например, стандартная функция `abs(X)` может принимать параметр как целого, так и вещественного типов, выдавая, соответственно, целый или вещественный результат. Конечно, надо понимать, что в этом случае есть две разные функции с одним именем `function abs(x:real):real;` и `function abs(x:integer):integer;`, и Паскаль-машина просто выбирает, какую из них вызвать, в соответствие с типом фактического параметра. Такие функции называются в языках программирования *полиморфными* (т.е. "много формовыми", а точнее было бы назвать их "много смысловыми"). Заметим, что в *объектно-ориентированных* языках программист может писать и свои *собственные* полиморфные процедуры и функции.

[2] Замечание для продвинутых студентов. В стандарте Паскаля можно попытаться сделать элементы такого 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 типов и следующее отсюда значительное 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.

```
      mov   cx,100
else
      mov   cx,N
endif
      K=type X
if K LT 1 or K GT 2
      %out Плохой тип массива!
      .err
      exitm
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),[1] то есть на машинном языке получается эффективная программа, настроенная на нужный тип данных и 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 и функции со стандартным соглашением о связях, передавая им каждый раз достаточно большое количество параметров. Естественно, программист хочет автоматизировать процесс вызова, написав макроопределение, которое позволяет вызывать процедуры и функции 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 в Паскале можно, например, вызвать оператором процедуры `P(X,400,Y)`. В Ассемблере для фактических параметров, описанных, например, так:

```
X     dw    400 dup (?)
Y     dw    ?
```

вызов такой процедуры со стандартным соглашением о связях, как мы знаем, будет, например, производится последовательностью команд:

---

[1] Такая настройка оказалась возможна только потому, что компилятор Ассемблера и Макропроцессор работают "рука об руку", поочерёдно обрабатывая в программе предложение за предложением. Именно поэтому при выполнении оператора `type X` внутри макроопределения компилятору уже известен тип имени фактического параметра, описанного где-то выше в тексте программы. Более подробно как это делается мы узнаем в главе, посвящённой схеме работы компилятора с Ассемблера.

```
        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 на Ассемблере 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. Ну, и уж для совсем любителей красоты и языка Паскаль предложим такой вызов процедуры:

```
CallProc P,<var X,400,var Y>
```

Этот вызов обеспечивается макроопределением с именем **CallProc**, внутри которого описано вспомогательное макроопределение с именем **InStack**:

```
CallProc macro Name,Param
;; Вспомогательное макроопределение
   InStack macro X,Y
   ifb <Y>; нет var
      mov ax,X
   else
      mov ax,offset Y
   endif
      push ax
      endm
;; Основное макроопределение
irp i,<Param>
      InStack i
endm
      call Name
      endm
```

Упражнение. Самостоятельно разберите, как работает это макроопределение и как его можно изменить для обеспечения контроля правильности передаваемых ему фактических параметров.

Как бы то ни было, главная наша цель достигнута, теперь вызов процедуры стал производиться в одно предложение Ассемблера с хорошо понятным смыслом. 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 другая строка символов, скажем такая, как `[bp+6]`. Поставленную задачу можно решить, например, с помощью такого фрагмента на Макроассемблере:

```
      Nom=1; Первый символ в имени
irpc i,<Name>
      Err=1; Признак ошибки в очередном символе имени
   if Nom EQ 1
      irpc j,<abcdefghijklmnopqrstuvwxyz
               ABCDEFGHIJKLMNOQPRSTUVWXYZ_?@#$>
            ifidn <i>,<j>
                  Err=0
                  exitm
            endif
      endm
   else
      irpc j,<abcdefghijklmnopqrstuvwxyz
               ABCDEFGHIJKLMNOQPRSTUVWXYZ_?@#$0123456789>
            ifidn <i>,<j>
                  Err=0
                  exitm
            endif
      endm
   endif
   if Err EQ 1
      exitm; Выход из цикла irpc
   endif
      Nom=Nom+1; Номер символа в имени
endm
if Err EQ 1
      %out Плохой Nom символ 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 из строки, символы которой и являются списком цикла. В нашем примере в списке цикла мы указали в первом операторе **irpc** все символы, с которых может *начинаться* имя в Ассемблере (как и в Паскале, это латинские буквы и некоторые символы, приравнивающиеся к буквам). Во втором операторе **irpc** из нашего примера мы просто добавили к этим символам ещё и 10 цифр. Заметим, что, как уже отмечалось ранее, каждое предложение Ассемблера 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 программой Ассемблера 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 (меткой) команды? Чтобы выяснить это, в наше макроопределение можно вставить проверку *типа* полученного имени, например, так:

```
if type Name NE near and type Name NE far
   %out Name не метка команды!
   .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):

.type <имя> = { 0 - имя команды или процедуры,
1 - имя переменной,
5 - имя как-то определено,
7 - имя описано в **extrn**.

Так, например, для имени, описанного в Ассемблере как

```
X      dw     ?
```

оператор **.type** X=00100010b=34_{10}**.** Полностью про этот оператор нужно прочитать в учебнике по Ассемблеру [5].

В заключение нашего краткого знакомство с возможностью макросредств языка Ассемблер разберём пример, когда макрокоманда используется не в сегменте команд, а, например, в сегменте данных. Предположим, что программисту приходится много работать с целочисленными скалярными матрицами различной размерности, с разным типом и значением элемента на главной диагонали.[1] Каждая скалярная (а, следовательно, квадратная) матрица в Ассемблере полностью определяется следующими атрибутами:

- размерностью матрицы N>1, будем считать этот параметр макроопределения *обязательным*;
- значением элемента на главной диагонали P, если он не задан, будем предполагать единичную матрицу;
- типом элемента матрицы Tip, в Ассемблере это **db,dw** или **dd**, если параметр опущен, будем предполагать матрицу слов (**dw**);
- именем матрицы Name (точнее, это имя первого элемента матрицы), если имя не задано, то под матрицу будет зарезервирована *безымянная* область памяти. Поймите, почему имя Name нельзя сделать меткой, поставленной перед макрокомандой.

Следовательно, для порождения, например, единичной матрицы с именем My_Mat из двойных слов размером 5x5 можно использовать макрокоманду

```
ScalMat 5,,dd,My_Mat
```

Соответствующее макроопределение может выглядеть таким образом**:**

```
ScalMat macro N,P,Tip,Name
      local R
ifb <N>
      %out Нет размера! [2]
      .err
```

[1] Напомним, что квадратная матрица является скалярной, если все её элементы на главной диагонали равны некоторой константе, а все остальные элементы равны нулю. Простейшим примером скалярной матрицы является единичная матрица.

[2] Напомним, что русские буквы в строках надо использовать с большóй осторожностью, наш "иностранный" Ассемблер MASM 4.0 может воспринимать их плохо и даже зацикливаться, мы использовали диагностику на русском языке только для наглядности изложения материала.

```
      exitm
endif
if type N NE 0 or N LT 2
      %out Плохая размерность!
      .err
      exitm
endif
ifb <P>
      %out Единичная матрица!
      R=1
else
      R=P
endif
ifb <Tip>
      %out Матрица слов!
Name  dw R,(N-1) dup (N dup (0),R)
else
Name  Tip R,(N-1) dup (N dup (0),R)
endif
      endm
```

Вот теперь в сегменте данных (как, впрочем, и в любом другом сегменте) можно резервировать области памяти под скалярные матрицы нашими макрокомандами, например:

```
Data  segment
X     dw    ?
      ScalMat 10,-1,db,Neg_10x10_Matrix
      . . .
Data  ends
```

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 3 или большее число 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`, Вы уже знаете, как можно исправить такую ошибку):

```
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, да ещё и использует стековый кадр. 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*10=200 команд. В случае же реализации алгоритма в виде процедуры (пусть её длина будет не 10, а даже 20 команд) и 20-ти вызовов этой процедуры нам потребуется всего 20*4+20=100 команд. Получается, что для достаточно сложных алгоритмов реализация в виде процедуры более выгодна, что легко понять, если учесть, что процедура присутствует в памяти 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 для гибкого и надёжного программирования весьма трудно и неэффективно реализовываются с помощью процедур функций.

Исходя из вышеизложенного, наиболее перспективным является *гибридный* метод. При этом надо реализовать алгоритм в виде макроопределения, в котором производится настройка на типы параметров и выдачу диагностики (как мы видели, такие действия не порождают дополнительных команд в полученной программе), а потом, если нужно, вызывается *процедура* для реализации основной части алгоритма. Именно так устроены достаточно сложные макроопределения для ввода и вывода целых чисел **inint** и **outint**, которыми Вы часто пользуетесь (с макроопределениями для этих макрокоманд можно ознакомиться по книге [5]).

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.

Упражнение. Обоснуйте, почему перед макрокомандой нельзя ставить метку без двоеточия (т.е. метку области памяти), хотя саму макрокоманду можно использовать, например, в сегменте данных. Для этого нужно вспомнить, как обрабатываются макрокоманды с метками.

[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 программирования 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.

Глава 13. Схема работы транслятора с языка Ассемблера

Сейчас мы рассмотрим, как транслятор преобразует входной модуль на "чистом" языке Ассемблера (уже *без* макросредств, которые обработал Макропроцессор) в объектный модуль.[1] Разумеется, мы изучим только общую схему этого достаточно сложного процесса. Наша цель – рассмотреть основные принципы работы транслятора с языка Ассемблера и ввести необходимую терминологию, а также снять тот покров таинственности с работы программы-переводчика, который может быть у некоторых студентов. Более подробно этот вопрос, который относится к большой теме "Формальные грамматики и методы компиляции", изучается в другом курсе.

Итак, как мы знаем, Ассемблер относится к классу системных программ, которые называются трансляторами[2] – переводчиками с одного алгоритмического языка на другой. Наш транслятор является компилятором, он 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. Выдачу листинга при желании, как правило, можно заблокировать.
- Генерация объектного модуля.

Разберём более подробно первый этап – анализ программы на наличие ошибок. Ясно, что найти ошибки 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 для реализации, 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 для каждого предложения Ассемблера работает специальная программа транслятора, которая называется *лексическим анализатором*. Она разбивает каждое предложение программы на *лексемы* – логически неделимые единицы языка. О лексемах мы уже должны немного знать из изучения 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 символы. Все имена Ассемблера делятся на *служебные* и имена пользователя. В отличие от Паскаля, в Ассемблере нет *стандартных* имён (напомним, что стандартное имя имело заранее определённый смысл, но эти имена в Паскале можно было *переопределить*).

---

[1] Вообще говоря, как мы уже говорили ранее, 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.

[2] Термин транслятор обозначает программу-переводчика с одного (формального) 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. В гнглийском языке для этого служат два разных глагола: `translate` и `inperpret`.

- 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. Исключением является строка-параметр директивы эквивалентности, например, `X equ ABC++`. Напомним, что выполнение этой директивы требует замены во всех следующих предложениях имени `X` на строку символов `ABC++`. После такой замены потребуется повторное разбиение строки на лексемы. Кроме того, 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 точкой 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 выполнение алгоритма, заложенного в программу, они переносятся в листинг, а из анализируемого текста удаляются. Не являются исключением макрокомментарии, которые начинаются с двух символов `';;'`, как мы знаем, такие комментарии не переносятся в макрорасширения и попадают в листинг программы в одном экземпляре (т.е. только внутри макроопределений).

В качестве примера ниже показано предложение Ассемблера, каждая лексема в нём выделена в прямоугольник, обратите внимание на лексему-пробел между котом операции **mov** и первым операндом `ax`:

`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:[1]

```
134X        'A'38       B"C
```

Все опознанные (правильные) лексемы записываются на первом проходе в специальную *таблицу лексем*. В этой таблице вместе с каждой лексемой указывается её класс и некоторые другие атрибуты. Это делается в основном для того, чтобы на втором проходе транслятор мог уже двигаться по программе *по лексемам*, а не по отдельным составляющим их символам, что позволяет существенно ускорить просмотр текста модуля.

После разбиения предложения Ассемблера на лексемы начинается второй этап обработки предложения – этап *синтаксического анализа*. Этот этап выполняет программа транслятора, которая называется *синтаксическим анализатором*. Алгоритмы синтаксического анализа весьма сложны, и здесь мы не будем их подробно 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 основные атрибута, ниже перечислены эти атрибуты (не у каждого имени есть все из них). Для удобства изложения этим четырём атрибутам присвоены имена `Segment`, `Offset`, `Type` и `Value`.

- Атрибут сегмента `Segment`. Этот атрибут имеет формат `i16`, он задаёт адрес начала (делённый на `16`) того сегмента, в котором описано или объявлено данное имя.[2]

---

[1] Здесь надо сказать, что компилятор с Ассемблера не предполагает глубокого знания программистом теории компиляции, поэтому в листинге программы такие ошибки называются общим термином "синтаксические ошибки", хотя, как мы узнаем немного позже, "настоящие" синтаксические ошибки определяются другой частью компилятора – синтаксическим анализатором.

[2] Напоминаем, что если имя пользователя встречается в позиции метки или в параметрах директивы **`extrn`**, то это его *описание* или *объявление*, а если имя встречается в поле операндов или в коде операции (для имён макрокоманд) – то это *использование* имени. Все используемые в программном модуле имена пользователя, естественно, должны быть описаны или объявлены в этом же модуле.

- Атрибут смещения `Offset`, он также имеет формат `i16` и задаёт смещение *описания* имени от начала того сегмента, в котором оно описано. Атрибуты `Segment` и `Offset` могут иметь только имена, определяющие области памяти и метки команд.
- Атрибут типа `Type`. С этим атрибутом имени мы уже знакомы, для имён переменных он равен длине переменной в байтах, а для меток равен **`near`**`=-1` и **`far`**`=-2` для близкой и дальней (в другом модуле)[1] 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 имён числовых констант и числовых 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; Value=100
M     equ   N+1; Value=101
      K=13;      Value=13
P     equ   K;   Value=13
Data  segment;   Value=Data
```

Рассмотрим теперь пример маленького, синтаксически правильного, но, вообще говоря, бессмысленного программного модуля на Ассемблере, для этого модуля мы построим таблицу имён пользователя:

```
Data  segment
A     dw    19
B     db    ?
      public B
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 пользователя, которая построится синтаксическим анализатором при просмотре показанной выше программы до предложения

```
      mov   ax,Data
```

включительно. Атрибуты, которые не могут быть у данного имени, отмечены прочерком.

Как мы уже знаем, значения `Value` некоторых имён остаются неизвестнымы на этапе компиляции, они будут определяться позже во время редактирования внешних связей и загрузки программы на выполнение. Такие значения мы обозначили в нашей таблице как `i16=?`. Для каждого заносимого в таблицу имени, кроме атрибутов, ещё запоминается и место в предложении, на котором встретилось это имя (т.е. отмечается, что имя уже описано или объявлено, или же только использовано до своего описания иди объявления).

---

[1] Здесь есть небольшая тонкость. Хотя передать управление на команду, помеченную, например, меткой `L` в *другом сегменте* этого же модуля можно только по команде дальнего перехода **`jmp far ptr`** `L`, однако тип этой метки в данном модуле будет равен **`near`**`=-1` (т.е. **`type`** `L=`**`near`**).

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, (ниже приведена строка таблицы для этого имени) Это имя ещё не описано и про него 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, вообще говоря, может иметь любой из допустимых для этой команды форматов `r16,m16` или `i16`. На этом примере ярко видна необходимость *второго просмотра* текста программы: только на втором просмотре, после получения информации об атрибутах имени R, возможно определить правильный формат этой команды (а значит, в частности, определить и *длину* этой команды в байтах).

После полного просмотра программы синтаксический анализатор построит таблицу имён, показанную на рис. 13.2.[1]

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 всего модуля.

На этапе синтаксического анализа выявляются все синтаксические ошибки в программе, например:

```
mov   ax,bl; Несоответствие типов
mov   ax,L; 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 не является внешним (**extrn**), но встречается только в поле операндов и ни разу не встречается в поле метки.
2. Дважды описанное имя. Одно и то же имя дважды (или большее число раз) встретилось в поле метки.[2]

---

[1] Для простоты изложения мы все имена поместили в одну таблицу. В то же время имена, являющиеся входными точками (**public**), а также внешние имена модуля (**extrn**), либо должны иметь соответствующие дополнительные атрибуты, либо помещаться в другую таблицу *внешних и входных* имён. Информация о таких именах, как мы знаем, помещается компилятором с Ассемблера в паспорт объектного модуля.

[2] Исключением из этого правила, как мы знаем, являются имена сегментов и имена процедур, которые должны встретиться в поле метки соответствующих директив ровно два раза, а также имена макроопределений

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 и отсутствует в параметрах директив **`public`** (видимо, программист что-то упустил).

Кроме того, синтаксический анализатор может обнаружить и ошибки, относящиеся к модулю в целом, например, превышение максимальной длины некоторого сегмента, отсутствие сегмента стека в *головном* модуле, использование метки команды (т.е. имени с двоеточием) в предложении резервирования памяти и т.д.

Итак, на втором проходе синтаксический анализатор обнаруживает все ошибки в программном модуле и однозначно определить формат каждой команды.[1] Теперь можно приступать 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.

(макроопределение, описанное позже, как мы знаем, *переопределяет* одноимённое макроопределение, описанное ранее).

[1] Замечание для продвинутых студентов. Вспомним, что при написании макроопределений мы использовали средства условной генерации для контроля типов переданных параметров и выдавали необходимую диагностику об ошибках использования макрокоманды. Этим мы, по существу, дополняли средства синтаксического анализатора Ассемблера. Таким образом, обычный пользователь-программист, используя макросредства, по-существу способен расширить возможности служебной программы – компилятора.

Глава 14. Понятие о мультипрограммном режиме работы

Одним из принципов Фон Неймана, как мы знаем, является принцип последовательного выполнения команд программы. Более того, архитектура машин Фон Неймана предполагает, что последовательно выполняются не только команды текущей программы, но также и сами эти программы. Другими словами, пока одна программа полностью не заканчивается, следующая программа не загружается в память и не начинает выполняться (вспомним нашу учебную трёхадресную ЭВМ). Такой режим счёта программ называется *пакетным* режимом работы ЭВМ. Это название подразумевает, что подлежащие счёту программы собираются в некоторый "пакет" (для ЭВМ первых поколений это был деревянный или металлический ящик, наполненный программами, каждая программа была в отдельной пачке перфокарт, скреплённой резинками). Разумеется, осознание того, что это особый режим появилось только после того, как стали использоваться и другие режимы работы ЭВМ, а до этого программисты и не подозревали, что они считают свои задачи в пакетном режиме ☺.

Сейчас мы познакомимся с новым весьма сложным понятием – мультипрограммным (иногда говорят, многопрограммным) режимом работы ЭВМ. Мультипрограммный режим работы означает, что в оперативной памяти компьютера одновременно находятся несколько независимых друг от друга и готовых к счёту программ пользователей.[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, а для чего вообще может потребоваться, чтобы в памяти одновременно находилось несколько программ пользователей. 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 центральный процессор производит 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, подключённых к одной ЭВМ, отлаживают свои программы, см. рис. 14.1).

Правда, здесь имеет место уже упомянутая ранее трудность: так как центральный процессор на компьютере чаще всего только один, то в каждый момент времени может выполняться или программа Васи, или программа Пети (ну, или служебная программа операционной системы при обработке прерывания). Эта трудность преодолевается введением 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), служебная процедура-диспетчер переключает центральный процессор с одной задачи *пользователя*

---

[1] Эти программы, вообще говоря, могут присутствовать в оперативной памяти не целиком. Во-первых, они могут использовать знакомую нам схему динамической загрузки, и, во-вторых, работать на так называемой виртуальной памяти, при этом некоторые части программы могут временно отсутствовать в оперативной памяти, находясь в так называемом файле подкачки (swap file) на дисковой памяти.

Звметим также, что при мультипрограммном режиме работы в памяти ЭВМ одновременно могут находиться не только программы *разных* пользователей, но и несколько *независимых* программ одного пользователя. Независимость программ означает, что они не обмениваются автоматически между собой данными в процессе счёта.

[2] На ЭВМ первых поколений "обычным" 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.

на другую по истечении определённого кванта времени (обычно порядка нескольких единиц или десятков миллисекунд). В таком режиме разделения времени (в русскоязычной литературе этот режим иногда метко называли *коммунальным* использованием ЭВМ) и у Васи, и у Пети создаётся иллюзия, что только его программа всё время считается на компьютере (правда, почему-то медленно ☺).

Рис. 14.1. Одновременное нахождение в памяти
нескольких программ пользователей.

Если отвлечься от несколько шутливого примера с Васей и Петей, то можно заметить, что потребность в таком псевдо-одновременном 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 предназначены для связи центральной части машины с "внешним миром", и 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 лазерный принтер напечатает на бумаге всего 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.

Из этих соображений, очевидно, что в то время, когда по запросу некоторой программы производится обмен данными с медленными внешними устройствами, центральный процессор, как правило, не сможет выполнять команды этой программы, т.е. будет простаивать. Например, рассмотрим случай, когда в программе Васи, написанной на Паскале, 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 программы-диспетчера переключаться на выполнение других программ пользователей, тоже расположенных в оперативной памяти. Теперь, пока одна программа пользователя 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 отно-

сительно *независимо* от центрального процессора. Действительно, вспомните, что в машине Фон Неймана всеми операциями с внешними устройствами управлял именно центральный процессор, посылая им особые управляющие сигналы, которые мы изображали на схеме одинарными стрелками, и, естественно, занимаясь этой работой, у центрального процессора уже не было возможности ещё и выполнять другие команды программы.

Как мы уже 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 является совершенно *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 сильно повышает производительность компьютера и 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. Особо подчеркнём, что это требования именно к аппаратуре ЭВМ, а не к программному обеспечению.

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 она 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 предумышленно обратиться в память другой программы (по записи или даже по чтению данных). Очевидно, что без такого механизма мультипрограммный режим просто невозможен.[1]

Механизм защиты оперативной памяти на современных ЭВМ устроен весьма сложно, и часто связан с механизмом так называемой виртуальной памяти, который изучается в курсе, посвящённом операционным системам.[2] 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 (или адрес начала и конца любого сегмента при сегментной организации 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

$A_{нач} := 200000_{10}$ и $A_{кон} := 500000_{10}$

---

[1] Даже если не принимать во внимание "вредных" программистов, которые специально захотят испортить или незаконно прочитать данные других программ, всегда существуют вероятность таких действий из-за семантических ошибок в программах даже у "добропорядочных" программистов. Незаконное обращение к чужым ресурсам (в частности, к чужой оперативной памяти) "по-научному" называется несанкционированным доступом.

[2] На факультете Вычислительной математики и кибернетики МГУ этот курс называется "Системное программное обеспечение", он читается в третьем семестре.

Далее, в центральный процессор добавлена способность перед *каждым* обращением в оперативную память по физическому адресу $A_{физ}$ автоматически проверять условие

$$A_{нач} \le A_{физ} \le A_{кон}$$

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 центральный процессор вырабатывает 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 это ограничение несущественно, так как на них реализован уже упоминавшейся 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. С другой стороны, если реализован механизм виртуальной памяти, то на его базе легко сделать и другой, более совершенный механизм защиты памяти.[1]

14.1.3. Аппарат привилегированных команд

Сейчас мы рассмотрим ещё одно *необходимое* свойство аппаратуры, без которого невозможно реализовать мультипрограммный режим работы ЭВМ. Это свойство иногда называется аппаратом привилегированных команд, а иногда – защищённым режимом работы центрального процессора, и 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.

Далее, в центральном процессоре располагается специальный одноразрядный регистр *режима работы*, который может, естественно, принимать только два значения: `0` и `1`. Значение этого регистра и определяют тот *режим*, в котором в данный момент работает центральный процессор: обычный режим (или режим пользователя) или привилегированный режим.[2] 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 центральным процессором 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 для рассмотренного выше механизма защиты памяти заносят на регистры защиты $A_{нач}$ 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. Ясно, что при этом и механизм защиты памяти становится совершенно бесполезным.

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 (файлы), принадлежащие другим программам. То же самое относится и к печа-

---

[1] На факультете Вычислительной математики и кибернетики МГУ это изучается в курсе под названием "Системное программное обеспечение", он читается в третьем семестре.

[2] В архитектуре нашего компьютера регистр режима работы содержит два разряда и может принимать значения `0`,`1`,`2` и `3`. Практически всегда, однако, для указанных выше целей используются только два из этих четырёх значений (`0` и `3`).

тающему устройству: если разрешить всем программам бесконтрольно выводить свои данные на печать, то, конечно, разобраться в том, что же получится на бумаге, будет чаще всего невозможно.[1]

Итак, в мультипрограммном режиме программе пользователя запрещается выполнять многие "опасные" команды, в частности команды, работающие с внешними устройствами (дисками, принтерами, линиями связи и т.д.). Как же тогда быть, если программе необходимо, например, считать данные из *своего* файла на диске в оперативную память? Выход один – программа пользователя должна обратиться к определённым служебным процедурам, с просьбой выполнить для неё ту работу, которую сама программа пользователя сделать не в состоянии. Эти служебные процедуры, естественно, должны работать в привилегированном режиме. Перед выполнением запроса из программы пользователя, такая служебная процедура проверяет, имеет ли эта программа 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 некоторой (не привилегированной) машинной команде. Значительно сложнее обстоит дело с такой опасной операцией, как переключение центрального процессора из обычного в привилегированный режим работы. Это переключение невозможно выполнить по какой-либо машинной команде (докажите, что это так!). Обычно переключение в привилегированный режим производится автоматически при обработке центральным процессором сигнала прерывания, в этом случае процедура-обработчик прерывания уже начинает свою работу в привилегированном режиме. Иногда переключение в привилегированный режим производится центральным процессором 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 тоже 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 – зациклиться), и ничто не могло бы вывести компьютер из этого состояния.[3]

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, большая ёмкость дисков 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 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.

---

[1] Например, если работающие в мультипрограммном режиме программы Васи и Пети производят вывод на единственный общий принтер, то на самом деле данные, которые печатает каждая программа пользователя, не выводятся сразу на печать, а записываются в специальный файл, который будет выводиться на печать только после полного завершения этой программы.

[2] Способы задания таких полномочий студенты факультета Вычислительной математики и кибернетики МГУ изучатют в третьем семестре на примере операционной системы Unix.

[3] Обычно при счёте в мультипрограммном режиме программа пользователя может сообщить операционной системе своё максимальное время счёта. Это не физическое время, а сумма всех квантов времени центрального процессора, выделяемых для этой задачи. Можно сказать, что программа заводит для себя "будильник", на котором выставляется время окончания её работы. По истечению этого максимального времени счёта программа пользователя получит соответствующий сигнал и может быть завершена.

Глава 15. Архитектурные особенности современных ЭВМ

Для лучшего понимания тех важных архитектурных особенностей, которыми обладают современные компьютеры, исследуем сейчас следующий вопрос: оценим скорость работы различных устройств ЭВМ. Оперативная память современных ЭВМ массовог производства способна считывать и записывать данные примерно каждые 10 наносекунд (нс., 1нс = 10^{-9} сек.), а центральный процессор может выполнить машинную команду примерно за 1–2нс. После некоторого размышления становится понятным, что "здесь что-то не так".

Действительно, рассмотрим, например, машинную команду **add ax,X**. Для выполнения этой команды центральный процессор должен сначала считать из оперативной памяти саму команду (в нашей архитектуре это 4 байта), затем операнд X (это ещё 2 байта), потом произвести операцию сложения. Таким образом, центральный процессор потратит на выполнение этой команды 6байт*10нс+2нс=62нс, причём собственно центральный процессор будет работать только 2нс и 60нс будет ждать, пока команды и числа не поступят из оперативной памяти. Спрашивается, зачем делать центральный процессор таким быстрым, если всё равно 97% своего времени он будет ждать, пока будет производиться обмен командами и данными между оперативной памятью и его регистрами? Налицо явное *несоответствие* в скорости работы оперативной памяти и центрального процессора ЭВМ.

Первое, что приходит в голову для преодоления этого несоответствия, это увеличить в 10 раз скорость работы оперативной памяти. Технически это вполне возможно, однако при этом стоимость компьютера также возрастёт в несколько раз, что, конечно же, недопустимо, так как такие дорогие ЭВМ не будут покупать. Кроме того, как мы вскоре увидим, одно лишь увеличение в 10 раз скорости работы оперативной памяти не решит всех проблем.

На современных ЭВМ данная проблема несоответствия скорости работы памяти и центрального процессора решается несколькими способами, которые мы сейчас и рассмотрим. Сначала, выяснив, что главным тормозом в работе является оперативная память, эту память стали делать таким образом, чтобы за одно обращение к ней она выдавала не по одному байту, а сразу по несколько байт с последовательными адресами. Для этого оперативную память разбивают на блоки (обычно называемые *банками* памяти), причём эти банки памяти могут работать параллельно. Этот приём называют *расслоением памяти*. Например, если память разбита на 8 банков, то за одно обращение к ней можно сразу считать 8 байт, при этом байты с последовательными адресами располагается в разных банках памяти. Например, байт с адресом 1000 располагается в банке памяти с номером 0, байт с адресом 1001 – в банке памяти с номером 1 и т.д. Таким образом, за одно обращение к памяти можно считать несколько команд или данных, расположенных в памяти подряд (т.е. в байтах с последовательными адресами). Поэтому понятно, почему оперативную память современных ЭВМ разбивают на 8,16 и даже 32 банка.

Скорость работы оперативной памяти современных ЭВМ так велика, что требуется какое-то образное сравнение, чтобы это почувствовать. Легко подсчитать, что за одну секунду из оперативной памяти, разбитой на 8 банков, можно прочитать $8*10^8$ байт. Если считать каждый байт символом печатного текста и учесть, что на стандартной странице книги помещается примерно 2000 символов, то получается, что за 1 секунду центральный процессор может прочитать в свою оперативную память целую библиотеку из 80 томов по 500 страниц в каждом томе. Очень впечатляюще, не правда ли?

Легко, однако, вычислить, что, несмотря на такую огромную скорость, даже такая, обладающая расслоением по банкам, оперативная память продолжает тормозить работу центрального процессора. Проведя заново расчёт времени выполнения нашей предыдущей команды **add ax,X** мы получим:

10нс (чтение команды) + 10нс (чтение числа X) + 2нс (выполнение команды) = 22нс.

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 современных компьютеров встраивается специальная память, которую называют *памятью типа кэш*, или просто кэшем.[1]

---

[1] По-английски слово кэш (cash) чаще всего обозначает наличные деньги "в кошельке", в противовес безналичным деньгам, хранящимся, например, в банке. Как мы вскоре увидим, это название весьма точно отражает суть памяти типа кэш и её взаимодействие с расслоённой на банки оперативной памятью.

Кэш делается на очень быстрых интегральных схемах *статической* памяти и работает почти с такой же скоростью, как и центральный процессор, т.е. может, например, выдавать по 8 байт каждые 2-3нс. 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 данных.[1] Такого рода память, в которой команды и числа не имеют адресов, как мы уже не упоминали, называется ассоциативной памятью.[2] Центральный процессор работает с кэшем по следующей схеме.

Когда центральному процессору нужна какая-то команда или данное, то сначала он смотрит, не находится ли уже 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 при этом в кэше. Заметим, что используя расслоение оперативной памяти, вместе с требуемыми байтами памяти в кэш одновременно попадают и несколько соседних с ними байтов памяти (обычно этих байт даже больше, чем банков памяти, это называется строкой кэш памяти – cash line), что как мы далее увидим, является очень полезным для увеличения быстродействия ЭВМ.

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 из центрального процессора.[3] При этом, если эти данные в кэш памяти изменялись, то они переписываются в оперативную память.[4]

Таким образом, в кэше накапливаются наиболее часто используемые команды и данные выполняемой программы, например, все команды не очень длинных циклов после их первого выполнения будут находиться в памяти типа кэш.[5] Часто говорят, что кэш образует *буфер* между быстрой регистровой памятью центрального процессора и медленной оперативной памятью (буфером обычно называется устройство, сглаживающее неравномерность скорости работы других взаимодействующих между собой устройств). На рис. 15.1 показана схема взаимодействия центрального процессора и оперативной памяти с использованием кэша.

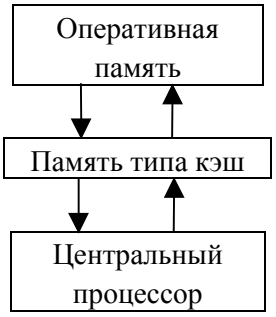


Рис. 15.1. Схема обмена с использованием памяти типа кэш.

Как уже говорилось, память типа кэш строится из очень быстрых и, следовательно, дорогих интегральных схем, поэтому из экономических соображений её объём сравнительно невелик, примерно

---

[1] Конечно, существуют *привилегированные* команды для работы с кэшем как с единым целым, это, например, команда очистки кэша от всех находящихся там команд и данных.

[2] Скорее всего, так же устроена память и у человека. Действительно, было бы удивительно, если бы у того, что мы помним, были бы ещё какие-то адреса.

[3] Проще всего это сделать, если вести *очередь* обращений к данным в кэше, при этом при каждом чтении или записи некоторого данного, оно ставится в начало этой очереди. Легко понять, что при этом в конце очереди автоматически соберуться те данные, к которым дольше всего не было обращения.

[4] В архитектуре некоторых ЭВМ может быть два отдельных кэша, один для команд и другой для данных. Такие ЭВМ нарушают принцип Фон Неймана неразличимости команд и данных. При этом команды запрещается менять, поэтому в кэш команд никогда не производится запись, что позволяет упростить реализацию этого кэша.

[5] Как видно из этого алгоритма работы кэша, он весьма "болезненно" реагирует на прерывания, так как при этом производится переключение на другую программу, и информацию в кэше необходимо по большей части сменить.

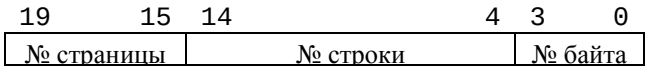
5% от объёма оперативной памяти. Однако, несмотря на свой относительно малый объём, кэш вызывает значительное увеличение скорости работы ЭВМ, так как по статистике примерно 90-95% всех обращений из центрального процессора за командами и данными производится именно в память типа кэш.

На персональной ЭВМ для изучения влияния памяти типа кэш на производительность компьютера можно произвести такой, как говорят, вычислительный эксперимент. Сначала замеряется усреднённая производительность ЭВМ с помощью какой-нибудь предназначенной для этого программы (например, программой SysInfo из пакета Norton Utilities). После этого при перезагрузке машины память типа кэш отключается в настройках BIOS'a,[1] и снова замеряется производительность компьютера, которая при этом снижается примерно на порядок. Главное, после этого эксперимента не забыть опять включить на своём компьютере память типа кэш ☺.

Рассмотрим одну из простейших реализаций памяти типа кэш, так называемый кэш *прямого отображения* (смысл этого названия вскоре будет ясен из алгоритма его работы). Пусть размер оперативной памяти составляет 1 Мбайт (2^{20} байт), и выберем объём кэш памяти равным 1/32 от объёма оперативной памяти, это 32 Кбайта (2^{15} байт), что составляет около 3% от объёма оперативной памяти. Далее, разобьём всю оперативную память на участки размером по 32 Кбайта, будем называть такие участки *страницами*, как видим, в нашем случае размер страницы совпадает с размером самого кэша. Каждую страницу, как находящуюся в оперативной памяти, так и страницу кэша, в свою очередь, будем рассматривать как состоящую из *строк* длиной по 16 байт. Таким образом, имеем

$$\text{Вся память } 2^{20} = 2^5 \text{ страниц x } 2^{11} \text{ строк x } 2^4 \text{ байт.}$$

Тогда 20-разрядный физический адрес любого байта оперативной памяти тоже можно разбить на три поля: адрес страницы (5 бит), адрес строки в странице (11 бит) и адрес байта в строке (4 бита):



Теперь каждую строку в кэш памяти снабдим шестью дополнительными битами, пять из которых будут хранить номер некоторой страницы оперативной памяти, а в шестом бите будет содержаться признак изменения строки в кэш памяти (0 – в строку кэш памяти не было записи, 1 – была запись). Объём этой дополнительной (служебной) памяти составляет всего около 6% (6 бит/16 байт), что незначительно увеличивает сложность кэша.

Алгоритм работы центрального процессора при наличие такой кэш памяти будет заключаться в следующем. Сначала из физического адреса байта, по которому необходимо обратиться в оперативную память, выделяется номер строки (разряды с 4 по 14), и проверяется строка в кэш памяти с этим же номером. Если номер стриницы, приписанный данной строке в кэше, совпадает с номером той страницы, к которой мы хотим обратиться, то нужный нам байт уже находится именно в этой строке кэша, и обращаться в оперативную память не надо. В противном случае строку с данным номером в кэш памяти необходимо заменить на строку с этим же номером из нужной нам страницы оперативной памяти. Разумеется, при такой замене сначала проверяется бит-признак изменения этой строки в кэш памяти, если строка менялась, то её, конечно, надо записать на своё место в оперативной памяти.

Таким образом, получается, что, если некоторая строка оперативной памяти присутствует в кэше, то она находится в нём на том же месте, что и в своей странице оперативной памяти. Именно поэтому так организованный кэш и называется кэшем прямого отображения (строк из страниц оперативной памяти на строки кэш памяти). Заметим, что если нужного нам байта в кэше не оказалось (как говорится, случился *промах* кэша), то из оперативной памяти считывается в кэш сразу вся строка (16 байт), это занимает всего два обращения к оперативной памяти при её восьмикратном расслоении (и два дополнительных обращения, если в строку была запись).

Описанный выше кэш прямого отображения быстро работает и его просто реализовать, но он имеет и существенный недостаток. Заметим, что, если попеременно обращаться к байтам с одинаковыми номерами в двух страницах оперативной памяти, то строки из этих страниц будут всё время сменять друг друга в кэш памяти, что, конечно, весьма негативно скажится на скорости

[1] К сожалению, на компьютерах новых моделей возможность отключать кэш из BIOS'a обычно не предусмотрена, как совершенно бесполезная в практической работе пользователя. Кэш память можно отключить только с помощью специальных привилегированных команд.

выполнения программы. Современные компьютеры снабжаются более сложно устроенной кэш памятью, в которой вероятность описанной выше неприятной ситуации значительно ниже.

Для ещё большего увеличения скорости чтения и записи команд и данных центральным процессором можно включить в архитектуру не один, а два и даже три последовательно подключённых кэша. При этом самый внутрений (ближайший к центральному процессору) кэш называется кэшем первого уровня, следующиё – второго уровня, и т.д.

Здесь, однако, необходимо уяснить для себя следующее. Успешное применение памяти типа кэш базируется на свойстве *локальности* программ, это свойство уже упоминалось нами при изучении близких относительных переходов. Свойство локальности заключается в том, что выполняемые команды и обрабатываемые данные программы не разбросаны по памяти в хаотическом беспорядке, а обнаруживают тенденцию группироваться в некоторые относительно небольшие области. Это, например, команды в теле циклов, данные внутри массивов и т.д. Например, можно специально написать программу, не обладающую свойством локальности, такая программа будет после каждой команды случайным образом переходить на выполнение следующей команды в любой части программы, а обрабатываемые данные также выбирать из областей памяти со случайными адресами. Так вот, Вам необходимо понять, что при выполнении такой программы кэш будет бесполезен.[1]

Вот теперь, на машине с памятью типа кэш, наша рассмотренная выше команда `add ax,X` будет чаще всего выполняться уже за 2(команда)+2(данные)+2(сложение)=6нс.[2] Как видим, ситуация коренным образом улучшилась, хотя всё равно получается, что сам центральный процессор работает только 30% от времени выполнения команды, а остальное время ожидает поступления на свои регистры команд и данных. Для того чтобы исправить эту неприятную 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 либо иметь несколько центральных процессоров, либо центральный процессор такого компьютера строится по так называемой *конвейерной* (pipeline) архитектуре.[3] Рассмотрим схему работы таких конвейерных ЭВМ.

Выполнение каждой команды любым центральным процессором, как мы уже знаем, состоит на нескольких этапов или шагов. Можно, например, выделить следующие основные шаги выполнения команды.

1. Выбор команды из оперативной памяти (или кэша) на регистр команд.
2. Определение кода операции (так называемое декодирование команды).
3. Вычисление исполнительных адресов операндов в памяти.
4. Выбор операндов из оперативной памяти (или кэша) на регистры арифметико-логического устройства.
5. Выполнение требуемой операции (сложение, умножение, сдвиг и т.д.) над операндами на регистрах арифметико-логического устройства.
6. 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. Эти блоки стараются строить так, чтобы все они выполняли свою работу по выполнению шага команды за одно и то же время. Теперь понятно, что такие блоки можно заставить работать параллельно, обеспечивая, таким образом, одновременное выполнение центральным процессором нескольких последовательных команд программы. На рис. 15.2 приведена схема работы центрального процессора конвейерной ЭВМ, направление движения команд на конвейере показано толстой стрелкой. Так как выполнение каждой команды мы разби-

---

[1] Замечание для программистов-математиков: объём сегментов команд и сегментов данных такой "нелокальной" программы должен существенно превышать размер памяти типа кэш.

[2] Заметим, что за одно обращение из памяти теперь читается не одна команда, а в среднем две-три последовательные команды программы, поэтому можно считать, что среднее время выполнения команды будет ещё меньше: 1+2+2=5нс.

[3] На персональных ЭВМ впервые конвейер реализован на процессоре Intel 486 в 1989 году, но на больших компьютерах это произошло раньше. Например, конвейер имела отечественная ЭВМ БЭСМ-6, выпускавшаяся в 70-х годах прошлого века.

ли на шесть шагов, то одновременно на нашем конвейере может находиться шесть команд программы.[1]

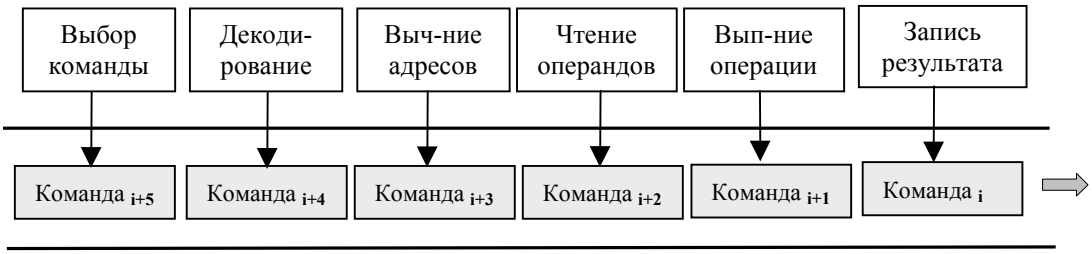


Рис. 15.2. Схема работы конвейера.

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, что теперь в центральном процессоре уже нет привычного нам *регистра команд*, на котором располагается текущая выполняемая команда. Вместо этого поток выполняемых команд располагается в специальном *буфере команд*.

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.

Вот теперь, после введения в архитектуру ЭВМ конвейера, мы, наконец, 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 `1нс`, тогда каждая команда выполняется на конвейере за `6нс`, и за это время она успевает произвести все необходимые обмены и командами и данными с памятью (чаще всего из кэша). В то же время, как мы уже отмечали, скорость выполнения *потока команд* центральным процессором получается в 6 раз больше за счёт работы конвейера. Поэтому и говорят, что, например, компьютер работает со скоростью 10^9 операций в секунду (то есть как бы выполняет по одной команде каждую наносекунду), хотя мы теперь знаем, что на самом деле выполнение каждой команды занимает в несколько раз больше времени.

Разумеется, как всегда 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 достаточно 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.[2] Таким образом, вторая команды будет задержана на третьей позиции конвейера (на четвёртой позиции уже надо читать операнд из регистра `al`, а этот операнд ещё не помещён в этот регистр). Вместе со второй командой из нашего примера остановится, и выполнение следующих за ней команд и на конвейере образуются два "пустых места". В таблице 15.1 показана схема движения команд из приведённого выше фрагмента программы на конвейере, каждая строка соответствует одному шагу работы конвейера.

[1] Конвейеры современных ЭВМ могут разбивать выполнения команд и на большее число шагов, например, в компьютере Pentium-IV конвейер состоит из 20 шагов.

[2] Такая зависимость между данными называется в научной литературе RAW (Read after Write – чтение после записи, т.е. мы читаем ещё не записанный операнд). Путём рассуждений можно понять, что существуют и две другие зависимости между данными: WAR (Write after Read – запись после чтения) и WAR (Write after Write – запись после записи).

Таблица 15.1. Движение команд по конвейеру.

Выборка	Декодирование	Вычисление адресов	Чтение операндов	Выполнение операции	Запись Результата
add al,[bx]					
sub X,al	**add** al,[bx]				
inc bx	**sub** X,al	**Add** al,[bx]			
inc di	**inc** bx	**Sub** X,al	**add** al,[bx]		
	inc bx	**Sub** X,al		**add** al,[bx]	
	inc bx	**Sub** X,al			**add** al,[bx]
	inc di	**Inc** bx	**sub** X,al		
		Inc di	**inc** bx	**sub** X,al	
			inc di	**inc** bx	**sub** X,al
				inc di	**inc** bx
					inc di

Ясно, что скорость выполнения всей программы с такими, как говорят, *зависимостями по данным*, может при этом сильно упасть. Зная такую особенность работы конвейера, "умный" центрального процессора, исправляя оплошность программиста, может изменить порядок команд в машинной программе, получив, например, такой *эквивалентный* фрагмент: [1]

```
add  al,[bx]
inc  bx
inc  di
sub  X,al
```

При выполнении этого фрагмента, как легко увидеть, ячейки конвейера уже не будут пустовать.

Упражнение. Нарисуйте новую схему работы конвейера для этого оптимизированного фрагмента программы и убедитесь, что нам не будет пустых мест.

Другая неприятность в работе конвейера случается, когда на него поступает команда *условного перехода*. Что надо делать после выполнения этой команды, производить переход в другое место программы, или же продолжить последовательное выполнение команд, выяснится только тогда, когда команда условного перехода сойдёт с конвейера. Так, спрашивается, из какой же ветви условного перехода выбирать на конвейер следующую команду? Обычно при конструировании конвейера для обработки команд условного перехода принимается какое-либо одно из следующих двух решений.

Во-первых, можно выбирать команды из наиболее *вероятной* ветви условного оператора. Например, очевидно, что в нашем компьютере для команды цикла **loop** переход на повторение тела цикла значительно более вероятен, чем выход из цикла вниз. Для остальных команд условного перехода центральный процессор может накапливать *статистику*, насколько часто та или иная команда осуществляет переход, а не естественное продолжение программы. Этот метод называется в научной литературе *предсказанием ветвления*. Вероятность такого предсказания у современных конвейерных ЭВИ может приближаться к 90%.

Во-вторых, можно поочерёдно выбирать на конвейер команды из *обеих* ветвей условного перехода. Разумеется, в этом случае половина команд будет выполняться зря и их потом "недоделанными" придётся выбросить с конвейера. В литературе этот метод называется *спекулятивным выполнением* (режущее для российского уха название связано с неудачным выбором значения соответствующего английского термина speculative: лучше переводить его не как *спекулятивный*, а как *умозрительный*, имея в виду умозрительное, а не настоящее движение алгоритма сразу по двум ветвям условного перехода, что противоречит сути вычислительного процесса).

[1] Так как переставлять команды в программе для её более эффективного выполнения умеют только достаточно мощные центральные процессоры, то можно поручить эту работу и компилятору. Такие компиляторы называются *оптимизирующими* компиляторами, оптимизация машинного кода программы выполняется обычно на одном или нескольких дополнительных проходах компилятора перед получением объектного модуля. Необходимо, однако, заметить, что оптимизирующие компиляторы обычно реализуются для языков *высокого* уровня и не занимаются оптимизацией собственно машинного кода. В то же время современные процессоры (например, семейства Pentium) могут сами при выполнении программы (динамически) переставлять её команды для повышения производительности работы 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 полностью очищать от частично выполненных команд предыдущей программы.[1]

Ещё одна неприятность, связанная с функционированием конвейера, подстерегает нас, когда в программе возникает аварийная ситуация (в русскоязычной литературе сокращённо АВОСТ). Это может быть деление на ноль, выполнения привилегированной команда в режиме пользователя, попытка нарушения защиты памяти, неверный код операции и т.д. Заметим, что теперь обработчик прерывания аварийной ситуации часто не сможет *однозначно* определить то место (конкретную команду) в программе пользователя, где возникла эта ситуация.

Действительно, аварийную ситуацию может вызвать любая команда, находящаяся на конвейере, это, естественно, вызывает трудности при отладке программ (появляются, как их называют в специальной литературе, "неопределённые" прерывания, т.е. прерывания с неопределённым местом их возникновения в программе). А при упоминавшимся выше спекулятивном способе выполнения на конвейере условных переходов вообще возможны "ложные" прерывания в той ветви условного перехода, которая на самом деле *не должна* выполняться. Здесь приходится идти на беспрецендентный шаг и *откладывать* обработку такого "сомнительного" прерывания до того момента, пока команда условного перехода не сойдёт с конвейера, и можно будет определить, произошло ли это прерывание "на самом деле".

Ситуация ещё более запутывается, если в компьютере есть несколько конвейеров, например, один конвейер выполняет команды плавающей арифметики, а другой – все остальные команды. В этом случае, например, команда, стоящая в некотором фрагменте программы второй, может завершиться раньше, чем команда, стоящая первой. Спрашивается, с какой команды возобновить выполнение программы после возврата из прерывания? Вот и приходится в нужные моменты искусственно *тормозить* работу одного из конвейеров, чтобы не нарушался естественный порядок следования команд в программе. Как видим, конструкторам центральных процессоров современных ЭВМ не позавидуешь ☺.

На этом мы закончим наше краткое знакомство с архитектурными особенностями современных ЭВМ, ориентированными на повышение производительности их работы, и перейдём к сравнению между собой ЭВМ разных классов.

15.2. ЭВМ различной архитектуры

Отметим сейчас одно важное обстоятельство. До сих пор мы изучали, в основном, архитектуру центральной части компьютера, т.е. центрального процессора и оперативной памяти. При этом практически не рассматривалась архитектура ЭВМ в целом, то есть способы взаимодействия центральной части компьютера с периферийными устройствами, а также способы управления устройствами ввода/вывода со стороны центрального процессора. Такое "однобокое" изучение архитектуры ЭВМ имело свою причину. Дело в том, что, несмотря на большое разнообразие архитектур центральных процессоров современных ЭВМ, различие в этих архитектурах всё же значительно меньше, чем в архитектурах компьютеров в целом. Теперь же нам пора обратить внимание на связь центральной части ЭВМ и её периферийных устройств.

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 архитек-

[1] На супер-ЭВМ для целей более эффективного использования конвейера 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 в нашем курсе.

туры этого класса ЭВМ была, как можно легко догадаться из названия, *общая шина*.[1] 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 составляет несколько десятков. Все устройства компьютера в архитектуре с общей шиной соединяются между собой посредством подключения к такому общему для них набору электрических проводов – шине. На рис. 15.3 показана схема соединения всех устройств компьютера между собой с помощью такой общей шины.

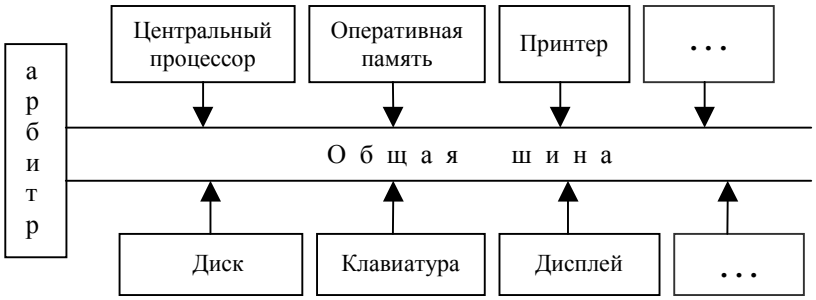


Рис. 15.3. Архитектура компьютера с общей шиной.

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 компьютера связаны с общей шиной посредством специальных электронных схем, которые чаще всего называются *портами* ввода/вывода. Каждый такой порт имеет на шине уникальный номер (в нашей архитектуре этот номер имеет формат `i16`). Обычно каждому устройства компьютера приписан 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 общей шины многими устройствами могут возникать конфликты, когда два или более устройств захотят одновременно обмениваться между собой данными. Для разрешения таких конфликтов предназначен *арбитр шины* – специальная электронная схема, которая обычно располагается на одном из концов этой шины. Разрешение конфликтов производится по принципу приоритетов устройств: при конфликте арбитром отдаётся предпочтение устройству с большим приоритетом. В простейшем случае приоритеты устройствам явно не назначаются, а 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. Не правда ли, это несколько похоже на известный солдатский принцип "держись подальше от начальства и поближе к кухне" (неясно только, почему арбитр шины – это не начальник, как можно было бы ожидать, а кухня ☺).

Разберём теперь 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 (для этого ведущее устройство может прочитать из специального служебного 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. Это означает, что для остальных устройств шина теперь будет выдавать признак занятости.

После захвата шины ведущее устройство определяет, готово ли ведомое устройство для обмена данными. Для этого ведущее устройство посылает ведомому устройству специальный сигнал и ждёт ответа, или же читает из порта ведомого устройства его *флаг готовности.* Определив готовность

[1] В некоторых книгах, посвящённых архитектуре ЭВМ, такую шину часто называют также *внешней* шиной. Это название призвано подчеркнуть, что в архитектуре таких компьютеров, наряду с этой внешней шиной, существуют и другие (внутренние) шины.

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, выведя весь компьютер из строя). Все описанные выше операции с общей шиной (запрос, захват и т.д.) производятся по строгим правилам, эти правила обычно называются *протоколом* работы с общей шиной. Обычно все действия, связанные с обменом по шине одной порцией данных, называются *циклом* [работы] шины (говорят о циклах чтения и записи в память, цикле передачи сигнала прерывания и т.д.).

Такова в простейшем изложении схема обмена данными по общей шине. Рассмотрим теперь, как видит общую шину нашего компьютера программист на Ассемблере. Как уже было сказано, у каждого периферийного устройства обязательно есть один или несколько портов с номерами, закреплёнными за этим устройством. Программист может обмениваться с портами байтами или словами (в зависимости от вида порта). Для записи значения регистра в некоторый порт используется машинная 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 регистром `al` (если производится запись в порт байта) или `ax` (если производится запись в порт слова).

Для чтения данных в регистр 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 регистром `al` (если производится чтение из порта байта) или `ax` (если производится чтение слова). Далее мы рассмотрим небольшой пример с использованием этих команд.

Уяснить для себя способ взаимодействия программы на Ассемблере и "внешнего мира" с помощью общей шины можно на таком образном примере. Программа ведёт своё "существование" во внутренней части ЭВМ (в оперативной памяти и центральном процессоре), и не может покидать этой своей "резиденции". Для связи с "внешним миром" у нашей программы имеется только одна возможность – это порты, которые можно рассматривать как своеобразные "почтовые ящики" на внутренней стороне двери "резиденции" программы. В некоторые из этих ящиков-портов программы может бросать свои короткие "телеграммы" для внешних устройств, из других ящиков можно доставать "телеграммы", приходящие от внешних устройств (длина каждой "телеграммы" в младших моделях нашего семейства только один или два байта). Понятно, что для того, чтобы получить достаточно большой объём данных (например, строку текста с клавиатуры), программа должна обменяться с устройством большим числом таких "телеграмм". Можно сказать, что у программы довольно скучная жизнь в её "резиденции", никаких тебе газет и журналов, и тем более радио и телевидения, для общения с "внешним миром" одна только телеграфная связь ☺.

Теперь нам будет полезно рассмотреть общую архитектуру связи центрального процессора и периферийных устройств с точки зрения пользователей разного уровня. Рассмотрим, как обстоит дело на внешнем, концептуальном, внутреннем и инженерном уровне в*и*дения архитектуры ЭВМ.

- Конечный пользователь. Пользователь-непрограммист бухгалтер Иванов твёрдо уверен, что в его компьютере есть команда "Распечатать ведомость на зарплату", так как именно это происходит каждый раз, когда он нажимает на кнопку меню "Печать ведомости". Он также знает, что в его распоряжении имеются также такие удобные команды его компьютера, как "Подведение баланса", "Проводка счёта" и другие, непонятные неосведомлённым лицам операции.
- Прикладной программист. Программист Петров, который и написал бухгалтерскую программу на языке Паскаль, только улыбнётся наивности бухгалтера Иванова. Уж он то точно знает, что даже для того, чтобы вывести только один, например, символ `'A'`, надо написать оператор стандартной процедуры `Write('A')`. Для Петрова общение его программы с внешним миром заключается в использовании стандартных процедур ввода/вывода, доступных в его

языке программирования высокого уровня. Правда, Петрову известно, что на самом деле его программа сначала переводится (транслируется) с Паскаля на машинный язык, а лишь потом выполняется на компьютере. Поэтому он из любопытства поинтересовался у программиста на Ассемблере Сидорова, что тот напишет в своей программе, чтобы вывести символ `'A'`. Сидоров, немного подумав, ответил, что обычно для этой цели он пишет предложение Ассемблера `outch 'A'`. Разница между этими двумя способами вывода символа в Паскале и в Ассемблере показалась Петрову несущественной. Например, он читал о том, что, скажем, в языке С для этой же цели надо вызвать библиотечную функцию `printf("%c",'A');`.[1]

- Программист на Ассемблере. Сидоров, однако, знает, что предложение `outch 'A'` является на самом деле не командой машины, а *макрокомандой*, на её место Макропроцессор подставит на этапе компиляции 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*, которое может быть, в частности, как экраном, так и печатающим устройством или текстовым файлом. Таким образом, общение с внешним миром представляется для Сидорова множеством системных вызовов в его программе на Ассемблере с использованием команды языка машины с кодом операции **`int`**.

- Системный программист. Системный программист Антонов (раньше иногда говорили *системный аналитик*), однако, снисходительно пояснит Сидорову, что его системный вызов – это просто переход по команде **`int`** на служебную процедуру-обработчика прерывания с номером `21h`. А уж эта процедура и произведёт *на самом деле* вывод символа, используя, в частности, специальные команды языка машины для обмена с внешними устройствами **`in`** и **`out`**.
- 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 многоступенчатой сложной последовательности действий с общей шиной. Эти действия включает в себя такие операции с общей шиной, как запрос, захват, передача данных и освобождение этой шины – цикл 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 в*и*дения архитектуры компьютера. И, как мы уже говорили, опускаться на более низкий уровень рассмотрения архитектуры следует только тогда, когда это абсолютно необходимо для дела. В этом смысле точка зрения бухгалтера Иванова для него самого ничем не хуже, чем системного программиста Антонова.

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 перемещения курсора на экране компьютера в позицию с координатами `(X,Y)`.

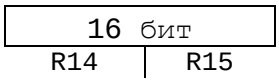
Для прикладного программиста, как Вы знаете, для этой цели надо выполнить, например, оператор стандартный процедуры Турбо-Паскаля `GotoXY(X,Y)`. Для программиста на Ассемблере позицирование курсора можно выполнить с использованием такого системного вызова:

```
mov  ah,2
mov  bl,0
mov  dl,X
mov  dh,Y
int  10h
```

[1] Это очень яркий пример того, как макросредства *повышают* уровень языка Ассемблера: макрокоманда вывода символа в языке *низкого* уровня оказывается для Петрова по внешнему виду (если отвлечься от деталей синтаксиса), очень похожа на соответствующие средства языков *высокого* уровня.

Как видим, параметры X и Y для позицирования курсора 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 этого системного вызова, который выполняет запрос на позицирование 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 на экране. 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, как это часто бывает в нашей архитектуре, может хранить длинное целое число, как показано ниже



Далее, дисплей "считает", что его экран имеет не 25 строк и 80 столбцов,[2] как думают обычные 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, как считает программист на Паскале, а от 0 до 79, во второй строке – от 80 до 159 и т.д. Другими словами, все позиции экрана рассматриваются как одномерный массив, пронумерованный, начиная с нуля. Так вот, чтобы курсор переместился в нужную нам позицию (X,Y) в пару регистров <R14,R15> 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. Мы уже знаем, что это можно сделать, например, такими командами:

```
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 ax на bx
```

Теперь необходимо переслать содержимое регистров bl и bh соответственно в регистры R15 и R14 нашего дисплея. Для этого мы будем использовать два порта дисплея (в каждый можно записывать для 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 десятичные номера 14 и 15). После выбора номера регистра назначения, запись в этот регистр производится посредством посылки байта в "транспортировочный" 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 регистры 3D5h
mov  al,bl; младший байт из bx
```

---

[1] Этот пример относится только к младшей модели нашего семейства, которые снабжались достаточно простым видеоконтроллером, управляющим выводом данных на простой VGA дисплей в компьютерах первых поколений нашего семейства. Сейчас, наверное, такие ЭВМ можно найти только в музеях вычислительной техники. Современные компьютеры оснащаются сложными видеоплатами, которые по существу являются специализированными процессорами вывода, программирование операций для таких плат значительно сложнее, и обычно напрямую недоступно программисту.

[2] Мы рассматриваем, естественно, работу дисплея только в стандартном текстовом режиме, когда на экране располагается 25 строк по 80 символов в каждой строке.

```
out   dx,al; Запись в R15
dec   dx; Порт выбора регистра
mov   al,14
out   dx,al; Выбираем R14
inc   dx; Порт записи в регистры
mov   al,bh; старший байт из bx
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,[1] 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 из определённого регистра дисплея некоторые его *флаги состояния*.

Надеюсь, что этот простенький и сильно упрощённый фрагмент реализации системного вызова не отобьёт у Вас охоту стать 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}). Далее, конечно, системный программист (обычно работающий на заводе-изготовителе данного внешнего устройства), должен написать программу-драйвер, которая предоставляется пользователям вместе с устройством. При работе компьютера этот драйвер будет процедурой-обработчиком соответствующего прерывания.

Главный недостаток этой архитектуры тоже очевиден: пока два устройства обмениваются данными, остальные не могут этим заниматься и должны простаивать (говорят, что они пропускают циклы шины). 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* внутренние шины, соединяющие избранные, самые быстрые устройства (например, центральный процессор и оперативную память). Ясно, однако, что невозможно соединить своими шинами все возможные пары устройств, это просто экономически нецелесообразно, не говоря уже о том, что такую архитектуру очень трудно реализовать.

Здесь можно ещё упомянуть и о трудностях определения состава оборудования, подключённого к общей шине. Действительно, как может центральный процессор (точнее, служебная программа) определить, какие устройства поделючены в данный момент к общей шине? Практически единственная возможность – это посылать сообщения ко всем возможныи портам (а их 2^{16}) в надежде, что некоторые из них "откликнутся", и можно будет прочитать из них тип и характеристики данного устройства. Те пользователи, которым приходилось устанавливать на свой компьютер операционную систему (например, Windows), могли заметить, что когда дело доходило до определения состава подключённого к компьютеру оборудования, программа установки надолго "задумывалась". Частичное решение этой проблемы предоставляют периферийные устройства типа Plug and Play (хорошего перевода нет, что-то вроде "Вставь и сразу пользуйся"). Эти устройства при включении компьютера сами уведомляют операционную систему о своём присутствии и характеристиках.

[1] Визуально это иногда могло наблюдаться на экране как эффект идущего "снега".

Исходя из таких очевидных недостатков архитектуры с общей шиной, была разработана и другая архитектура связи устройств компьютера между собой. Обычно в литературе она называется архитектурой с каналами ввода/вывода [1,3].

15.2.3. Архитектура ЭВМ с каналами ввода/вывода

Архитектура ЭВМ с каналами ввода/вывода предполагает возможность параллельной работы нескольких устройств компьютера. Поймём сначала, какие же работы при общении с внешними устройствами нам надо производить параллельно. Оказывается, что в основном нужно обеспечить параллельный обмен данными нескольких устройств с оперативной памятью (такая оперативная память называется *многовходовой*). Действительно, когда мы рассматривали мультипрограммный режим работы ЭВМ, мы говорили, что для эффективного использования аппаратных 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 должна параллельно работать 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 вспомогательной памяти типа кэш)[1].

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). Машинные языки каналов ввода/вывода обычно проще машинного языка центрального процессора, так как они предназначены только для узкой задачи описания алгоритмов обмена данными между компьютером и "внешним миром".

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 computers). Здесь имеется в виду, что все внешние устройства, а, следовательно, и пользователи, могут общаться с центральной частью компьютера (обычно это супер-ЭВМ) только через эти машины переднего плана. Кроме того, эти машины 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 супер-ЭВМ 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*.[2] 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. 15.4 показана архитектура ЭВМ с каналами ввода/вывода.

---

[1] Отсюда ясно, что свой кэш может использоваться как буфер между оперативной памятью и внешними устройствами. Например, обмен между оперативной памятью и магнитным диском производиться через дисковый кэш, выполняющий те же функции, что и кэш между оперативной и регисровой памятью, и т.д.

[2] На современных персональных ЭВМ хорошим примером такого мальтиплексного канала является так называемый контроллер USB шины, к нему может подключаться до 127 различных внешних устройств.

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 работает как бы с некоторыми *обобщёнными* внеш-

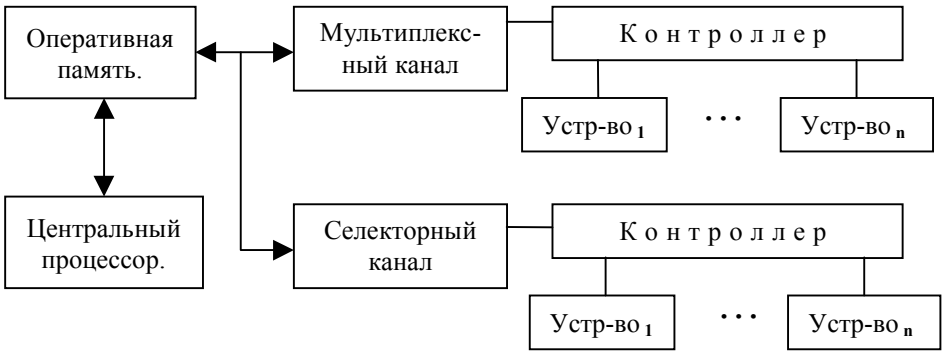


Рис. 15.4. Схема ЭВМ с каналами ввода/вывода.

ними устройствами, а все особенности связи с конкретными устройствами реализуются в контроллерах. Например, один контроллер предназначен для подключения к каналу жёстких дисков, другой – архивных накопителей на магнитной ленте (так называемых стриммеров), третий – накопителей на лазерных дисках 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, так как центральный процессор занят счётом служебной процедуры.

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 может производиться немедленный возврат на продолжение выполнения 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]

Естественно, что *немедленное* продолжение выполнения программы пользователя после системного вызова возможно только в том случае, если этой программе не требуется *немедленно* 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 диспетчером в состояние ожидания, пока канал полностью 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. Для предотвращения таких ситуаций существуют особые аппаратные и программные средства, позволяющие, как говорят, *синхронизировать* параллельную работу нескольких устройств. С этими средствами Вы познакомитесь в другом курсе. Эти примеры, в частности, показывают, насколько усложняются большинство программ операционной системы в архитектуре с каналами ввода/вывода по сравнению с архитектурой с общей шиной.

В заключение нашего краткого рассмотрения организации взаммодействия центральных и периферийных частей компьютера стоит отметить, что в настоящее время архитектура с общей шиной в

[1] В теории программирования ввод/вывод называется блокирующим, если программа, производящая операцию обмена, ждёт полного завершения операции ввода/вывода, и лишь потом продолжает свой счёт. До сих пор в языках Паскаль и Ассемблер Вы имели дело только с таким блокирующим вводом/выводом. Неблокирующий ввод/вывод позволяет производить счёт программы *параллельно* с операцией обмена. Как видим, архитектура ЭВМ с каналами позволяет производить и неблокирующий ввод/вывод.

чистом виде встречается только в самых простых ЭВМ (обычно в тех специализорованных компьютерах, которые встраиваются в стиральные машины, холодильники, автомобили и т.д.). Даже современные персональные ЭВМ массового выпуска содержат в своей архитектуре, помимо общей шины, различные средства прямого обмена с оперативной памятью (DMA – Direct Memory Access), похожие на простейшие каналы ввода/вывода (правда, обычно без средств их программирования пользователем).

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 значительно большей степени, на *параллельной обработке данных*. Это связано с тем, что увеличение быстродействия каждого отдельного узла ЭВМ уже приближается к тем критическим ограничением, которые накладывает конечность скорости света. Исходя из этого, мы 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.[1] В таких компьютерах много потоков команд одновременно обрабатывают много потоков данных, в научной литературе это обозначается сокращением МКМД (или по-английски MIMD – Multiple Instruction Multiple Data).
- Параллельные процессы в рамках одной программы. Программа пользователя может породить несколько параллельных вычислительных процессов обработки данных, каждый такой процесс для операционной системы является самостоятельной единицей работы (с этой возможностью мы уже сталкивались, когда изучали реентерабельные программы). Таким образом, вычислительные процессы *одной* задачи могут псевдопараллельно выполняться в мультипрограммном режиме точно так же, как и процессы, порождаемые задачами других пользователей.

  В качестве примера рассмотрим простой 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 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 другом курсе.[2]

---

[1] В настоящее время популярен также подход, при котором большое количество самостоятельных ЭВМ объединяются высокоскоростными шинами связи в вычислительный комплекс, называемый *кластером*. Такие кластеры могут состоять из десятков, сотен и даже тысяч компьютеров, которые все могут параллельно решать одну большую задачу.

[2] Студенты факультета вычислительной математики и кибернетики МГУ изучают это в курсе третьего семестра "Системное программное обеспечение".

- Параллельное выполнение нескольких команд одной программы может производиться, как мы знаем, *конвейером* центрального процессора. В мощных ЭВМ центральный процессор может содержать и 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 одной программе производится на так называемых *векторных* и *матричных* ЭВМ.

  У **векторных** 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 с именами `axv` и `bxv`, каждый из которых может хранить вектор из 64 чисел, тогда, например, команда векторного сложения **`addv axv,bxv`** будет производить параллельное покомпонентное сложение всех элементов таких векторов по схеме `axv[i]:=axv[i]+bxv[i]`.

  У **матричных** ЭВМ на одно устройство управления приходится много (иногда до нескольких сотен или даже тысяч) арифметико-логических устройств. Таким образом, выбранная в устройство управления команда, например сложения, параллельно исполняется над разными операндами в каждом из арифметико-логических устройств. Можно сказать, что на векторных и матричных ЭВМ один поток команд обрабатывает много потоков данных. Отсюда понятно сокращённое название ЭВМ такой архитектуры – ОКМД (по-английски SIMD: Single Instruction Multiple Data).

Деление архитектур ЭВМ по способу организации вычислительного процесса (ОКОД, МКМД и ОКМД) называется в компьютерной литературе *классификацией по Флинну*. Для полноты этой классификации следует упомянуть и архитектуру МКОД (MISD – Multiple Instruction Single Data), при этом один поток данных обрабатывается параллельно многими потоками команд. В качестве несколько надуманного примера можно привести специализированную ЭВМ, управляющую движением самолётов над крупным 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 входных данных, и рассматривать это как частный случай архитектуры МКМД, так и делается во многих учебниках, в которых класс МКОД считается пустым.[1]

Параллельная обработка команд и данных позволяет значительно увеличить производительность компьютера. Необходимо сказать, что в современных высокопроизводительных компьютерах обычно реализуется сразу несколько из рассмотренных выше уровней параллелизма. Познакомится с историей развития параллельной обработки данных можно, например, по книгам [15,23]. 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 подлежащих решению задач всё время *опережает* рост производительности компьютеров. Например, только для составления *местного* (например, для Московской области) и краткосрочного (на одни сутки вперёд) *достоверного* прогноза погоды необходим компьютер с производительностью порядка 100 млрд. операций над вещественными числами в секунду [1].

Производительность мощных компьютеров меряется в единицах, называемых *флопами* (flps). Один flps равен одной операции над *вещественными* числами в секунду (floating point per second). Подчеркнём, что это именно операции не над целыми, а над вещественными числами, которые для компьютера значительно более трудоёмкие. Таким образом, для расчёта нашего прогноза погоды необходима производительность компьютера в 100 Gflops (100 Гигафлоп). Заметим, что производительность мощных современных супер-ЭВМ составляет порядка десятков и сотен Терафлоп ($1\ \text{Tflps} = 1000\ \text{Gflps} = 10^{12}\ \text{flps}$).

[1] Данная классификация компьютеров предложена в 1966 году Майклом Флинном, мы привели её в несколько урезанном виде. У самого Флинна эта классификация более разветвлённая, в каждом классе выделяются свои подклассы по способам связи между собой элементов вычислительной системы и единицам обраюатываемых данных.

Например, производительность суперкомпьютеров серии IBM Blue Gene/L выпуска 2005 года оценивается примерно в 138 Tflps.[1] Это мощная вычислительная система, имеющая в своём составе 2^{16} процессоров, которые, в принципе, могут параллельно решать одну и ту же задачу. Разумеется, ЭВМ этого класса выпускаются в единичных экземплярах по специальному заказу. Необходимо также учитывать, что такая высокая производительность достигается суперкомпьютерами только на специальных задачах, допускающих глубокое распараллеливание вычислений. При выполнении "обычных" программ производительность суперкомпьютеров может упасть в несколько сотен и даже тысяч раз.

[1] Мощность ЭВМ для работы с вещественными числами часто оценивается с помощью так называемых тестов LINPACK. Данные о 500 самых мощных компьютерах по этой системе тестирования приведены на сайте http://www.top500.org.

Глава 16. Дополнительные особенности архитектуры ЭВМ

В этой главе мы рассмотрим некоторые особенности архитектуры ЭВМ, которые, в основном из-за недостатка времени, (пока?) не входят в наш основной курс, но могут оказаться полезными для лучшего понимания основ архитектуры современных ЭВМ.

16.1. Дискретные и аналоговые вычислительные машины

При рассмотрении в прошлом семестре основных свойств алгоритмов отмечалось такое важное свойство алгоритма, которое называлось *дискретностью*. Это свойство алгоритма означало, что любой достаточно сложный алгоритм состоит из этапов или шагов, причём каждый шаг, в свою очередь, тоже является алгоритмом. Это же свойство структурности алгоритма позволяло нам строить из одних алгоритмов другие по определённым правилам, например, путём составления суперпозиции (последовательного выполнения) двух алгоритмов.

Как мы знаем, если некоторый алгоритм *применим* к определённым входным данным, то он вводит эти данные,[1] обрабатывает их за конечное число шагов и останавливается с выдачей результата. Содержание наименьшего шага алгоритма определяется исполнителем этого алгоритма, например, для Паскаль-машины это оператор языка Паскаль, а для компьютера – машинная команда. Разумеется, как уже говорилось, мы можем спуститься на следующий уровень изучения архитектуры компьютера и рассмотреть шаги выполнения одной команды. Далее можно рассмотреть реализацию каждого шага выполнения команды в виде интегральной схемы, составленной из минимальных логических элементов – вентилей. Как уже говорилось, каждый вентиль срабатывает, когда на него приходит специальный тактовый импульс – можно считать, что на инженерном уровне в*и*дения архитектуры это и есть логический предел дробления алгоритма на всё более мелкие шаги для исполнителя-компьютера.

Компьютеры, построенные по рассмотренному выше принципу, так и называются – **дискретными** (или цифровыми) вычислительными машинами. Сейчас мы постараемся понять, что это не единственно возможный принцип построения устройства автоматической обработки данных (компьютера).

Сначала рассмотрим, как решает задачи обработки данных человек. Оказывается, что он может делать это двумя принципиально различными путями. Один путь называется логической обработкой данных, например, рассмотрим, как сыщик узнаёт преступника по его словесному портрету. "Подбородок квадратный, глаза узкие, нос прямой, на кисти левой руки небольшой шрам. При ходьбе припадает на правую ногу, при общении щурит левый глаз, говорит без акцента и т.д." Здесь явно проглядывает уже знакомое нам разбиение алгоритма узнавания на отдельные шаги. За такой процесс обработки данных у человека отвечает так называемой *логическое мышление*.

В то же время, как мы все знаем, это не единственный способ узнать человека. Так, мы "с одного взгляда" или, как говорят, интуитивно, узнаём в толпе своего приятеля, и при этом совсем не разбиваем для себя это узнавание ни на какие отдельные шаги. Такой процесс обработки информации человеком называется *интуитивным мышлением*, этот процесс на осознаваемом уровне восприятия не обладает свойством дискретности, хотя, конечно, и занимает некоторый отрезок времени. Как установлено, за логическое и интуитивное мышление отвечают две разные половины (полушария) человеческого мозга – левое и правое.[2]

В нашем курсе мы изучаем архитектуру дискретных (цифровых) вычислительных машин, можно сказать, что они обрабатывают входные данные по принципу, очень похожему на логическое мышление человека. Резонно задать вопрос, а можно ли построить устройство автоматической обработки

---

[1] Конечно, более строго надо было бы сказать, что это делает *исполнитель* алгоритма, но в литературе обычно так говорить не принято.

[2] Не стоит из праздного любопытства запоминать, за какое мышление отвечает левое, а за какое правое полушарие человеческого мозга, так как у людей-левшей всё обычно будет наоборот. Заметим также, что не стоит понимать всё это слишком буквально, на самом деле оба полушария обрабатывают данные совместно, обмениваясь при этом между собой информацией. Можно говорить только об определённой *специализации* полушарий на логическую и интуитивную обработку данных.

данных (компьютер), которое работало бы по принципу, похожему на интуитивное мышление человека? [1]

Да, такие компьютеры возможны, в отличие от уже знакомых нам *дискретных* ЭВМ они называются *аналоговыми* (или непрерывными) компьютерами. Более того, в начале развития вычислительной техники аналоговые ЭВМ были весьма распространены. Разберём вкратце, как работали такие ЭВМ (мы рассмотрим их архитектуру только на концептуальном уровне).

Одним из принципов работы аналоговых ЭВМ является аналоговое представление данных. Мы с Вами привыкли представлять данные в виде чисел, состоящих из определённого количества цифр (неважно, в какой системе счисления), такое представление данных называется цифровым или *дискретным*. Теперь необходимо понять, что данные можно представить и в непрерывной форме, например, в виде определённого напряжения на конденсаторе, расположением стрелок на часах или положением подвижной части логарифмической линейки (кто не знает, это такой странный калькулятор раньше был ☺). Так вот, аналоговые ЭВМ обрабатывали данные, представленные в такой непрерывной форме (обычно в виде электрических напряжений и токов).

Процесс решения задачи аналоговой ЭВМ заключается в том, что входные данные подаются на входное устройство такой ЭВМ в виде набора электрических напряжений. Затем, через определённый (не очень большой) промежуток времени выходные данные (результаты решения задачи) появлялись на устройстве вывода, тоже в виде набора электрических напряжений. Более всего такой ответ походит на решение дифференциальных уравнений, представленное в виде графиков, а не числовых таблиц. Разумеется, выходные данные могли потом преобразовываться устройством вывода в числовой вид, печататься и выдаваться пользователю.

Аналоговые ЭВМ не поддаются программированию в привычном для нас виде, так как они преобразуют входные данные в выходные "за один шаг" своей работы. Можно сказать, что аналоговая ЭВМ *настраивается* на конкретную задачу, которую необходимо решить. Чтобы понять, в чём заключается такая настройка на решаемую задачу, посмотрим более пристально на процесс программирования на машинно-ориентированном языке для привычных для нас дискретных ЭВМ.

Составляя алгоритм решения некоторой задачи на машинно-ориентированном языке, программист, как мы знаем, записывает этот алгоритм в виде последовательности команд для конкретной ЭВМ. Можно сказать, что алгоритм решения задачи *отображается* на язык машины, подстраиваясь к особенностям этого языка и архитектуры самого компьютера. Такое отображение, как мы знаем, является достаточно трудным делом, так как язык машины очень далёк от того языка, на котором прикладной программист мыслит в рамках своей задачи.

Например, известно, что очень многие физические задачи описываются на языке дифференциальных и интегральных уравнений, который является "естественным" языком для описания таких задач.[2]

В отличие от дискретных ЭВМ, аналоговые компьютеры реализуют другой **принцип** решения задач. Вместо того чтобы отображать задачу на язык машины (т.е. "приспосабливать" задачу к машине), как делается на дискретных ЭВМ, аналоговые ЭВМ *изменяют свою структуру*, чтобы самим соответствовать решаемой задаче! Таким образом, аналоговые ЭВМ имеют очень "гибкую" архитектуру, узлы такой ЭВМ могут по разному соединяться друг с другом, "настраиваясь" на решаемую задачу.[3] Заметим, что сам принцип "настройки" исполнителя под конкретную задачу нам уже встречался. Например, как мы знаем, для машины Тьюринга правильнее говорить не "написать программу, решающую данную задачу", а "*построить* машину Тьюринга, решающую данную задачу". В такой машине Тьюринга будет нужное для конкретной задачи число состояний, она будет распознавать нужный набор символов и т.д.

---

[1] Рассматриваемый вопрос не имеет никакого отношения к так называемым системам искусственного интеллекта, т.е. комплексам *программ*, имитирующим процесс обработки данных человеком в отдельных предметных областях.

[2] Студенты факультета Вычислительной математики и кибернетики МГУ уже сталкивались с решением одной такой задачи в рамках практикума работы на языке Паскаль, и должны представлять, как далёк язык машины (да и используемый при решении язык Паскаль) от задачи вычисления определённого интеграла.

[3] Некоторые из дискретных компьютеров также могут в определённых (весьма небольших) пределах изменять связи между своими узлами, подстраиваясь под решаемую задачу. Компьютеры с такой архитектурой называются *транспьютерами*, они не очень широко распространены. К сожалению, их рассмотрение далеко выходит за рамки нашего начального курса по архитектуре ЭВМ.

В некотором смысле аналоговая ЭВМ похожа на детские конструкторы, с которым некоторым из Вас приходилось иметь дело в юные годы. Как Вы знаете, по-разному соединяя детали такого конструктора, можно собирать **модели** самых разных предметов (подъёмный кран, качели, стул и стол и т.д.). Точно так же аналоговая ЭВМ, изменяя свою структуру, строит (электрическую) **модель** решаемой задачи. На практике настройка аналоговой ЭВМ на решаемую задачу состоит в установлении с помощью проводов многочисленных электрических соединений между узлами такого компьютера, и изменению электрических характеристик (сопротивлений, ёмкостей и индуктивностей) этих узлов с помощью соответствующих элементов управления (ручек, кнопок, ползунков и т.д.).

Здесь необходимо сказать, что использование в качестве "рабочего тела" в аналоговых ЭВМ именно электричества не является принципиальным. В истории вычислительной техники известна реализация аналоговой ЭВМ, работающей на обыкновенной воде. В этом случае конструктивными элементами являются водяные резервуары с соединяющими их трубками разного сечения, а текущая вода моделирует решаемую задачу путём изменения скорости своего течения и давления в разных узлах такой аналоговой "ЭВМ". Кроме того, в качестве "рабочего тела" можно использовать, например, звуковые волны, в этом случае конструктивными элементами являются различные плоскости и решётки, отражающие, поглощающие и модулирующие звуковые волны.[1]

Далее, стоит заметить, что в детских конструкторах строятся в основном *статические* модели, в то время как аналоговая ЭВМ строит *динамическую* модель некоторого объекта или явления. В качестве примера динамической модели из детского конструктора рассмотрим модель ветряной мельницы, которая *на самом деле* может крутиться под действием ветра, да ещё и изменяет наклон своих лопастей в зависимости от скорости ветра. Ясно, что изучение такой модели в принципе может помочь нам в конструировании *настоящих* ветряных мельниц, позволяющих эффективно использовать силу ветра.

Продолжая аналогию с детским конструктором можно заметить, что с его помощью можно собирать далеко не всякие модели, а только из определённого набора (по-научному это называется – из определённой *предметной области*). Если вспомнить, то конструкторы бывают механические, электрические, оптические, радио-конструкторы и т.д. Отсюда можно сделать вывод, что всякая аналоговая ЭВМ тоже *специализированная*, предназначенная для решения задач в достаточно узкой предметной области. В то же время надо заметить, что почти все *цифровые* ЭВМ являются *универсальными*, их можно с почти одинаковым успехом использовать во всех областях, это использование ограничивается только аппаратными ресурсами компьютера, которые, как мы знаем, непрерывно увеличиваются с развитием вычислительной техники.[2]

Аналоговая ЭВМ реализует достаточно большую скорость обработки информации, т.к. у неё отсутствует пошаговый алгоритм работы. Кстати, это же верно и для "аналогового компьютера" из соответствующего полушария человеческого мозга. Например, когда человек бегом спускается с холма, именно это полушарие "руководит" движениями человека, в частности, решая, как ему переставлять ноги. Попытка решить такую же задачу *в реальном времени* с помощью программы для обычного (цифрового) компьютера показывает, что с этой задачей с трудом справляются даже мощные современные супер-ЭВМ.

Однако, несмотря на большую скорость отработки информации, в настоящее время аналоговые ЭВМ практически проиграли в конкурентной борьбе с дискретными компьютерами. Дело заключается в том, что у аналоговых ЭВМ, даже если не принимать во внимание специфику их "программирования", есть один очень крупный недостаток, который заключается в том, что они могут выдавать

---

[1] Исследования последних лет позволяют предположить, что такого рода аналоговая звуковая "микро-ЭВМ" располагается внутри каждого нейрона нервной ткани человека и животных. Основой этого аналогового компьютера является особая решётка, работающая на звуковых волнах очень большой частоты. Эти звуковые волны генерируются ударами ионов о стенки нейрона и саму решётку, такие волны имеют частоты от 10 до 100 ГГц, что превышает тактовую частоту центральных процессоров современных ЭВМ.

[2] Разумеется, существуют и так называемые *специализированные* цифровые ЭВМ, например, предназначенные для управления различными устройствами (стиральными машинами, кухонными комбайнами и т.д.). Необходимо, однако, заметить, что специализация таких ЭВМ заключается только в том, что они, как правило, выполняют только одну записанную в их памяти программу, и не снабжаются аппаратными средствами для развитого интерфейса с пользователями. Другими словами, хранимую в памяти специализированных ЭВМ программу невозможно (или очень трудно) сменить, и у них нет таких устройств для ввода и вывода данных, как клавиатура, мышь, дисковод и т.п.

числовые результаты своей работы с маленькой точностью – две-три значащие десятичные цифры. Для большинства современных научных задач это совершенно неприемлемая точность вычислений, поэтому аналоговые ЭВМ имеет смысл использовать только в тех областях, где обрабатываемые данные имеют преимущественно нечисловую природу (например, в так называемых задачах распознавания образов). Заметим, что в своё время были реализованы и гибридные (аналогово-цифровые) ЭВМ, однако они тоже не оправдали возлагаемых на них надежд. С принципами работы аналоговых ЭВМ можно познакомится , например, по книге [22].

В заключение нашего краткого рассмотрения аналоговых ЭВМ заметим, что сам *принцип* их работы ничем не хуже принципа работы дискретных (цифровых) компьютеров. Более того, многие современные направления развития вычислительной техники, например, такие, как нейрокомпьютеры или квантовые вычисления, плодотворно используют идеи, взятые из схемы работы аналоговых ЭВМ.

16.2. Принцип микропрограммного управления

Принцип микропрограммного управления задаёт особую схему реализации центрального процессора ЭВМ. Чтобы разобраться в этом вопросе, вернёмся мысленно в начало нашего курса и вспомним, как центральный процессор выполняет отдельную команду программы. Пусть, например, на регистр команд трёхадресной ЭВМ выбрана некоторая команда `КОП op1,op2,op3`. Как мы знаем, какая команда обрабатывается центральным процессором по схеме

```
R1:=op2; R2:=op3; S:=R1 КОП R2; op1:=S
```

где `R1,R2` и `S` – служебные регистры центрального процессора. Как видим, сама команда тоже выполняется по некоторому алгоритму, состоящему из отдельных шагов, а, следовательно, можно составить *программу* выполнения каждой команды компьютера. Вот такой "полёт мысли": компьютер выполняет программу, состоящую из команд, а каждая команда, в свою очередь, выполняется по некоторой своей программе.

Несмотря на кажущуюся абсурдность приведённой выше схемы работы ЭВМ, она вполне имеет право на существование и даже обладает определёнными и существенными преимуществами. ЭВМ, построенные по изложенной выше схеме, и называются компьютерами с микропрограммным управлением.[1] При этом, как обычно, в распоряжении программиста находится язык машины,[2] на котором он записывает свои программ. В то же время, алгоритм выполнения каждой команды из языка машины реализуется в виде отдельной программы на специальном *микроязыке.* Эту программу, естественно, и называют *микропрограммой*. Все микропрограммы хранятся в специальной очень быстродействующей памяти внутри центрального процессора. После считывания из памяти на регистр команд очередной команды, центральный процессор по коду операции находит соответствующую ей микропрограмму и выполняет её, затем, как всегда, читает на регистр команд следующую команду и т.д.

Микроязык содержит весьма небольшое количество различных микрокоманд, которые реализуют такие простые действия, как пересылка операндов с одного места памяти в другое, целочисленное сложение, сравнения операндов на равенство, некоторые сдвиги, логические операции **`and`**, **`or`** и **`not`** и т.п. Вследствие того, что теперь центральному процессору надо уметь выполнять только небольшое число простых микрокоманд, его архитектура очень сильно упрощается, теперь его можно делать из очень быстро работающих (и дорогих) интегральных схем. Следовательно, несмотря на то, что теперь каждая команда выполняется по микропрограмме, можно избежать значительного снижения производительности компьютера, построенного по принципу микропрограммного управления.

Разберём теперь те преимущества, которые открывает принцип микропрограммного управления. Во-первых, из-за небольшого числа простых микрокоманд в центральном процессоре легко реализовать несколько быстродействующих конвейеров, сильно повышающих производительность ЭВМ.

Во-вторых, набор микропрограмм можно менять. Обычно это делается при включении машины, когда предоставляется возможность загрузить в память микропрограмм новый набор микрокоманд.

---

[1] Первым предложил использовать принцип микропрограммного управления Морис Уилкс в 1951 году.

[2] Имеется в виде программист на языке Ассемблера, он, как, впрочем, и прикладной программист, имеет право ничего не знать о том, что его компьютер работает по принципу микропрограммного управления. Другими словами, принцип микропрограммного управления находится не на *внутреннем*, а на инженерном уровне в*и*дения архитектуры ЭВМ.

Эта уникальная возможность позволяет в значительной степени менять архитектуру компьютера. Действительно, можно в очень широких пределах менять язык машины, например, сменить двухадресную систему команд на трёхадресную. Можно также изменить способы адресации, поменять форматы обрабатываемых данных (скажем, ввести 128-битные целые числа и новые команды, которые их обрабатывают).

Компьютеры с микропрограммным управлением были достаточно широко распространены в 60-х и 70-х годах прошлого века, обычно на этих принципах строились ЭВМ средней и высокой производительность (но не супер-ЭВМ). В настоящее время, в связи с резким удешевлением элементной базы компьютеров и значительным увеличением скорости работы интегральных схем, принцип микропрограммного управления в значительной мере потерял свою актуальность и чистом виде применяется редко.

В современных компьютерах процесс выполнения команд, похожий на микропрограммное управление, используется при работе конвейера центрального процессора. Как мы уже знаем, каждая команда выполняется на конвейере за несколько шагов, которые можно рассматривать как некоторый аналог микрокоманд. В то же время эта аналогия достаточно условная, так как последовательность шагов для каждого конвейера фиксирована, и писать свои микропрограммы для выполнения команд машинного языка невозможно (и тем более нельзя таким образом ввести в язык машины *новые* команды). Другим "отзвуком" принципа микропрограммного управления можно считать способность центральных процессоров современных ЭВМ динамически (во время счёта) заменять сложные команды машинного языка на последовательность более простых, которые и подаются на конвейер для выполнения.

Идеи, похожие на принцип микропрограммного управления, заложены в ЭВМ так называемой RISC архитектуры (Reduced Instruction Set Code – компьютеры с уменьшенным набором кодов операций). Основная идея здесь состоит в том, чтобы оставить в машинном языке только небольшой набор (порядка 50-ти) простый команд, а все остальные операции реализовывать как программы в этом урезанном машинном языке. При этом делают так, чтобы все команды в таком компьютере имели одинаковую длину, производили операции над данными только в регистрах (т.е. имели формат RR) и выполнялись за одинаковое время (обычно за один дакт). Такой подход позволяет сильно упростить структуру центрального процессора и реализовать в нём очень эффективный конвейер. RISC архитектура показала свою жизнеспособность и в настоящее время используется в нескольких выпускающихся семействах ЭВМ (например, SUN SPARC, Power PC и др.).

16.3. ЭВМ, управляемые потоком данных

Все рассматриваемые нами до сих пор ЭВМ базировались на изученных нами принципах Фон Неймана. Как уже говорилось, все современные ЭВМ в той или иной мере отказываются от большинства этих принципов для повышения своей производительности. Тем не менее, все рассмотренные до сих пор архитектуры ЭВМ продолжают следовать одному *основополагающему* принципу Фон Неймана. Это принцип программного управления, который состоит в том, что ЭВМ автоматически обрабатывает *данные*, выполняя расположенную в своей памяти *программу*. То, что именно программа должна обрабатывает данные, представляется большинству программистов совершенно очевидным. Разберёмся, однако, с этим вопросом более подробно.

Как Вы помните, ЭВМ является алгоритмической системой, в частности, она является *исполнителем* алгоритма на языке машины. В прошлом семестре Вам давали примерно следующее *неформальное* определение алгоритма. "Алгоритм – это чёткая система правил (шагов алгоритма). Обрабатывая входные данные, к которым алгоритм *применим*, исполнитель алгоритма за конечное число выполнения своих шагов остановится и выдаст результат (выходные дванные)".

Как видим, по самому определению алгоритма он выполняет в нашей системе *главную* роль, а обрабатываемые данные – *подчинённую*. В начале развития программистской науки такое положение вещей было очевидным, оно находило своё отражение и в "информативности" программы и данных. Например, изучая текст определённой программы (даже без комментариев), программист мог выяснить, что она предназначена, например, для вычисления суммы элементов некоторого массива. В то же время, сколько бы программист не рассматривал числа, составляющие входной массив данных, он, не видя программы, конечно, не может догадаться, для чего предназначены эти данные. Другими словами, если принять во внимание, что программу в совокупности с её данными можно рассматривать как *модель* некоторого объекта, то большая часть сведений об этом объекте была сосредоточена

именно в программе, а не в обрабатываемых данных. Например, исследуя программу решения систем линейных уравнений, мы сможем понять, для чего предназначена эта программа. В то же время, как бы тщательно мы не изучали вводимые этой программой данные (т.е. числовые матрицы), мы вряд ли поймём, в чём заключается задача. Можно сказать, что в некотором смысле в этой задаче программа играет главную роль, а обрабатываемые данные – второстепенную.

Такая ситуация сохранялась в программировании примерно до начала 80-х годов прошлого века. Далее, однако, обрабатываемые данные непрерывно усложнялись, пока, наконец, для некоторых задач не превзошли по сложности программы, которые обрабатывали эти данные. Рассмотрим в качестве примера большую базу данных, которая хранит текущее состояние дел некоторого крупного предприятия. Можно сказать, что база данных содержит сложно структурированную, изменяющуюся во времени *модель* этого предприятия. Программы, обрабатывающие информацию из базы данных (БД) называются обычно системой управления базой данных (СУБД) [20]. СУБД позволяет вводить, удалять и модифицировать данные в БД, а также обрабатывать запросы на поиск и выдачу из БД нужных сведений. Так вот, сколько бы программист не исследовал программы, входящие в СУБД, он практически ничего не узнает о самом предприятии, ни что оно выпускает, ни сколько человек на нём работает и т.д. Очевидно, что в нашей модели предприятия (БД+СУБД) данные играют основную роль, а обрабатывающие их программы – уже второстепенную.

Как Вы можете догадаться, примерно в это же время появилась идея коренным образом изменить архитектуру компьютера так, чтобы отказаться от принципа *программного управления*. Таким образом, если компьютеры традиционной архитектуры управляются потоком (или потоками) команд, то компьютеры новой, нетрадиционной архитектуры должны управляться потоком данных. Можно сказать, что не команды должны определять, когда какие данные надо обрабатывать, а, наоборот, данные выбирают для себя действия (операторы), которые в определённый момент надо выполнить над этими данными. Компьютеры такой архитектуры принято называть потоковыми ЭВМ (по-английски DFC – Data Flow Computers) [1,16].

Заметим, что сам по себе принцип потоковой обработки данных не представляет собой ничего загадочного или экзотического. Например, отметим, что уже изученные Вами ранее такие алгоритмические системы, как машина Тьюринга и Нормальные алгоритмы Маркова были обработчиками данных именно этого класса. Действительно, например, в машине Тьюринга именно *данные* (текущий символ, на который указывает головка), определял, какие именно *операции* необходимо было выполнить на этом шаге работы!

В то же время оказалось, что архитектура потоковых ЭВМ весьма сложна и сильно отличается от архитектуры традиционных ЭВМ. Например, в устройстве управления таких ЭВМ нет никакого счётчика адреса, а в их памяти нет программы (по крайней мере, в нашем традиционном понимании). Мы рассмотрим функционирование такой ЭВМ на одном очень простом примере.

Пусть нам необходимо вычислить такой оператор присваивания:

```
x := (x+y)*p + (x+p)/z - p*y/z
```

Представим этот оператор в виде так называемого графа потока данных, показанного на рис. 16.1. В этом ориентированном графе есть два вида узлов: это сами обрабатываемые данные, изображённые в виде квадратиков (в нашем примере – значения переменных `x,y,z` и `p`) и *обрабатывающие элементы* потоковой ЭВМ, которые мы обозначили просто знаками соответствующих арифметических операций в кружочках.[1]

Основная идея потоковых вычислений состоит в следующем. Все действия над данными производят обрабатывающие элементы (операторы), в нашем примере эти элемента обозначены арифметическими операциями сложения, вычитания умножения и деления. Можно сказать, что всё арифметико-логическое устройство потоковой ЭВМ – это набор таких обрабатывающих элементов. Каждый обрабатывающий элемент начинает автоматически выполняться, когда есть в наличие (готовы к обработке) требуемые для него данные. Так, в нашем примере автоматически (и параллельно!) начинают выполняться операторы `+` , `+` и `*` во второй строке нашего графа, затем, на втором шаге работы, параллельно выполняются операторы `*` , `/` и `/` в третьей строке и т.д. Здесь просматривается большое сходство со способом работы электронных схем, составленных из вентилей. Действительно, если граф потока данных "положить на левый бок", то он будет весьма похож на

[1] Заметка на будущее для продвинутых студентов: граф потока данных базируется на принципах построения так называемых сетей Петри.

электронную схему двоичного сумматора, как мы показывали её на рис. 2.2а). При этом роль вентилей будут играть обрабатывающие элементы, а входных и выходных сигналов – обрабатываемые данные.

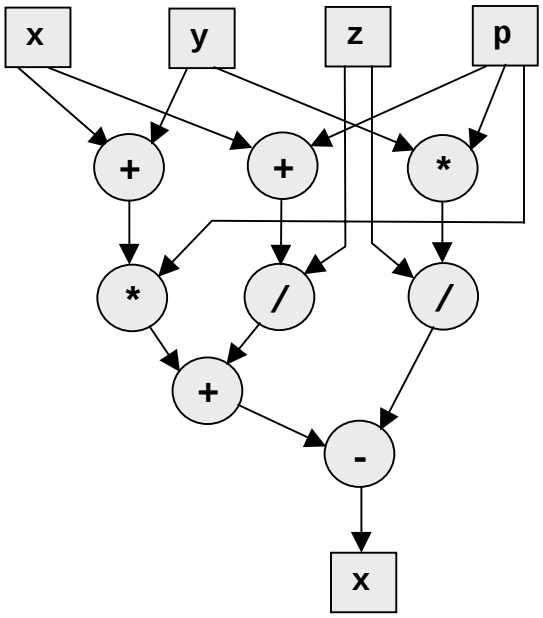


Рис.16.1. Граф потока данных для оператора присваивания.

Заметим, что обрабатываемые данные в потоковом компьютере вовсе не являются *пассивными*, как в ЭВМ традиционной архитектуры, наоборот, они "громко заявляют" о своей готовности к обработке (можно сказать, требуют к себе "внимания" со стороны обрабатывающих элементов). Вообще говоря, здесь есть все основания отказаться от принципа Фон Неймана, согласно которому память только *хранит* данные, но не обрабатывает их. Другими словами, представляется естественным совместить функции хранения и обработки данных, и разместить обрабатывающие элементы не в арифметико-логическом устройстве, а прямо в оперативной памяти.

Вообще говоря, граф потока данных и является "программой" для потоковой ЭВМ, а программ в нашем привычном понимании (запись алгоритма на языке машины) здесь не существует. Можно сказать, что здесь сами данные управляют процессом своей обработки. Отсюда можно сделать вывод, что обычные языки программирования плохо подходят для записи алгоритмов обработки данных в потоковых ЭВМ, так как приходится делать сложный компилятор, преобразующий обычную последовательную программу в граф потока данных. Неудивительно поэтому, что вместе с идеей потоковых ЭВМ появились и специальные языки потоков данных (Data Flow Languages),[1] ориентированные на прямое описания графа потока данных [1].

Из рассмотренной схемы обработки данных в потоковых ЭВМ можно сделать вывод, что в них может быть реализован принцип *максимального параллелизма* в обработке данных, так как любые готовые данные тут же могут поступать на выполнение соответствующему обрабатывающему элементу потоковой ЭВМ. Ясно, что *быстрее* решить задачу просто невозможно.

Немного подумав, можно выделить две фундаментальные трудности, которые встают при реализации потоковой ЭВМ. Во-первых, для достаточно сложного алгоритма невозможно полностью построить граф потока данных до начала счёта. Действительно, алгоритм вводит свои входные данные и содержит условные операторы, выполнение которых может, в частности, зависеть и от этих входных данных. Следовательно, компилятор с некоторого традиционного языка программирования может построить только некоторый первоначальный граф потока данных, а устройство управления потоковой ЭВМ должно будет в процессе счёта *динамически* изменять этот граф. Ясно, что это весьма сложная задача для аппаратуры центрального процессора потоковой ЭВМ.

Вторая проблема заключается в том, что арифметико-логическое устройство потоковой ЭВМ должно содержать много одинаковых обрабатывающих элементов. Даже для приведённого нами в качестве примера простейшего графа потока данных нужно по два обрабатывающих элемента для

[1] Язык не поворачивается назвать эти языки языками программирования (это каламбур ☺).

выполнения операций сложения и деления. Для реальных алгоритмов число таких одинаковых обрабатывающих элементов должно исчисляться десятками и сотнями, иначе готовые к счёту данные будут долго простаивать в ожидании освобождения нужного им обрабатывающего элемента, и вся выгода от потоковых вычислений будет потеряна (и тогда ни стоило, как говорится, и огород городить). Другая трудность заключается здесь в том, что выход каждого обрабатывающего элемента (результат его работы) может быть подан на вход любого другого обрабатывающего элемента. Такие вычислительные системы называются *полносвязными*. При достаточно большом числе обрабатывающих элементов (а выше мы обосновали, что иначе и быть не может), реализовать все связи между ними становится технически неразрешимой задачей.

Ясно, что перед конструкторами потоковой ЭВМ встают очень большие трудности. В настоящее время универсальные потоковые ЭВМ не нашли широкого применения из-за сложности своей архитектуры, пока существуют только экспериментальные образцы таких ЭВМ.[1]

Несколько лучше обстоит дело, если обрабатывающие элементы в потоковых ЭВМ, сделать достаточно сложными, т.е. предназначенными для выполнения крупных шагов обработки данных. В этом случае каждый такой обрабатывающий элемент можно реализовать в виде отдельного микропроцессора, снабженного собственной (локальной) памятью и соответствующей программой (программой в традиционном понимании). Все такие обрабатывающие элементы объединены высокоскоростными линиями связи (шинами), причём каждый элемент начинает работать, как только он имеет все необходимые входные данные. Эти данные могут располагаться в локальной памяти или поступать на линии связи данного обрабатывающего элемента от других обрабатывающих элементов.

Идея управления обработки информации потоками данных нашла применение и в таком быстроразвивающемся направлении вычислительной техники, как нейрокомпьютеры. Более подробно про систолические массивы и нейрокомпьютеры можно почитать, например, в книге [16].

[1] Информация к размышлению: по каким принципам обрабатывает информацию мозг человека?