В.Л.ГРИГОРЬЕВ

ПРОГРАММИРОВАНИЕ ОДНОКРИСТАЛЬНЫХ МИКРОПРОЦЕССОРОВ

МОСКВА ЭНЕРГОАТОМИЗДАТ 1987

ББК 32.973
Г 83
УДК 681.31—181.48.06

Рецензент Б. М. Каган

Григорьев В. Л.

Г 83 Программирование однокристальных микропроцессоров. — М.: Энергоатомиздат, 1987. — 288 с.: ил.

Рассмотрены программные ресурсы однокристального микропроцессора К1810ВМ86 с фиксированными длиной слова и системой команд, а также вопросы программирования на языке ассемблера. Приведены многочисленные примеры выполнения команд и программ несложных задач.

Для инженерно-технических работников в области вычислительной техники и автоматики, занятых разработкой и применением микропроцессорных систем. Может служить учебным пособием студентам вузов, специализирующимся по электронной обработке данных.

Г $\frac{2405000000-049}{051(01)-87}$ 252-87 ББК 32.973

ПРЕДИСЛОВИЕ РЕДАКТОРА

Бурное развитие и широкое распространение микропроцессоров, микропроцессорных устройств и систем разнообразного назначения (от простейших контроллеров объектов до высокопроизводительных систем обработки данных) породило устойчивый и постоянно возрастающий интерес к публикациям по микропроцессорным средствам со стороны инженерно-технических работников самых разнообразных специальностей. Этот читательский интерес во многом удовлетворен центральными издательствами нашей страны, которые в последние годы выпустили ряд монографий советских и зарубежных авторов, посвященных этой тематике. Эти издания в некоторой степени удовлетворили читательский спрос на литературу, в которой отражен ряд вопросов организации, особенностей работы и специфики применения микропроцессорных устройств и систем. Однако номенклатура средств микропроцессорной техники, выпускаемой отечественной промышленностью, постоянно растет, а сама микропроцессорная техника стремительно совершенствуется и усложняется. Этот процесс сопровождается возрастающей потребностью в публикациях, которые посвящены изложению вопросов, связанных со структурной организацией новейших микропроцессоров, архитектурой систем на их основе, средствами и методами программирования и разработки системного и прикладного программного обеспечения.

В настоящей книге рассматриваются вопросы программирования высокопроизводительного однокристального 16-разрядного микропроцессора К1810ВМ86 на языке ассемблера.

Основная особенность книги заключается в том, что систематическое описание средств языка ассемблера и системы команд микропроцессора позволяет автору подробно и доказательно (с использованием многочисленных примеров программ) изложить особенности внутренней организации микропроцессора К1810ВМ86 и его функционирование в системах обработки данных. Книга является уникальной в том смысле, что в ней с предельной степенью детализации излагаются вопросы программирования микропроцессора К1810ВМ86 на языке ассемблера с тщательным учетом

влияния особенностей работы этого сложного прибора на эффективность ассемблерных программ.

Следует отметить, что микропроцессор К1810ВМ86 предназначен прежде всего для использования в профессиональных персональных компьютерах типа ЕС1840 и в специализированных высокопроизводительных «интеллектуальных» контроллерах объектов и процессов. Сложная внутренняя организация микропроцессора, его система команд с многочисленными и нетривиальными режимами адресации, архитектура систем, основным атрибутом которой является сегментная структура памяти, предопределяют использование языков высокого уровня для программирования микропроцессора и, следовательно, высокоэффективных трансляторов, которые позволяют программисту не знать деталей внутренней организации микропроцессора. Однако в тех случаях, когда в силу специфики задачи или внешних условий работы микропроцессора необходимо обеспечить предельное быстродействие микропроцессора или предельную плотность упаковки прикладных программ в памяти системы, программирование на языке ассемблера является предпочтительным. В то же время программирование на языке ассемблера сопряжено с необходимостью детального знания особенностей внутренней организации микропроцессора. Такое знание редко встречается у специалистов по программированию. Поэтому уникальность данной книги состоит в том, что в ней автор, специалист по микропроцессорной технике, на высоком профессиональном уровне описывает специфику организации микропроцессора К1810ВМ86 и адресует это описание тем программистам, перед которыми стоит задача разработки предельно эффективных прикладных программ, т. е. программ, написанных на языке ассемблера.

Продуманная методика изложения материала, многочисленные примеры программ и хороший иллюстративный материал книги позволят читателю в полном объеме осмыслить детали внутренней организации микропроцессора К1810ВМ86 и особенности его взаимодействия с устройствами памяти и ввода-вывода информации.

Достоинства книги позволяют надеяться, что она окажется полезной для специалистов, связанных с использованием профессиональных персональных компьютеров, и для разработчиков специализированных устройств и систем обработки данных и управления.

В. В. Сташин

ПРЕДИСЛОВИЕ АВТОРА

Появившиеся в начале 70-х годов первые 4- и 8-битные микропроцессоры применялись в основном как программируемые логические устройства в калькуляторах, специализированных управляющих системах и т. д. Массовое производство более сложных 8-битных микропроцессоров привело к разработке систем, ресурсов которых оказалось достаточно для реализации простых задач обработки данных и более сложных управляющих функций. Однако и эти микропроцессоры, типичным представителем которых является микропроцессор КР580ИК80, по диапазону адресации памяти, быстродействию и возможностям обработки данных значительно уступали процессорам мини-ЭВМ.

В начале 80-х годов успехи микроэлектроники позволили довести плотность упаковки до нескольких десятков тысяч транзисторов на кристалле, что привело к появлению микропроцессоров нового поколения. Среди них наиболее известным и перспективным является 16-битный микропроцессор К1810ВМ86, который применяется в персональных компьютерах ЕС1840 и «Искра-1030». Он продолжает тенденцию разработок однокристальных микропроцессоров с фиксированными длиной слова и системой команд. Производительность этого микропроцессора примерно на порядок выше производительности его предшественника КР580ИК80, а по функциональным возможностям он приближается к процессорам мини-ЭВМ и в некоторых отношениях даже превосходит их. Важно подчеркнуть, что архитектуру микропроцессора К1810ВМ86 нельзя рассматривать как простое расширение архитектуры микропроцессора КР580ИК80. Прямая программная совместимость между ними на уровне языка ассемблера отсутствует. Однако ассемблерные программы микропроцессора КР580ИК80 можно преобразовать для выполнения на микропроцессоре К1810ВМ86, но обратный переход очень сложен и вряд ли потребуется на практике.

В микропроцессоре К1810ВМ86 имеются возможности и средства работы с операционными системами, мультипрограммирования, организации мультипроцессорных систем, обработки сложных структур данных и эффективной реализации языков высокого уровня. Адресное пространство памяти равно 1М байту (1 048 576 байт). Он оперирует с форматами данных, которые включают в себя биты, байты, слова (16 бит), длинные слова (32 бита), упакованные и неупакованные десятичные данные, цепочки байт и слов. Система команд включает в себя команды умножения и деления чисел с фиксированной точкой. Хотя число базовых команд около 100, наличие многочисленных режимов адресации привело к тому, что общее число различных машинных команд составляет несколько сотен. Это обстоятельство, а также сложный формат команд с несколькими полями и длиной, варьирующейся от 1 до 6 байт, препятствует разработке программ на машинном языке и делает достаточно сложным программирование на языке ассемблера.

Прямыми следствиями расширения возможностей микропроцессоров являются все более громоздкое и сложное описание их функционирования и усложнение языка ассемблера. Настоящая книга призвана познакомить инженерно-технических работников и студентов старших курсов, специализирующихся в области вычислительной техники, с архитектурой и основами программирования на языке ассемблера микропроцессора К1810ВМ86. Она построена таким образом, что читатель после ее изучения сможет программировать сравнительно простые прикладные задачи, а по мере накопления опыта переходить к более сложным.

Ориентация на язык ассемблера объясняется следующими обстоятельствами. Разработка прикладных программ ведется либо на одном из языков высокого уровня, либо на языке ассемблера, у каждого из которых имеются свои достоинства и недостатки. Основные положительные качества языков высокого уровня связаны с удобством и быстротой программирования, компактностью и «самодокументированием» программ, меньшим числом ошибок и т. п. Главные преимущества языка ассемблера проявляются в уменьшении времени выполнения программ и доступности для программиста всех ресурсов системы, часть из которых недоступна при программировании на языках высокого уровня.

Известно, что в типичной программе значительная часть выполняемых процессором операций приходится на небольшую секцию или несколько секций программы. Поэтому на практике целесообразно локализовать такие секции и запрограммировать их как ассемблерные модули, а большую часть решаемой задачи запрограммировать на языке высокого уровня. Например, в матричных операциях часто встречается вычисление скалярного произведения, которое для повышения производительности следует запрограммировать на языке ассемблера. Кроме того, язык ассемблера незаменим при разработке программ для многочисленных специализированных ЭВМ.

Основу книги составляют материалы лекций, прочитанных автором в Рязанском радиотехническом институте для студентов специальности «Электронные вычислительные машины» и на курсах повышения квалификации инженерно-технических работников.

Считаю приятным долгом выразить благодарность рецензенту доктору техн. наук, профессору Б. М. Кагану и редактору канд. техн. наук, доценту В. В. Сташину, замечания которых способствовали улучшению содержания книги. Заранее благодарю всех читателей, которые сочтут возможным высказать свои предложения и замечания.

Автор

ГЛАВА 1

МИКРОПРОЦЕССОР К1810ВМ86 — НОВЫЙ ЭТАП РАЗВИТИЯ МИКРОСХЕМОТЕХНИКИ

1.1. Общая характеристика микропроцессора

Понятие архитектуры процессора образуется такими атрибутами, как структура памяти, спецификация внутренних регистров, система команд и интерфейс. Ниже кратко описаны указанные характеристики микропроцессора (МП) К1810ВМ86 (далее используется сокращенное обозначение К1810).

Структура памяти. Адресные пространства памяти и ввода-вывода МП К1810 рассматриваются единообразно и в совокупности образуют структуру памяти. Программы и данные находятся в пространстве памяти, а порты ввода-вывода периферийных устройств размещаются в пространстве ввода-вывода, хотя допускается организация ввода-вывода, отображенного на память.

Потенциально адресуемая память представляет собой область из 1М байт. Два любых смежных байта образуют слово, причем адресом слова считается адрес младшего байта, а старший байт хранится по большему адресу.

Физический адрес памяти содержит 20 бит, однако МП К1810 рассчитан на 16-битную арифметику, и все обрабатываемые им адресные величины должны иметь длину 16 бит. Следовательно, для формирования физических адресов памяти необходим специальный механизм сегментации памяти. Пространство памяти состоит из областей (сегментов), имеющих размер 64К байт. Полный начальный или базовый адрес каждого сегмента содержит 20 бит и кратен 16, т. е. его младшие 4 бита содержат нули, которые можно не хранить. В любой момент времени МП может обращаться к ячейкам четырех сегментов, которые называются текущими сегментами кода (программы), данных, стека и дополнительного сегмента (экстрасегмента). Каждый из сегментов идентифицируется начальным адресом, находящимся в соответствующем 16-битном внутреннем указателе (сегментном регистре).

Команды могут обращаться к байтам и словам в пределах сегмента, используя 16-битный внутрисегментный адрес, называемый смещением, а также эффективным (исполнительным) адресом. Физический адрес памяти образуется суммированием смещения и сегментного адреса, сдвинутого на 4 бита влево. Таким образом, полный адресный объект МП К1810 представляется в виде сегмент:смещение или база:смещение.

Адресное пространство ввода-вывода МП К1810 состоит из 64К портов. Порты адресуются аналогично байтам и словам памяти, но сегментные регистры для адресации не привлекаются — можно считать, что все порты ввода-вывода находятся в одном и том же сегменте. Первые 256 портов ввода и 256 портов вывода адресуются непосредственно (адрес содержится в команде), а все 64К портов адресуются косвенно, т. е. с использованием регистров.

Спецификация регистров. Микропроцессор имеет 14 16-битных регистров, объединяемых в три группы. Первую из них образуют четыре регистра общего назначения (РОН), которые могут хранить любые данные и которыми программист может распоряжаться по своему усмотрению. Их можно адресовать как 8- и 16-битные регистры, что обеспечивает простую обработку байт и слов. Регистры двух остальных групп можно адресовать только как 16-битные.

Вторая группа представлена двумя указательными и двумя индексными регистрами, которые адресуют данные в пределах сегмента. Обычно они содержат внутрисегментные адреса (смещения) и предназначены для формирования эффективных адресов в соответствии с указанным в команде режимом адресации. Указательные регистры обеспечивают удобный доступ к данным в текущем сегменте стека, а индексные — в текущем сегменте данных.

Третью группу образуют четыре сегментных регистра, идентифицирующих начальные адреса четырех сегментов. Выборка команд программы осуществляется только из текущего сегмента кода с использованием смещения, находящегося в программном счетчике, называемом также указателем команды. Сегмент, к которому производится обращение за данными, либо принимается по умолчанию, либо указывается специальным однобайтным префиксом, находящимся в программе перед командой. Допускается динами-

ческое перемещение программ с соответствующим изменением содержимого сегментных регистров при условии, естественно, что сама программа не модифицирует их.

Регистр флажков (признаков) содержит девять флажков. Пять арифметических флажков соответствуют флажкам МП КР580ИК80 (далее используется сокращенное обозначение К580) и отражают особенности результатов операций. В МП К1810 введен также флажок переполнения, который упрощает программирование вычислительных алгоритмов. Три флажка управляют действиями МП: флажок направления определяет направление сканирования цепочек в сторону увеличения или уменьшения адресов; флажок прерываний разрешает или запрещает восприятие внешних маскируемых прерываний и флажок прослеживания (трассировки) переводит МП в одношаговый (покомандный) режим работы, удобный при отладке программ.

Система команд и режимы адресации МП К1810 намного разнообразнее, чем у МП К580. В нее включены почти все команды МП К580, но редко используемые команды, например условные вызовы и возвраты, отсутствуют. Большинство команд оперирует байтами и словами.

Микропроцессор К1810 имеет организацию регистр — память, которая подразумевает, что двухоперандные команды имеют типы регистр — регистр, регистр — память и память — регистр, а команды типа память — память отсутствуют (за исключением команды передачи цепочки [1] байт или слов). Результат операции в МП К1810 может быть помещен на место первого или второго операнда.

В МП имеются все основные режимы адресации, характерные для современных процессоров и ориентированные на эффективную реализацию языков высокого уровня, — прямая (абсолютная), регистровая, косвенная, непосредственная, базовая, индексная и их некоторые модификации.

Команды по функциональному признаку разделяются на несколько групп. Ниже дается краткая характеристика команд каждой группы.

Команды передач данных осуществляют стандартные действия пересылок данных регистр — регистр, регистр — память, память — регистр, а также включения в стек и извлечения из стека. В эту же группу входят команды ввода-вывода и табличного преобразования. Имеются также команды обмена типа регистр—регистр и регистр—память и пересылок адресных объектов (сегмент:смещение), которые позволяют программисту управлять механизмом адресации микропроцессора.

Арифметические команды представлены набором операций сложения, вычитания, умножения и деления, операндами которых могут быть 8-, 16-битные знаковые и беззнаковые целые числа. Имеются команды коррекции, обеспечивающие выполнение сложения и вычитания упакованных двоично-кодированных десятичных чисел (BCD-числа), а также выполнение всех арифметических операций над неупакованными десятичными числами, представленными в символьном коде (ASCII-числа).

Кроме обычных команд логических операций дизъюнкции, конъюнкции и исключающего ИЛИ (сложения по модулю 2), имеется команда неразрушающей проверки. Она выполняет поразрядную конъюнкцию операндов, модифицирует состояния флажков в соответствии с получающимся результатом, но сам результат нигде не запоминает.

Сдвиги представлены командами логических и арифметических сдвигов влево и вправо, а также циклических сдвигов (ротаций) влево и вправо. Сдвиг можно осуществить как на 1 бит (так называемый статический сдвиг), так и на произвольное число бит, определяемое содержимым регистра (так называемый динамический сдвиг).

Команды операций с цепочками манипулируют отдельными элементами цепочек данных. Обработка цепочек традиционно реализуется программными циклами, в которых расходуется значительное время на служебные действия (коррекция указателей, декремент счетчика цикла и проверка достижения им нуля). В МП К1810 имеется несколько команд, которые значительно сокращают время выполнения цепочечных операций.

Команды передачи управления — условные и безусловные переходы, вызовы подпрограмм и возвраты из подпрограмм — классифицируются на внутрисегментные, передающие управление в текущем сегменте кода путем модификации содержимого только программного счетчика, и межсегментные, которые передают управление в произвольный сегмент, определяя новое содержимое программного

[1] В литературе обычно используется термин «строка». Введение термина «цепочка» объясняется тем, что здесь речь идет о произвольной последовательности байт или слов, а «строка» обычно ассоциируется с символьной строкой. Кроме того, термин «строка» применяется во многих смыслах, например строка развертки, строка на экране дисплея, строка матрицы и др.

счетчика и сегментного регистра кода. Оба типа команд могут быть прямыми (адрес перехода содержится в команде) и косвенными (адрес перехода находится в адресуемом командой регистре или ячейке памяти). Команды возврата допускают коррекцию указателя стека с целью удаления из стека параметров, включенных в него перед вызовом подпрограммы. В командах условных переходов предусматривается учет всех отношений между знаковыми и беззнаковыми числами (больше, не меньше, равно, не равно, не больше, меньше).

Команда программного прерывания включает в стек содержимое регистра флажков, программного счетчика и сегментного регистра кода, а затем загружает в программный счетчик и сегментный регистр кода новые адреса, определяемые содержащимся в команде однобайтным вектором. Эта двухбайтная команда предназначена для вызовов подпрограмм, начальные адреса которых находятся в первых 1024 байтах памяти. Возможен вызов одной из 256 подпрограмм. В микропроцессоре имеется также команда прерывания при наличии переполнения, передающая управление через фиксированный вектор.

Команды управления МП осуществляют установку и сброс некоторых флажков, переводят МП в режимы ожидания или останова, а также формируют специальный сигнал блокировки шины. Этот сигнал предназначен для управляемого доступа к разделенной системной шине в мультипроцессорных системах. Имеется специальная команда, упрощающая введение в систему сопроцессора (вспомогательного процессора).

Интерфейс. Взаимодействие МП с «внешним миром» осуществляется через мультиплексную шину адреса/данных и несколько управляющих сигналов. Отличительной чертой МП является возможность работы в двух режимах (конфигурациях): минимальном и максимальном, определяемых уровнем сигнала на одном из входных контактов. В зависимости от режима определяются функции восьми управляющих сигналов. Минимальный режим рассчитан на сравнительно простые однопроцессорные системы; в нем управляющие сигналы системной шины генерирует сам МП. В максимальном режиме, который предназначен для сложных мультипроцессорных систем, формирование управляющих сигналов системной шины осуществляется специальной микросхемой контроллера шины.

1.2. Конструктивное оформление МП и назначение выводов корпуса

Микросхема К1810ВМ86 представляет собой 16-битный центральный процессор, выполненный по высококачественной *n*МОП-технологии. На кристалле с размерами 5,5×5,5 мм размещено около 29 000 транзисторов. Микропроцессор имеет напряжение питания $+(5\pm0{,}25)$ В, синхронизируется однофазными сигналами с частотой 2—5 МГц и потребляет максимальную мощность 1,75 Вт. Все входные и выходные сигналы микропроцессора ТТЛ-совместимы. Емкостная нагрузка на любой вывод не должна превышать 100 пФ. Микропроцессор сохраняет работоспособность в температурном диапазоне 10—70 °С. Конструктивно МП оформлен в 40-контактном корпусе «Монтаж 2—40» с двусторонним расположением выводов. Разводка выводов МП приведена на рис. 1.1.

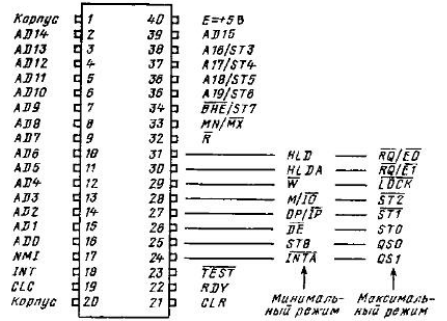


Рис. 1.1. Разводка выводов МП

Уровень напряжения на входном выводе MN/$\overline{\text{MX}}$ определяет режим работы МП. Если этот вывод подключен к источнику +5 В, МП работает в минимальном режиме, а если на землю — в максимальном. В зависимости от установленного режима интерпретируются функции восьми выходных сигналов на выводах 24—31.

Рассмотрим функциональное назначение сигналов в минимальном режиме. Чтобы лучше разобраться в функциях многочисленных сигналов МП, целесообразно обратиться к рис. 1.2, на котором показано группирование сигналов по общему функциональному назначению.

Системную шину адреса/данных/состояния образуют 20 линий. **AD15—AD0** — мультиплексная шина адреса/данных.

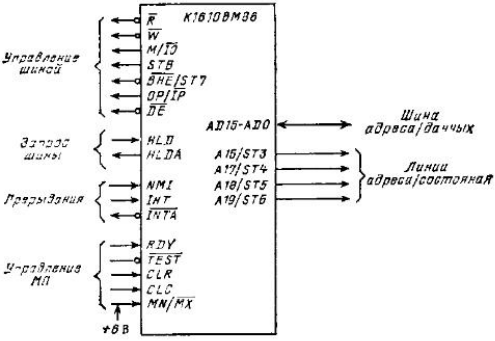


Рис. 1.2. Группирование сигналов МП

По 16 двунаправленным линиям шины адреса/данных с разделением во времени передаются младшие 16 бит адреса памяти и полные адреса ввода-вывода. В первом такте машинного цикла, называемого также циклом шины, МП выдает на линии AD15—AD0 адрес, а затем принимает или передает данные. Следовательно, для фиксации адреса на весь машинный цикл необходим регистр-защелка. Выходные каскады линий AD15—AD0 при подтверждении запроса шины переводятся в высокоимпедансное состояние (в литературе встречаются также термины «состояние высокого сопротивления» и даже «третье состояние»).

A19/ST6—A16/ST3 — мультиплексные выходные сигналы адреса/состояния. В первом такте на эти линии выводятся значения старших 4 бит адреса памяти, а при адресации ввода-вывода действуют сигналы низкого уровня.

В остальных тактах машинного цикла МП выдает на эти линии сигналы состояния ST6—ST3.

Значения сигналов ST4—ST3 идентифицируют сегментный регистр, участвующий в формировании физического адреса памяти. Другими словами, эти сигналы определяют, к какому сегменту памяти производится обращение в текущем машинном цикле. Принято следующее кодирование сегментов: 00 — дополнительный сегмент, 01 — сегмент кода или никакой, например при вводе-выводе, 10 — сегмент стека, 11 — сегмент данных.

Сигнал ST5 соответствует состоянию флажка прерывания: 0 — прерывания запрещены, 1 — прерывания разрешены. Сигнал ST6 пока не используется и всегда имеет низкий уровень.

При подтверждении запроса шины выходные буфера сигналов A19/ST6—A16/ST3 переводятся в высокоимпедансное состояние.

Семь выходных сигналов предназначены для управления шиной; они определяют операцию МП в текущем машинном цикле и использование сигналов на линиях адреса/данных/состояния. Все эти сигналы, за исключением сигнала STB, имеют тристабильные выходные буфера, которые при подтверждении запроса шины переводятся в высокоимпедансное состояние.

$\overline{R}$ (или **$\overline{RD}$**) — считывание, или чтение. Этот сигнал идентифицирует выполнение цикла считывания из памяти или устройств ввода-вывода (УВВ) в зависимости от значения сигнала M/$\overline{IO}$ (см. ниже). Активный (низкий) уровень сигнала $\overline{R}$ определяет направление передачи по шине AD в микропроцессор.

$\overline{W}$ (или **$\overline{WR}$**) — запись. Сигнал на линии $\overline{W}$ определяет выполнение цикла шины, в котором осуществляется запись в память или УВВ (в зависимости от уровня сигнала M/$\overline{IO}$). Активный (низкий) уровень сигнала $\overline{W}$ показывает, что шина AD содержит записываемые (выводимые из МП) данные.

M/$\overline{IO}$ — память — ввод-вывод. Этот сигнал разделяет адресные пространства памяти и ввода-вывода: если M/$\overline{IO}$=1, МП обращается к памяти, а если M/$\overline{IO}$=0, — к портам ввода-вывода. Сигнал M/$\overline{IO}$ действует на протяжении практически всего машинного цикла.

STB (или **ALE**) — строб адреса. Сигнал STB действует

в первом такте каждого машинного цикла и предназначен для загрузки адреса с линий шины AD в регистр адреса. Другими словами, он осуществляет демультиплексирование шины адреса/данных.

$\overline{\text{BHE}}$/ST7 — разрешение старшего байта/состояние. Данный сигнал выполняет две функции. В первом такте машинного цикла низкий уровень этого сигнала означает, что 8-битные данные передаются по старшей половине AD15—AD8 шины адреса/данных. Устройства с байтной организацией, подключенные к старшей половине шины, обычно используют сигнал $\overline{\text{BHE}}$/ST7 как условие их выбора. Наличие этого сигнала позволяет подключать одну часть устройств с байтной организацией к младшей половине шины AD, а другую часть — к старшей.

В остальных тактах на линию $\overline{\text{BHE}}$/ST7 выдается резервный сигнал состояния ST7, не имеющий определенного значения.

DT/$\overline{\text{R}}$ (или **DT/$\overline{\text{R}}$**) — передача/прием данных. Этот сигнал определяет направление передачи по шине AD: если DT/$\overline{\text{R}}$ = 1, выполняется запись данных из МП во внешнюю подсистему, а если DT/$\overline{\text{R}}$ = 0, МП принимает данные из внешней подсистемы. Сигнал предназначен для управления шинными формирователями (драйверами), а его временна́я диаграмма аналогична диаграмме сигнала M/$\overline{\text{IO}}$.

$\overline{\text{DE}}$ (или **$\overline{\text{DEN}}$**) — разрешение передачи данных. Сигнал $\overline{\text{DE}}$ представляет собой строб данных и применяется для управления шинными формирователями в системе, работающей в минимальном режиме.

Два сигнала предназначены для управления запросом (подтверждением запроса) шины, когда ее запрашивает внешняя подсистема, например контроллер прямого доступа к памяти или другой процессор в мультипроцессорной системе.

HLD (или **HOLD**) — запрос (захват) шины. Это входной сигнал, идентифицирующий высоким уровнем сигнала запрос шины внешней подсистемой.

HLDA — подтверждение запроса. Этим выходным сигналом МП подтверждает восприятие запроса шины, приостанов вычислительного процесса и перевод выходных буферов шины AD и некоторых управляющих сигналов в высокоимпедансное состояние. После этого шиной может распоряжаться подсистема, инициировавшая запрос.

Микропроцессор имеет три сигнала, связанные с прерываниями.

NMI — немаскируемое прерывание. Входной сигнал NMI распознается микропроцессором всегда по завершению текущей команды независимо от того, разрешены прерывания или нет. Реагируя на сигнал NMI, МП переходит к соответствующей подпрограмме обслуживания через фиксированный адрес в таблице векторов прерываний, находящейся в системной памяти. Немаскируемые прерывания предназначены для сигнализации о некоторых критических событиях, например аварийном отключении сетевого питания, ошибке в памяти и др.

INT — прерывание (или запрос прерывания). Это входной сигнал, который фиксируется во внутреннем триггере. Микропроцессор проверяет состояние этого триггера в последнем такте каждой команды. При наличии запроса прерывания МП формирует цикл подтверждения прерывания, в результате которого он вводит по шине AD однобайтный указатель, идентифицирующий прерывающее устройство. Далее в соответствии со значением указателя МП переходит через таблицу векторов прерываний к нужной подпрограмме обслуживания прерывания. Восприятием или игнорированием сигнала INT можно управлять программно с помощью флажка разрешения прерываний. Обычно на вход INT подается выход прерывания программируемого контроллера прерываний.

$\overline{\text{INTA}}$ — подтверждение прерывания. Это выходной сигнал, который выполняет функцию считывания указателя в цикле подтверждения прерывания по входу INT.

Пять входных сигналов предназначены для управления работой МП.

RDY — готовность. Этот сигнал дает возможность приостановить действия МП (низким уровнем на входе RDY). Обычно сигнал RDY используется в интерфейсах устройств ввода-вывода и памяти, быстродействия которых недостаточно для синхронной работы с МП. Переход сигнала RDY на высокий уровень свидетельствует о том, что адресуемое устройство закончило передачу или прием данных.

$\overline{\text{TEST}}$ — проверка. Данный сигнал используется вместе с командой ожидания WAIT. Выполняя ее, МП проверяет уровень сигнала $\overline{\text{TEST}}$. Если $\overline{\text{TEST}}$ = 0, МП переходит к выполнению следующей по порядку команды, а если $\overline{\text{TEST}}$ = 1, МП вводит холостые состояния и периодически

проверяет уровень сигнала $\overline{TEST}$ с интервалом в пять тактов синхронизации. Таким образом, команда WAIT и сигнал TEST обеспечивают синхронизацию действий МП с внешними событиями.

CLR — сброс. Сигнал CLR переводит МП в известное начальное состояние. При активном (высоком) уровне сигнала действия МП прекращаются. Минимальная продолжительность сигнала CLR составляет при первом включении МП 50 мкс, а при повторном запуске — четыре такта синхронизации. При снятии сигнала CLR работа МП возобновляется из начального состояния.

CLC — синхронизация (или тактирование). Периодические сигналы на входе CLC от внешнего генератора предназначены для синхронизации всех действий МП.

MN/$\overline{MX}$ — минимальный/максимальный. Как уже указывалось, уровень сигнала на этом входе определяет режим работы (или конфигурацию) МП.

В максимальной режиме (MN/$\overline{MX}$=0) изменяется интерпретация восьми управляющих сигналов.

$\overline{ST2}$—$\overline{ST0}$ — состояние. В максимальном режиме линии M/$\overline{IO}$, OP/$\overline{IP}$ и $\overline{DE}$ превращаются в линии состояния, которые в каждом машинном цикле содержат информацию, определяющую тип машинного цикла в соответствии с табл. 1.1. Сигналы состояния подаются в контроллер шины,

Таблица 1.1. **Идентификация типа цикла шины**

$\overline{ST2}$	$\overline{ST1}$	$\overline{ST0}$	Тип цикла шины
0	0	0	Подтверждение прерывания
0	0	1	Считывание ввода-вывода
0	1	0	Запись ввода-вывода
0	1	1	Останов
1	0	0	Выборка команды
1	0	1	Считывание из памяти
1	1	0	Запись в память
1	1	1	Пассивный (цикла шины нет)

который формирует расширенный набор управляющих сигналов шины. Всякое изменение сигналов $\overline{ST2}$—$\overline{ST0}$ определяет инициирование следующего цикла шины, а переход их в пассивное состояние указывает на то, что МП не запрашивает цикл шины.

$\overline{LOCK}$ — блокировка (занятость) шины. Активный сигнал $\overline{LOCK}$ на этом выходе информирует внешние подсистемы о том, что они не должны пытаться запрашивать шину. Он формируется по однобайтной команде LOCK, называемой префиксом блокировки шины, и поддерживается активным до окончания следующей за префиксом команды. При подтверждении запроса шины выходной буфер сигнала $\overline{LOCK}$ переводится в высокоимпедансное состояние.

Префикс LOCK не влияет на прерывания. Если в состоянии блокировки шину запрашивает внешняя подсистема по линиям $\overline{RQ}/\overline{E1}=\overline{RQ}/\overline{E0}$ (см. ниже), МП фиксирует запрос, но не подтверждает его до завершения выполнения «заблокированной» команды. Префикс LOCK можно использовать и в минимальном режиме, когда сигнал $\overline{LOCK}$ отсутствует. В этом случае префикс LOCK задерживает генерирование сигнала подтверждения HLDA на запрос шины HLD до завершения выполнения команды.

Программисты обычно используют префикс LOCK в операциях над семафорами, индентифицирующими состояние разделяемых ресурсов (см. § 1.7).

QS1—QS0 — состояние очереди. Уровни этих выходных сигналов идентифицируют состояние внутренней 6-байтной очереди команд МП (см. ниже) в соответствии с табл. 1.2. Они действуют в течение такта синхронизации после выполнения операции над очередью.

Таблица 1.2. **Идентификация состояния очереди**

QS1	QS0	Операция над очередью
0	0	Операции нет, в последнем такте из очереди ничего не выбиралось (значение по умолчанию)
0	1	Из очереди выбран последующий байт команды
1	0	Очередь реинициализирована после передачи управления
1	1	Из очереди выбран последующий байт команды

Сигналы QS1—QS0 предназначены для сопроцессора, который воспринимает команды и операнды с помощью команды ESC (см. § 1.7). Сопроцессор контролирует шину и фиксирует момент, когда из программной памяти выбирается команда ESC, а затем следит за очередью команд и определяет, когда фактически выполняется эта команда.

$\overline{RQ}/\overline{E1}$—$\overline{RQ}/\overline{E0}$ (или $\overline{RQ}/\overline{GT1}$—$\overline{RQ}/\overline{GT0}$) — запрос/разрешение (подтверждение). Сигналы на двунаправленных линиях $\overline{RQ}/\overline{E1}$ и $\overline{RQ}/\overline{E0}$ используются внешними подсистемами для управления шиной. Сигналы на этих линиях независимы, но линия $\overline{RQ}/\overline{E0}$ имеет более высокий приоритет, чем линия $\overline{RQ}/\overline{E1}$. Но если на линии $\overline{RQ}/\overline{E0}$ появляется запрос в то время, когда МП обрабатывает предшествующий запрос $\overline{RQ}/\overline{E1}$, то второй запрос не подтверждается до освобождения шины по линии $\overline{RQ}/\overline{E1}$.

1.3. Структурная схема микропроцессора

Появление программируемых микросхем с большой степенью интеграции, которые содержат на кристалле десятки тысяч транзисторов, организованных в очень сложные структуры, несколько сместило акценты в уровне их описания. Традиционные структурные схемы становятся чрезвычайно громоздкими и почти не нужны пользователям. Главными здесь становятся те внутренние ресурсы, которыми пользователь может распоряжаться на уровне команд, а также особенности внешнего интерфейса. В связи с этим в настоящем параграфе рассмотрена довольно общая структурная схема МП К1810, которая необходима для понимания принципов и особенностей его функционирования.

Выполненне программы в ЭВМ представляет собой циклическую последовательность приведенных ниже действий, образующих цикл команды:

1) выборка команды из памяти и формирование адреса следующей по порядку команды;

2) считываиие операнда из памяти, если это требуется по смыслу команды;

3) собственно выполненне команды (операции);

4) запись результата в память, если это указано в команде, и переход к новому циклу команды.

Обычно в МП эти действия выполняются последовательно во времени. В МП К1810 основные этапы сохранены, но они распределены внутри МП по двум сравнительно независимым устройствам. *Операционное устройство* выполняет команды (в смысле реализации операций), а *устройство шинного интерфейса* (или просто *шинный интерфейс*) выбирает команды, считывает операнды и записывает результаты. Оба устройства могут работать параллельно и в

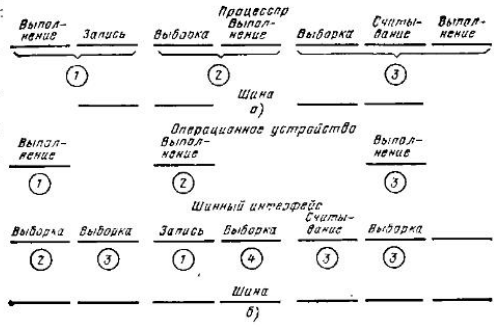


Рис. 1.3. Совмещение выборки и выполнения команд:
а — микропроцессор К580; *б* — микропроцессор К1810

большинстве случаев обеспечивают значительное совмещение выборки и выполнения команд. В результате этого время выборки команды как бы «исчезает» из цикла команды, так как операционное устройство выполняет команды, уже выбранные из памяти шинным интерфейсом. На рис. 1.3 показан эффект опережающей выборки команд. Для простоты показаны команды длиной в одно слово, хотя МП К1810 имеет команды длиной до трех слов. Как видно из рисунка, совмещение действий сокращает время выполнения трех команд и обеспечивает опережающую выборку двух дополнительных команд.

Операционное устройство (рис. 1.4) содержит группу общих регистров, арифметико-логическое устройство (АЛУ), основу которого составляет комбинационный 16-битный сумматор с последовательно-параллельным переносом, регистр флажков и несколько регистров для временного хранения операндов и результата операции. Оно выполняет команды, обменивается данными и адресами с шинным интерфейсом, оперирует общими регистрами и флажками. В его составе имеется блок микропрограммного управления, который дешифрирует команды и формирует необходимые управляющие сигналы. Операционное устройство изолировано от внешней шины, за исключением нескольких внешних сигналов.

Шинный интерфейс выполняет для операционного устройства все операции обмена. Данные передаются между МП и памятью или портами ввода-вывода по запросам операционного устройства. Когда операционное устройство занято выполнением команды, шинный интерфейс самостоятельно инициирует опережающую выборку из памяти очередных команд. Команды хранятся во внутренней регистровой памяти, называемой *очередью (буфером) команд*. Очередь команд выполняет по существу функции регистра команды процессора. Длина очереди составляет 6 байт. Очередь команд работает по принципу FIFO («первый пришел, первый ушел»), который сохраняет на выходе порядок поступления команд.

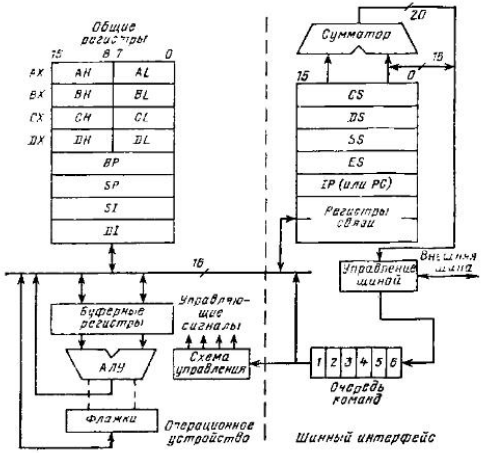


Рис. 1.4. Структурная схема МП

Шинный интерфейс инициирует выборку из памяти следующего командного слова, когда в очереди оказываются два свободных («пустых») байта.

В большинстве случаев очередь команд содержит минимум 1 байт потока команд, и операционное устройство не ожидает выборки команды. Конечно, очередь обеспечивает положительный эффект при естественном порядке выполнения команд. Когда операционное устройство исполняет команду передачи управления, шинный интерфейс сбрасывает очередь, выбирает команду по новому адресу, передает ее в операционное устройство, а затем начинает заполнение (реинициализацию) очереди из следующих ячеек. Эти действия инициируются в условных и безусловных переходах, вызовах подпрограмм, возвратах из подпрограмм и при обработке прерываний. Шинный интерфейс приостанавливает выборку команд, когда операционное устройство запрашивает операцию считывания или записи в память или порт ввода-вывода.

В состав шинного интерфейса входят несколько регистров и сумматор, которые формируют 20-битный физический адрес памяти из двух 16-битных логических адресов — сегмента (базы) и смещения (подробнее см. § 1.5).

При готовности операционного устройства выполнять команду оно считывает из очереди байт, а затем выполняет предписанную командой операцию. При многобайтных командах из очереди считываются и другие байты команды. Когда операционное устройство готово считать командный байт, а очередь команд пуста, оно ожидает выборки командного слова из памяти программ, которую инициирует шинный интерфейс (такие ситуации исключительно редки). Если команда требует обращения к памяти или порту ввода-вывода, операционное устройство запрашивает шинный интерфейс на выполнение необходимого цикла шины. Когда шинный интерфейс не занят выборкой команды, он удовлетворяет запрос немедленно; в противном случае операционное устройство ожидает завершения текущего цикла шины.

1.4. Модель МП для программиста

Модель МП для программиста, или для краткости — программная модель, составлена из регистров, которые доступны пользователю на уровне команд.

В программной модели МП К1810, представленной на рис. 1.5, имеется 14 16-битных регистров (в скобках указаны соответствующие регистры МП К580). Для удобства рассмотрения регистры разделены на три группы [1].

Группа регистров общего назначения, называемых также регистрами данных, образована регистрами AX, BX, CX и DX. Отличительной особенностью этих регистров является то, что допускается раздельно адресовать их старшую (H) и младшую (L) половины, т. е. каждый из них можно использовать как 16-битный регистр или два 8-битных

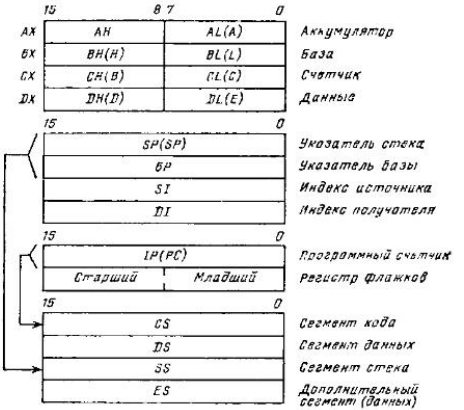


Рис. 1.5. Модель МП для программиста

регистра. Это обеспечивает простую обработку 8- и 16-битных данных (байт и слов). Остальные регистры МП можно использовать только как 16-битные.

Регистры AX, BX, CX и DX предназначены в основном для хранения данных, они находятся в полном распоряжении программиста и единообразно участвуют в арифметических и логических операциях. Однако имеется много команд, которые специализируют регистры данных на определенные функции, что отражено в их названиях (табл. 1.3) [1].

Программы получаются наиболее компактными, если в командах арифметических и логических операций и в командах пересылки данных использовать аккумулятор AX.

Таблица 1.3. **Специальные функции регистров микропроцессора**

Регистр	Назначение	Функция
AX	Аккумулятор	Умножение, деление и ввод-вывод слов
AL	Аккумулятор (младший)	Умножение, деление и ввод-вывод байт; преобразование; десятичная арифметика
AH	Аккумулятор (старший)	Умножение и деление байт
BX	База	Базовый регистр; преобразование
CX	Счетчик	Операции с цепочками; циклы
CL	Счетчик (младший)	Динамические сдвиги и ротации
DX	Данные	Умножение и деление слов; косвенный ввод-вывод
SP	Указатель стека	Стековые операции
BP	Указатель базы	Базовый регистр
SI	Индекс источника	Операции с цепочками; индексный регистр
DI	Индекс получателя	Операции с цепочками; индексный регистр

Регистр AL в основном соответствует аккумулятору А микропроцессора К580, а регистр AH называется *расширением* регистра AL. Регистр BX почти эквивалентен основному указателю памяти МП К580 — регистровой паре HL и участвует в командах как источник базового адреса.

Группа указательных и индексных регистров представлена адресными регистрами SP, BP, SI и DI. Они предназначены для хранения 16-битных адресов (внутрисегментных смещений) и обеспечивают при этом косвенную адресацию и динамическое вычисление эффективного адреса памяти.

Гибкость вычислений достигается тем, что эти регистры могут участвовать в арифметических и логических операциях так же, как и регистры предыдущей группы. Поэтому далее регистры данных и регистры настоящей группы называются общими регистрами.

Указательные регистры SP и BP предназначены для упрощения доступа к данным, находящимся в текущем сегменте стека, а не в сегменте данных. Индексные регистры SI и DI содержат смещения, которые по умолчанию относятся к текущему сегменту данных.

В некоторых командах адресные регистры специализированы, что отражено в их названиях (табл. 1.3).

Специализация общих регистров приводит к тому, что они по существу не являются регистрами общего назначе-

ния. Довольно нерегулярная структура внутренних регистров МП требует, чтобы программист или компилятор языка высокого уровня тщательно распределял и следил за использованием конкретных регистров. По существу МП К1810 имеет одноаккумуляторную архитектуру, напоминающую архитектуру его предшественника МП К580. Однако неявное использование некоторых регистров в операциях и режимах адресации позволяет закодировать команды в более коротком формате по сравнению с форматом команд для идеальных общих регистров.

Наличие в программной модели четырех сегментных регистров объясняется способом адресации памяти. Хотя МП имеет 20-битную шину физического адреса памяти, он оперирует 16-битными логическими адресами, состоящими из базового адреса сегмента и внутрисегментного смещения. Внутреннее устройство преобразования адресов превращает два логических адреса в 20-битный физический адрес.

Пространство памяти 1М байт разделено на логические сегменты емкостью 64К байт. Выполняя программу, МП может одновременно обращаться к четырем сегментам. Их базовые (начальные) адреса и содержатся в сегментных регистрах CS (кода), DS (данных), SS (стека) и ES (дополнительных данных).

Регистр CS указывает на текущий сегмент кода (программы), откуда выбираются команды. Регистр SS адресует текущий сегмент стека, в этом сегменте реализуются все стековые операции. Регистр DS указывает на текущий сегмент данных, в котором содержатся переменные. Наконец, регистр ES определяет текущий дополнительный сегмент, который обычно используется для хранения данных.

В большинстве команд содержится только одна часть логического адреса — внутрисегментное смещение. Оно может быть вычислено в соответствии с указанным режимом адресации, может находиться в команде или содержаться в общем регистре. Физический 20-битный адрес формируется путем суммирования смещения и содержимого одного из сегментных регистров (подробнее см. § 1.5). В системах, общая емкость памяти которых составляет не более 64К байт, сегментные регистры можно сбросить, и они не будут влиять на физические адреса.

Сегментные регистры доступны программе, и их содержимым манипулирует несколько команд. Естественно, при программировании необходимо очень внимательно следить за упорядоченным использованием сегментных регистров.

Отметим, что размер сегментов фиксирован, а средства защиты сегментных регистров отсутствуют. Обычно в системах с сегментной организацией памяти манипуляции сегментными регистрами осуществляются операционной системой в так называемом системном режиме работы, и пользователю (прикладной программе) сегментные регистры недоступны. В этом случае команды, оперирующие сегментными регистрами, называются *привилегированными*.

Последнюю группу регистров МП образуют 16-битные указатель команд IP (аналог стандартного программного счетчика PC; именно это более привычное сокращение используется в дальнейшем) и регистр флажков (или признаков). Шинный интерфейс осуществляет модификацию PC таким образом, что он содержит смещение следующей команды от начала текущего сегмента кода, т. е. указывает на следующую по порядку команду. При обычной работе PC содержит смещение следующей команды, которую будет выбирать шинный интерфейс из памяти программ. Оно не совпадает со смещением той очередной команды, которую будет выполнять операционное устройство. Поэтому при запоминании содержимого PC в стеке, например при вызове подпрограммы, оно автоматически корректируется, чтобы адресовать следующую команду, которая будет выполняться. Непосредственный доступ к PC имеют команды передачи управления.

Формат 16-битного регистра флажков приведен на рис 1.6. Его младший байт полностью соответствует регистру флажков МП К580, а старший байт содержит четыре флажка, которых нет в МП К580.

Старший байт								Младший байт							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	CF

X – состояние не определено

Рис. 1.6. Формат регистра флажков

Шесть арифметических флажков фиксируют определенные свойства (признаки) результата арифметической или логической операции. Группа команд условных переходов позволяет изменить ход программы в зависимости от состояний флажков. Различные команды влияют на флажки по-разному, но, в общем, флажки отражают следующие условия:

1) флажок вспомогательного переноса **AF** фиксирует перенос (заем) из младшей тетрады в старшую 8- или 16-битного результата. Он необходим только для команд десятичной арифметики;

2) флажок **CF** фиксирует значение переноса (заема), возникающего при сложении (вычитании) байт или слов, а также значение выдвигаемого бита при сдвиге операнда;

3) флажок переполнения **OF** сигнализирует о потере старшего бита результата сложения или вычитания. Имеется специальная команда прерывания при переполнении, которая в данной ситуации генерирует программное прерывание;

4) флажок знака **SF** повторяет значение старшего бита результата, который при использовании дополнительного кода соответствует знаку числа;

5) флажок паритета **PF** (или четности) фиксирует наличие четного числа единиц в младших 8 бит результата операции. Этот флажок предназначен для контроля правильности передач данных;

6) флажок нуля **ZF** сигнализирует о получении нулевого результата операции.

Три дополнительных флажка предназначены для управления некоторыми действиями микропроцессора:

1) флажок направления **DF** определяет порядок сканирования цепочек в соответствующих командах: от меньших адресов к большим (DF=0) или наоборот (DF=1);

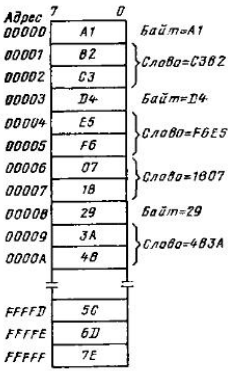
2) флажок прерывания **IF** определяет реакцию МП на запросы внешних прерываний по входу INT. Если IF=0, запросы прерываний игнорируются (говорят также, что прерывания запрещены или замаскированы), а если IF=1, микропроцессор распознает и соответственно реагирует на запрос прерывания. Отметим, что состояние флажка IF не влияет на восприятие внешних немаскируемых прерываний по входу NMI, а также внутренних (программных) прерываний;

3) установка в состояние 1 флажка прослеживания (трассировки) **TF** переводит МП в одношаговый (покомандный) режим работы, который применяется при отладке программ. В этом режиме МП автоматически генерирует внутреннее прерывание после выполнения каждой команды с переходом к соответствующей подпрограмме обработки, которая может, например, индицировать содержимое внутренних регистров.

Предусмотрены специальные команды, с помощью которых программист может задать требующееся ему состояние любого из этих флажков (кроме TF).

1.5. Организация памяти

В минимальном и максимальном режимах работы МП К1810 обеспечивает адресацию памяти емкостью 1М байт. Адресное пространство памяти, показанное на рис. 1.7, представляет собой одномерный массив *байт*, каждый из



←Рис. 1.7. Адресное пространство памяти

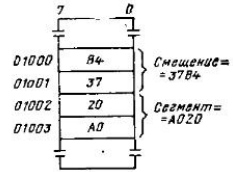


Рис. 1.8. Хранение в памяти полного указателя

которых имеет 20-битный физический адрес. Про такую память говорят, что она имеет *байтную организацию*. Минимальной адресуемой единицей памяти является байт, поэтому физический адрес, выдаваемый микропроцессором на шину адреса, считается адресом байта. Для записи адресов далее используется исключительно 16-ричная система счисления.

Любые два смежных байта в памяти образуют 16-битное *слово*. Младший байт слова имеет меньший адрес, а старший — больший. Такое соглашение было принято в микропроцессоре К580 и характерно для большинства современных мини- и микро-ЭВМ. Пользователь обычно не

замечает этого несколько неудобного соглашения, но о нем следует помнить при анализе действий шины и просмотре распечатки (дампа) содержимого памяти.

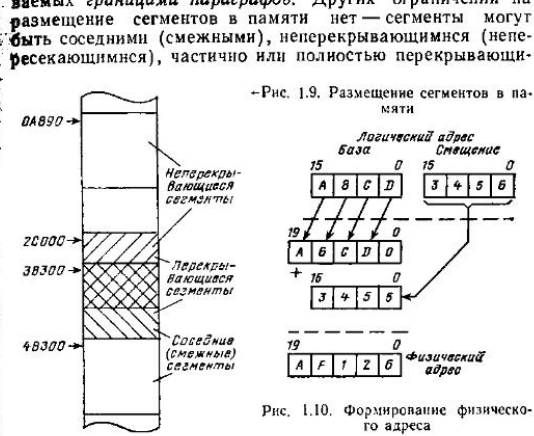
Адресом слова считается адрес его младшего байта. Таким образом, 20-битный адрес памяти можно рассматривать и как адрес байта и как адрес слова. Например, в соответствии с рис. 1.7 адрес 00000 может обозначать и байт с этим адресом, что условно записывается в виде ([00000]) = = A1, и слово с таким же адресом, что записывается в виде ([00000]) = B2A1. Круглые скобки здесь заменяют слово «содержимое», а квадратные подразумевают ячейку памяти, адрес которой находится в этих скобках.

Принцип *«младшее — по меньшему адресу»* сохраняется и для адресных объектов, называемых указателями и представляющих собой полные адреса памяти сегмент:смещение. Такие указатели применяются для адресации данных и команд, находящихся вне текущих сегментов. Слово с меньшим адресом содержит смещение, а с большим — базовый адрес сегмента. Каждое слово хранится обычным образом, т. е. байт с бо́льшим адресом содержит старшие 8 бит слова (рис. 1.8). Адресом всего указателя является адрес младшего байта смещения. Отметим, что полный указатель может адресовать любую ячейку памяти при различных комбинациях значений смещения и сегмента.

Команды, байты и слова данных можно свободно размещать по любому адресу байта, что экономит память за счет плотной упаковки программ. Однако целесообразно размещать слова в памяти по четным адресам, так как МП может передавать такие слова за один цикл шины. Слово с четным адресом называется *выравненным* на границе слова. Слова с нечетными адресами (невыравненные) также допустимы, но для их передачи требуются два цикла шины, что снижает производительность микропроцессора. Особенно важно иметь выравненные слова для операций со стеком, так как в них участвуют только слова. Следовательно, указатель стека SP всегда необходимо инициализировать на четный адрес. Отметим, что выравнивание команд почти не влияет на производительность МП, так как устройство шинного интерфейса выбирает их в очередь команд с опереженнем.

Сегментация. Программы «видят» пространство памяти 1М байт как группу сегментов, определяемых самой программой. *Сегмент* представляет собой логическую единицу памяти размером 64К байт. Он состоит из смежных ячеек

памяти и является независимой и отдельно адресуемой единицей памяти. Каждому сегменту программой назначается базовый (начальный) адрес, являющийся адресом его первого байта в адресном пространстве памяти. Все сегменты начинаются на 16-байтных границах памяти, называемых *границами параграфов*. Других ограничений на размещение сегментов в памяти нет — сегменты могут быть соседними (смежными), неперекрывающимися (непересекающимися), частично или полностью перекрывающи-



←Рис. 1.9. Размещение сегментов в памяти

Рис. 1.10. Формирование физического адреса

мися. Возможное размещение сегментов в пространстве памяти показано на рис. 1.9. Физическая ячейка памяти может принадлежать одному или нескольким сегментам.

Сегментные регистры CS, DS, SS и ES содержат начальные адреса (базы) четырех текущих сегментов: сегмента кода (программы), сегмента данных, сегмента стека и дополнительного сегмента данных соответственно. Для обращения к командам и данным, находящимся в других сегментах, необходимо изменять содержимое сегментных регистров.

В каждом применении сегменты определяются и используются по-разному. Текущие сегменты обеспечивают следующее рабочее пространство: 64К байт для кода (программы), 64К байт для стека и 128К байт для данных.

Если такого рабочего пространства достаточно, то сначала следует инициализировать сегментные регистры, а затем забыть о них. В программах, требующих большего адресного пространства, приходится модифицировать содержимое сегментных регистров. При этом необходимо тщательно следить за их определением и использованием.

Сегментная организация памяти рассчитана на модульные программы, так как во многих ситуациях сегментация обеспечивает определенные преимущества. Рассмотрим, например, редактирование текста с нескольких терминалов, работающих параллельно. Каждому терминалу отводится буферная область текста размером до 64К байт, адресуемая через сегментные регистры DS или ES. Всеми буферами управляет одна программа-редактор, которая изменяет содержимое сегментного регистра таким образом, чтобы он адресовал буфер терминала, требующего обслуживания.

Физические и логические адреса. Удобно считать, что каждая ячейка памяти имеет два адреса: физический и логический. *Физический адрес* представляет собой 20-битное значение в диапазоне от 0 до FFFFF, которое однозначно идентифицирует положение каждого байта в пространстве памяти 1М байт. Именно физический адрес выдается на шину адреса в начале каждого цикла шины, связанного с обращением к памяти.

Программы оперируют не физическими, а логическими адресами, что позволяет разрабатывать их без априорного знания того, как конкретная программа размещается в памяти. Кроме того, упрощается динамическое управление памятью, что имеет большое значение в мультипрограммной среде. Логический адрес состоит из двух 16-битных беззнаковых значений: базового (начального) адреса сегмента, который называется также просто базой или сегментом, и внутрисегментного адреса или смещения. Для любой ячейки памяти база идентифицирует первый байт содержащего ее сегмента, т.е. начало сегмента, а смещение определяет расстояние в байтах от начала сегмента до этой ячейки. Нулевое смещение имеет байт с наименьшим внутрисегментным адресом, а максимальное смещение равно FFFF.

Когда устройство шинного интерфейса обращается к памяти для выборки команды или считывания/записи данных, оно образует из логического адреса сегмент:смещение физический адрес. Для этого база сегмента сдвигается

Таблица 1.4. Источники логического адреса

Тип обращения к памяти	База (по умолчанию)	Вариант	Смещение
Выборка команды	CS	Нет	PC
Стековая операция	SS	Нет	SP
Переменная	DS	CS, SS, ES	EA
Цепочка-источник	DS	CS, SS, ES	SI
Цепочка-получатель	ES	Нет	DI
BP как базовый регистр	SS	CS, DS, ES	EA

влево на 4 бита и суммируется со смещением (рис. 1.10). Таким образом, принцип формирования физического адреса предполагает, что абсолютные адреса сегментов оканчиваются четырьмя двоичными нулями. Такие адреса называются границами сегментов, а четыре старшие 16-ричные цифры, т. е. содержимое сегментного регистра, — номерами параграфов. Следовательно, термины «база», «сегмент», «номер параграфа» являются по существу синонимами. Отметим также, что возникающий при суммировании перенос из старшего бита игнорируется. Это приводит к так называемой кольцевой организации памяти — за ячейкой с максимальным адресом FFFFF следует ячейка с нулевым адресом.

Устройство шинного интерфейса получает логический адрес ячейки памяти из различных источников в зависимости от типа выполняемого обращения к памяти. Возможные способы формирования физического адреса приведены в табл. 1.4.

Команды всегда выбираются из текущего сегмента кода — базовый адрес сегмента находится в регистре CS, а смещение — в регистре PC. Стековые команды всегда обращаются к текущему сегменту стека: базовый адрес находится в регистре SS, а смещение — в регистре SP. Считается, что большинство переменных (операндов) находится в текущем сегменте данных, адресуемом регистром DS, но программист может заставить МП обращаться к переменной, находящейся в другом сегменте (под «другим» здесь понимается сегмент, базовый адрес которого находится не в регистре DS, а в каком-то другом сегментном регистре).

Смещение переменной вычисляет операционное устройство в соответствии с определенным в команде режимом адресации. Результат этого вычисления называется *эффек-*

тивным или исполнительным адресом ЕА операнда в памяти. При вычислении ЕА перенос из старшего бита игнорируется. Адресация цепочек данных несколько отличается от адресации других переменных. Считается, что цепочка-источник находится в текущем сегменте данных, но может быть определен и другой текущий сегмент. Смещение всегда берется из регистра SI, что и объясняет его название — индексный регистр источника. Цепочка-получатель всегда находится в текущем дополнительном сегменте данных, а смещение берется из регистра DI — индексного регистра получателя. Команды обработки цепочек автоматически модифицируют содержимое регистров SI и DI по мере продвижения по цепочке.

Когда в команде в качестве источника адреса определен регистр BP, считается, что переменная находится в текущем сегменте стека. Следовательно, регистр BP обеспечивает удобное средство адресации данных, находящихся в стеке. Однако этот же регистр можно использовать для обращений к данным в любом из текущих сегментов.

В большинстве случаев соглашения о сегментах, принимаемых по умолчанию (без явной спецификации), удобны для программистов. Возможно, однако, целенаправленно заставить устройство шинного интерфейса обратиться к переменной в любом из текущих сегментов с единственным исключением — цепочка-получатель всегда должна находиться в дополнительном сегменте. Явная спецификация сегмента достигается с помощью *префикса замены*, т. е. принудительного задания сегмента. Однобайтный префикс замены сегмента должен предшествовать команде (отсюда его название — префикс) и сообщать устройству шинного интерфейса, какой сегментный регистр использовать в следующей за ним команде для формирования физического адреса памяти. Префикс определяется программистом и содержит двухбитное поле, идентифицирующее один из четырех сегментных регистров.

Позиционно-независимые программы. Сегментная структура памяти обеспечивает создание *позиционно-независимых или динамически перемещаемых (перемещаемых) программ*. Позиционная независимость программ в мультипрограммной среде необходима для эффективного использования имеющейся основной памяти. Неактивные программы передаются во внешнюю память (дисковый накопитель), а занимаемое ими место отводится для активных программ. Если в дальнейшем потребовалась программа,

находящаяся на диске, она передается в любую свободную область основной памяти с последующим возобновлением или продолжением выполнения. Далее, если программе требуется большая смежная область памяти, а она разбита на несколько несмежных фрагментов, возможно уплотнить сегменты программы и освободить необходимую область, как показано на рис. 1.11.

Чтобы быть позиционно-независимой, программа не должна модифицировать содержимое своих сегментных регистров и передавать управление вне текущего сегмента

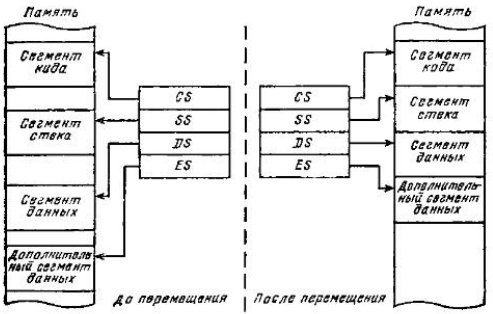


Рис. 1.11. Эффект позиционной независимости программы

кода. Другими словами, все смещения в программе должны быть относительно фиксированных значений, содержащихся в сегментных регистрах. Это позволяет перемещать программу в адресном пространстве памяти произвольным образом, изменяя при этом содержимое сегментных регистров, чтобы они содержали новые базовые адреса сегментов.

Стек. Стек МП К1810, как и стеки всех современных процессоров, реализован в оперативной памяти, и его местоположение определяется содержимым регистров SS и SP. Максимальный размер стека соответствует размеру сегмента и составляет 64К байт. Попытка расширить стек сверх 64К байт приводит к перезаписи начала стека. Регистр SS содержит базовый адрес текущего сегмента стека,

а регистр SP указывает на вершину стека (часто называемую TOS). Другими словами, регистр SP содержит смещение вершины стека от базового адреса сегмента стека. Отметим, что базовый адрес стека в регистре SS не является «дном» стека, так как имеется еще смещение, находящееся в регистре SP.

Команды с обращением к стеку (включение или извлечение) оперируют словами. Слово *включается* в стек путем декремента SP на 2 и записью слова в новую TOS. Слово *извлекается* из стека путем копирования содержимого TOS в операнде-получателе и последующего инкремента SP на 2. Все стековые операции сопровождаются автоматической модификацией содержимого указателя стека SP.

Зарезервированные области памяти. Две области адресного пространства памяти зарезервированы для выполнения особых функций, связанных с обработкой прерываний и системным сбросом. Этими областями являются первые 128 байт (физические адреса 00000 — 0007F) и последние 16 байт (физические адреса FFFF0 — FFFFF). Данные области использовать для других целей нельзя.

Особенности организации памяти. Микропроцессор К1810 может обращаться к находящемуся в памяти байту или слову. Слово с четным адресом передается за один цикл шины. Если слово находится по нечетному адресу, оно передается по байтам в двух смежных циклах шины. Для получения максимальной производительности системы слова следует хранить по четным адресам. Это обстоятельство особенно важно для стеков, оперирующих только словами. Тем не менее отсутствие выравнивания (размещение слов по четным и нечетным адресам) возможно, но приводит к снижению производительности. Следует отметить, что без выравнивания достигается наиболее плотная упаковка данных.

Команды МП всегда выбирает словами по четным адресам, за исключением первой выборки после передачи управления по нечетному адресу, когда выбирается 1 байт. Поток команд разделяется на байты внутри МП, и выравнивание команд не влияет на производительность.

Основным недостатком принятой в МП организации памяти является то, что при выполнении программы трудно манипулировать физическими адресами памяти, например сравнивать два физических адреса. Конкретное значение физического адреса может быть получено из многих комбинаций сегмент:смещение, и, следовательно, недостаточно просто сравнить сегмент:смещение логических адресов. Необходимо сравнивать результирующие физические адреса, а такая команда в системе команд МП К1810 отсутствует. Кроме того, нет и команды ЗАГРУЗИТЬ ФИЗИЧЕСКИЙ АДРЕС, которая вычисляла бы из пары сегмент:смещение физический адрес и загружала его в два регистра.

1.6. Ввод-вывод

Микропроцессор К1810 имеет достаточный набор средств ввода-вывода: большое адресное пространство ввода-вывода, изолированное от пространства памяти, и специальные команды, которые передают данные между регистрами МП и портами в пространстве ввода-вывода. При выполнении этих команд генерируются управляющие сигналы, идентифицирующие выбор пространства ввода-вывода. Допускается также организация ввода-вывода, отображенного на память, чтобы использовать для ввода-вывода всю систему команд (точнее, команды, осуществляющие обращения к памяти) и допустимые режимы адресации памяти. Для быстрых передач блоков данных применяются контроллеры прямого доступа к памяти и специализированные процессоры ввода-вывода.

Адресное пространство ввода-вывода содержит до 64К 8-битных портов или до 32К 16-битных портов. Команды IN (ввод) и OUT (вывод) передают данные между аккумуляторами AL (байты) или AX (слова) и адресуемыми портами.

Пространство ввода-вывода не сегментируется; для обращения к порту шинный интерфейс просто помещает 16-битный адрес порта на младшие 16 линий шины адреса/данных, а 8-битный адрес порта дублируется на линиях старшей и младшей половин этой шины. Различные типы команд ввода-вывода позволяют определять адрес как фиксированное 8-битное значение в команде или как переменную, значение которой находится в регистре DX (косвенная адресация). Второй способ удобен при использовании разделяемых подпрограмм (драйверов) ввода-вывода, когда адрес порта должен определяться при выполнении программы.

Восемь ячеек F8—FF в пространстве ввода-вывода зарезервированы для системных целей, и использовать их в прикладных программах не рекомендуется.

Микропроцессор может передавать по шине байт или слово в (из) УВВ, адрес которого находится в пространстве ввода-вывода. Чтобы слово передавалось за один цикл шины, адрес УВВ должен быть четным. Адрес байтного устройства может быть четным или нечетным, но всем внутренним регистрам устройства должны быть поставлены в соответствие только четные или только нечетные адреса.

Адреса портов ввода-вывода возможно также разместить в адресном пространстве памяти. Такой *ввод-вывод, отображенный на память,* обеспечивает дополнительную гибкость программирования, так как любая команда с обращением к памяти превращается в команду ввода-вывода. Например, команда MOV может передавать данные между регистром (любым общим регистром, а не только аккумулятором) и портом ввода-вывода, а логические команды AND, OR, XOR и TEST можно использовать для манипуляций битами в регистрах периферийного устройства. Дополнительным достоинством такого способа ввода-вывода является гибкость разнообразных режимов адресации памяти. Например, регистры группы терминалов можно организовать в памяти как массив, причем индексный регистр выбирает в группе терминал, который участвует в операциях ввода-вывода.

Естественно, за гибкость программирования ввода-вывода, отображенного на память, приходится расплачиваться. Специализация части памяти для периферийных устройств сокращает число доступных адресов памяти, хотя при адресном пространстве памяти 1М байт это обстоятельство вряд ли существенно. Кроме того, несколько усложняется дешифрирование 20-битного физического адреса памяти, и команды с обращением к памяти выполняются несколько дольше и имеют больший формат, чем простые команды IN и OUT.

При работе в минимальном режиме МП воспринимает сигнал HLD запроса шины (чаще всего для прямого доступа к памяти) и генерирует подтверждающий сигнал HLDA. Эти сигналы совместимы с традиционными контроллерами прямого доступа к памяти, например К580ВВ57. Сигналом HLD контроллер прямого доступа к памяти запрашивает шину для передачи данных непосредственно между периферийным устройством и памятью без участия микропроцессора. Микропроцессор завершает текущий цикл шины, если он был начат, а затем выдает ответный

сигнал HLDA и отключается от шины. Он не пытается использовать шину до снятия сигнала HLD.

В применениях с интенсивным и быстрым вводом-выводом, например при наличии в системе дисковых накопителей, МП может работать с процессором ввода-вывода. Этот процессор имеет два канала прямого доступа к памяти, а его система команд ориентирована на операции ввода-вывода. В отличие от простых контроллеров прямого доступа к памяти процессор ввода-вывода обеспечивает программное обслуживание периферийных устройств, освобождая от этой функции МП. Кроме того, процессор ввода-вывода может передавать данные по своей локальной шине или по системной шине, может сопрягать 8/16-битные периферийные устройства с 8/16-битными шинами, передавать данные из памяти в память или из устройства в устройство, преобразовывать данные в процессе передачи и выполнять другие функции. По своей сложности процессор ввода-вывода приближается к сложности самого микропроцессора.

1.7. Мультипроцессорные средства

По мере освоения массового производства МП и снижения их стоимости все более привлекательным подходом в проектировании становятся мультипроцессорные системы. При распределении системных задач между несколькими параллельно работающими процессорами достигается существенное повышение производительности системы. Кроме того, такой подход ведет к модульной структуре системы, что упрощает ее эксплуатацию и модернизацию. Повышается также надежность и живучесть системы, так как отказ в какой-либо ее подсистеме может быть локализован и скомпенсирован перераспределением функций среди других подсистем.

Однако в мультипроцессорных системах возникают специфические проблемы, связанные главным образом с организацией взаимодействия процессоров и разделением общих ресурсов системы. Одним из таких ресурсов становится системная шина, использовать которую могут теперь несколько процессоров, но которой в любой момент времени может распоряжаться (управлять) только один процессор.

Микропроцессор К1810 спроектирован с учетом возможности его работы в мультипроцессорной среде. В нем

предусмотрены встроенные аппаратные средства, помогающие решить те проблемы координации действий нескольких процессоров, которые сдерживали разработку мультипроцессорных систем.

Блокировка шины. При работе в максимальном режиме МП может генерировать сигнал блокировки шины $\overline{\text{LOCK}}$. Шинный интерфейс формирует сигнал $\overline{\text{LOCK}}$, когда операционное устройство выполняет однобайтную команду LOCK, называемую *префиксом блокировки*. Сигнал блокировки шины остается активным (имеет низкий уровень) в течение всего времени выполнения команды, которой предшествовал префикс LOCK. Следует отметить, что префикс LOCK не влияет на прерывания. Если шину запрашивает другой процессор (по линиям $\overline{\text{RQ}}/\overline{\text{E}}$ — см. ниже), МП фиксирует и запоминает запрос, но не подтверждает его до завершения выполнения соответствующей команды.

Сигнал $\overline{\text{LOCK}}$ остается активным на время выполнения одной команды. Если префиксы LOCK имеют две соседние команды, между ними все-таки остается незаблокированный временной интервал. При блокировке цепочечной команды сигнал LOCK сохраняется в течение всего времени ее выполнения.

При работе МП в минимальном режиме сигнал $\overline{\text{LOCK}}$ отсутствует. Все-таки префикс LOCK можно использовать для задержки генерирования подтверждающего сигнала HLDA на запрос HLD до завершения выполнения заблокированной команды. Напомним, что в обычных условиях МП генерирует сигнал HLDA при завершении текущего цикла шины.

Сигнал блокировки выполняет только информационную функцию. Когда он активен, другие процессоры, подключенные к разделенной шине, не должны пытаться запрашивать ее. Если в системе для управления доступом к разделенной шине применяется арбитр шины (см. ниже), он воспринимает свой входной сигнал $\overline{\text{LOCK}}$ и не освобождает шину до тех пор, пока активен этот сигнал.

Сигнал $\overline{\text{LOCK}}$ применяется для координации доступа к общим (разделяемым) ресурсам, например буферной области в памяти, файлу или указателю. Когда доступ к ресурсу неуправляем, один из процессоров может считать из него ошибочные данные в то время, когда другой процессор модифицирует их (рис. 1.12).

Рис. 1.12. Неуправляемый доступ к разделенному ресурсу

Доступом к разделяемому ресурсу можно управлять, используя префикс LOCK с командой XCHG (обмен регистра с памятью), что показано на рис. 1.13. Основу управляемого доступа к ресурсу составляет *семафор* — программно-устанавливаемый флажок или переключатель, который показывает, доступен ресурс (семафор=0) или занят (семафор=1). Процессоры, подключенные к шине, действуют по соглашению — не использовать ресурс до тех пор, пока семафор не покажет его доступность. Процессор

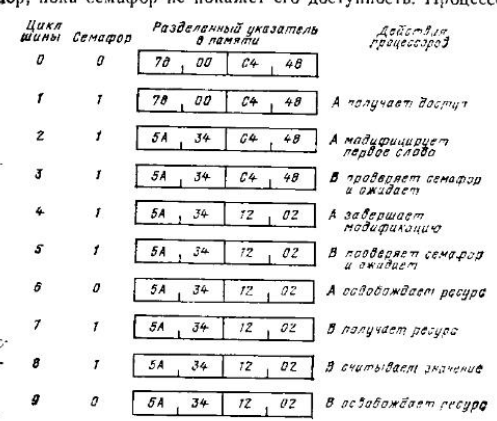


Рис. 1.13. Управляемый доступ к разделенному ресурсу

устанавливает семафор в состояние 1, когда он использует ресурс, и сбрасывает семафор в состояние 0, когда использование ресурса закончено.

Команда XCHG может считать текущее значение семафора и установить его в состояние «занят» в одном цикле команды. Однако для обмена 8-битных значений ей требуются два цикла шины. Другой процессор может получить доступ к шине между этими двумя циклами и считать неправильное значение семафора. Такую ситуацию можно предотвратить, если ввести перед командой XCHG префикс LOCK (рис. 1.14). Таким образом, с помощью блокировки

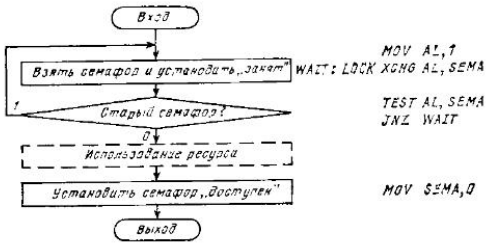


Рис. 1.14. Использование команды XCHG и префикса LOCK

шины реализуется управление доступом к семафору, а через него и к разделяемому ресурсу (в некоторых процессорах аналогичный эффект достигается тем, что команда XCHG выполняется с так называемым неделимым циклом шины, т. е. она обязательно выполняется полностью независимо от запросов шины другими процессорами).

Команда WAIT и вход TEST. В любой конфигурации можно синхронизировать действия МП с внешним событием с помощью команды ожидания WAIT и входного сигнала проверки TEST. Когда операционное устройство выполняет команду WAIT, результат зависит от уровня сигнала на входной линии TEST. Если сигнал TEST не активен (высокий уровень), МП вводит холостое состояние и периодически проверяет сигнал TEST через пять тактов синхронизации. Когда сигнал TEST активен, МП переходит к выполнению команды, следующей за командой WAIT.

Команда ESC. В мультипроцессорных системах для улучшения их технико-экономических показателей применяются так называемые подчиненные (вспомогательные) процессоры или сопроцессоры. Типичным примером сопроцессора является арифметический процессор, система команд которого ориентирована на сложные операции (тригонометрические и обратные тригонометрические функции, логарифмы, извлечение корня и др.) и представление чисел в формате с плавающей точкой. Сопроцессор не имеет собственной программы, а получает команды и, возможно, операнды из командного потока ведущего (главного) процессора. Взаимодействие ведущего процессора и сопроцессора осуществляется с помощью команды выдачи ESC, называемой также командой переключения на сопроцессор.

При записи этой команды в программах для МП К1810 программист определяет 6 бит ее кода операции путем задания мнемоники команды сопроцессора, например FSQRT — извлечение квадратного корня. Другими словами, система команд МП К1810 как бы расширяется, включая в себя команды сопроцессора. При выполнении программы сопроцессор анализирует управляющие сигналы и фиксирует цикл шины, связанный с выборкой команды. Если при этом на шину адреса/данных из памяти считывается команда ESC, сопроцессор перехватывает ее, считая своей командой. Программируемые 6 бит заставляют сопроцессор реализовать предопределенные действия по выполнению своей команды.

Когда МП К1810 работает в максимальном режиме, сопроцессор после определения выборки команды ESC контролирует линии состояния очереди команд QS и фиксирует момент, когда начинает выполняться команда ESC. Если команда инициирует цикл считывания из памяти, сопроцессор перехватывает физический адрес операнда и (или) значение операнда. Сигнализация о завершении операции сопроцессора может быть реализована с помощью механизма WAIT — TEST, чем обеспечивается взаимная синхронизация работы ведущего процессора и сопроцессора.

Отметим, что выборка команды ESC не эквивалентна началу ее выполнения, так как она попадает в очередь команд. Ей может предшествовать команда передачи управления, которая вызывает реинициализацию очереди команд. Поэтому начало выполнения команды ESC сопро-

цессор фиксирует по сигналам состояния очереди команд.

Линии $\overline{RQ}/\overline{E}$. В максимальном режиме работы линии HLD и HLDA превращаются в линии $\overline{RQ}/\overline{E0}$ и $\overline{RQ}/\overline{E1}$ для генерации более сложных управляющих сигналов. Эти двунаправленные линии можно использовать для разделения локальной шины между процессорами с помощью механизма квитирования, который представляет собой трехфазный цикл: запрос, разрешение и освобождение. Сначала запрашивающий шину процессор генерирует импульс на линии $\overline{RQ}/\overline{E}$. Микропроцессор отвечает выходным импульсом по той же линии, сообщая о переходе в состояние «Подтверждение запроса», и освобождает шину. В течение последующего временно́го интервала шинный интерфейс логически отключен от шины. Однако операционное устройство продолжает выполнять команды до тех пор, пока очередная команда не потребует доступа к шине или пока не исчерпается очередь команд (до первого из этих событий). Когда другой процессор закончил операции на шине, он генерирует сигнал на линии $\overline{RQ}/\overline{E}$ и МП возобновляет управление шиной. Обмен сигналами происходит синхронно, и оба взаимодействующих процессора должны синхронизироваться от одного генератора.

Линия $\overline{RQ}/\overline{E0}$ имеет бо́льший приоритет, чем линия $\overline{RQ}/\overline{E1}$. Если запросы шины появляются одновременно на обеих линиях, разрешение получает процессор, выдавший запрос по линии $\overline{RQ}/\overline{E0}$, а запрос по линии $\overline{RQ}/\overline{E1}$ подтверждается после того, как управление шиной будет возвращено микропроцессору. Если на линии $\overline{RQ}/\overline{E0}$ появляется запрос в то время, когда МП отреагировал на предыдущий запрос по линии $\overline{RQ}/\overline{E1}$, то второй запрос не подтверждается до тех пор, пока процессор на линии $\overline{RQ}/\overline{E1}$ не освободит шину. Оба сигнала $\overline{RQ}/\overline{E}$, а также сигнал HLD в минимальном режиме имеют более высокий приоритет, чем ожидающее прерывание.

Арбитр шины. В мультипроцессорных системах необходимы аппаратные средства координации использования процессорами разделяемой шины. Такие координацию и управление обеспечивает арбитр шины, который работает совместно с контроллером шины. Арбитр устраняет условия гонок, разрешает конфликты из-за доступа к шине и согласует асинхронно работающие процессоры. Когда возникают одновременные запросы шины, арбитр разрешает

конфликт и отдает шину процессору с наибольшим приоритетом. Обычно арбитр может быть программно настроен на один из нескольких способов учета приоритетов.

1.8. Прерывания

Микропроцессор К1810 имеет простую, но достаточно универсальную систему прерываний. Каждому прерыванию поставлен в соответствие код типа (указатель), который идентифицирует прерывание для МП. Допускается обработка до 256 типов прерываний. Прерывания могут инициироваться устройствами, внешними по отношению к микропроцессору, а также командами программных прерываний. Наконец, в некоторых ситуациях прерывание генерирует сам микропроцессор. Возможные источники прерываний показаны на рис. 1.15.

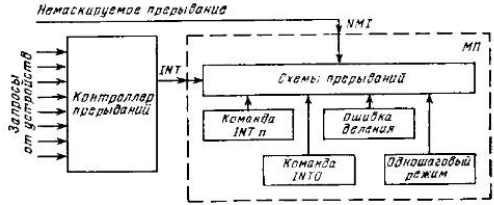


Рис. 1.15 Источники прерываний

Общая реакция МП на прерывание представлена на рис. 1.16; далее содержание этого рисунка поясняется подробнее.

Прерывание следует рассматривать как некоторый сигнал, заставляющий МП прервать выполнение текущей программы и переключиться на выполнение другой, более важной или срочной программы, называемой процедурой (подпрограммой) обслуживания прерывания, или процедурой прерывания. После обслуживания прерывания возобновляется выполнение прерванной программы.

Главным условием правильной реакции МП на прерывание является возобновление прерванной программы так, как будто прерывания вообще не было. Внешнее прерыва-

ние может появиться в произвольный момент времени, т. е. асинхронно по отношению к действиям МП. Следовательно, реагируя на прерывание, МП должен временно запомнить место в прерываемой программе, где возникло прерывание. Это место идентифицирует адрес той команды, которая выполнялась бы, если бы прерывания не было. Его

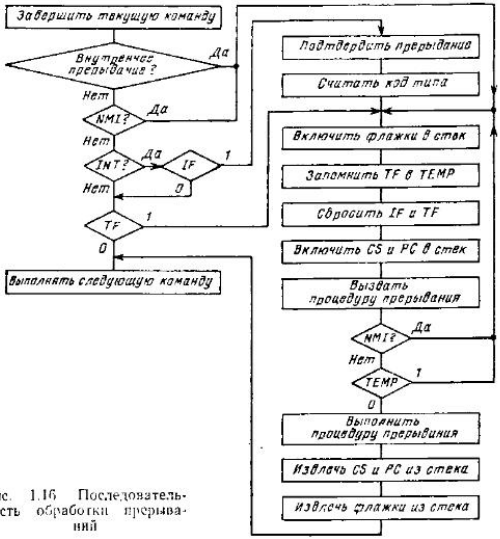


Рис. 1.16. Последовательность обработки прерываний

называют адресом возврата. Адрес возврата содержат регистры PC и CS, а удобным временным «хранилищем» адреса возврата является стек. Следует специально отметить, что в стек включается скорректированное содержимое PC, соответствующее адресу команды, перед которой МП начал обслуживать прерывание. Текущее содержимое PC с учетом очереди адресует команду с опережением.

Кроме содержимого регистров PC и CS, важная информация для прерываемой программы находится в регистре флажков. Процедура прерывания почти наверняка изменит состояния флажков. Следовательно, при восприятии прерывания следует автоматически запомнить в стеке и содержимое регистра флажков. Наконец, при необходимости сама процедура прерывания может включить в стек содержимое тех регистров, которые потребуются для ее выполнения, но которое без изменений должно быть возвращено прерванной программе. Конкретные действия, реализуемые процедурой прерывания, определяет пользователь при ее программировании. Когда обслуживание прерывания завершается, запомненные значения извлекаются из стека и прерванная программа продолжает выполняться из того состояния, в котором она была прервана.

Внешние прерывания. В микропроцессоре имеются два входа прерываний — маскируемого прерывания INT и немаскируемого прерывания NMI. На вход INT обычно подключается выход прерывания программируемого контроллера прерываний, на входы которого подаются сигналы запросов прерываний от периферийных устройств. Для программы контроллер прерываний выглядит как порт ввода-вывода, в котором фиксируется код типа (указатель) прерывания. Его основная функция заключается в восприятии сигналов от подключенных периферийных устройств, определении запроса от устройства с максимальным приоритетом и генерировании сигнала INT, если выбранное устройство имеет больший приоритет, чем приоритет выполняемой микропроцессором программы (ею, в частности, может быть процедура прерывания, инициированная запросом другого устройства.

Когда сигнал на линии INT активен (высокий уровень), действия МП зависят от состояния флажка IF разрешения прерываний. Однако до завершения текущей команды МП вообще не предпринимает никаких действий.

Примечание. Имеется несколько случаев, когда сигнал INT не распознается и до завершения следующей команды. Префиксы повторения, блокировки шины и замены сегмента считаются частью команды, перед которой находится любой из них; поэтому прерывание не воспринимается между префиксом и командой. Команды передачи (MOV) в сегментный регистр и извлечения (POP) из стека в сегментный регистр рассматриваются аналогично: прерывание не распознается до завершения следующей за ними команды. Такой механизм защищает программу, которая переходит к новому стеку путем изменения содер-

жимого регистров SS и SP. Если бы прерывание распознавалось после модификации регистра SS, но до модификации регистра SP, микропроцессор включал бы содержимое регистров флажков, CS и PS в неправильную область памяти. Это следует из того факта, что, когда сегментный регистр и другой регистр должны модифицироваться вместе, сегментный регистр должен изменяться первым, а затем должна следовать команда, которая изменяет содержимое другого регистра.

Имеются два особых случая (команда WAIT и цепочечная команда с повторением), когда запрос прерывания распознается в середине команды. В этих случаях прерывания воспринимаются после любой законченной цепочечной операции или цикла проверки сигнала на входе $\overline{TEST}$.

Если флажок IF сброшен, т. е. прерывания по входу INT запрещены (замаскированы), МП игнорирует запрос прерывания и переходит к следующей команде. Микропроцессор не запоминает состояние сигнала INT, поэтому он должен сохраняться активным до тех пор, пока прерывающее устройство не получит сигнала подтверждения или само не снимет запрос. Когда прерывания по входу INT разрешены (IF=1), МП распознает запрос прерывания и обрабатывает его. Состоянием флажка IF программист может управлять с помощью команд STI (установка) и CLI (сброс). Кроме того, допускается селективное маскирование прерываний от отдельных устройств путем передачи в контроллер прерываний соответствующих приказов. Команда STI (а также команда IRET — см. ниже) разрешает прерывания только после завершения следующей за ней команды.

Микропроцессор подтверждает запрос прерывания, выполняя два последовательных цикла INTA — подтверждения прерывания. Если в этих циклах появляется запрос шины по линии HLD или $\overline{RQ}$/E, он не воспринимается до завершения обоих циклов INTA. В максимальном режиме работы МП генерирует в этих циклах сигнал $\overline{LOCK}$, чтобы другие процессоры не пытались запрашивать шину. Первый цикл INTA сигнализирует контроллеру прерываний о восприятии запроса. Во втором цикле INTA контроллер выдает на шину данных байт типа прерывания (беззнаковое целое в диапазоне 0—255), ассоциируемый с запрашивающим обслуживание периферийным устройством. Микропроцессор считывает код типа прерываний и использует его для вызова процедуры прерывания, соответствующей прерывающему устройству. В общем вход INT микропро-

цессора К1810 аналогичен соответствующему входу микропроцессора К580.

Запросы внешних прерываний подключаются также к входной линии немаскируемого прерывания NMI. Этот вход воспринимает переход сигнала от низкого уровня к высокому (положительный фронт), чтобы текущая программа в МП не прерывалась от одного сигнала несколько раз. Вход NMI применяется для сигнализации микропроцессору о катастрофическом событии, требующем немедленной реакции, например об аварийном отключении сети, обнаружении ошибки в памяти и др. Запросы NMI нельзя запретить; они запоминаются в микропроцессоре и имеют более высокий приоритет, чем прерывания по входу INT. Немаскируемые прерывания имеют фиксированный код типа 2; для вызова соответствующей процедуры прерывания микропроцессору не требуется код типа, поэтому в ответ на NMI циклы шины подтверждения прерывания INTA не формируются. Этим достигается ускоренная реакция МП на запросы немаскируемых прерываний.

Время, необходимое МП для распознавания внешнего прерывания, т. е. запаздывание обслуживания прерывания, зависит от времени завершения текущей команды. В среднем наибольшее запаздывание получается при выполнении команд умножения, деления и многобитного сдвига. Как указывалось выше, иногда запаздывание определяется временем выполнения двух команд.

Внутренние прерывания. В отличие от предшествующих микропроцессоров в МП К1810 предусмотрено несколько внутренних прерываний, генерируемых при выполнении программы. Команда INT *n* вызывает прерывание сразу после своего завершения. Тип прерывания *n*, закодированный программистом в команде, определяет вызываемую процедуру прерывания. Следовательно, эту команду можно использовать для отладки процедур прерываний, обслуживающих периферийные устройства.

Команда INTO генерирует прерывание типа 4 после своего завершения, если установлен флажок переполнения OF.

Микропроцессор самостоятельно генерирует прерывание типа 0 сразу после выполнения команд деления DIV и IDIV, если формат частного превышает формат получателя (ошибка деления).

Если, наконец, установлен флажок TF пошаговой работы, МП автоматически генерирует прерывание типа 1 пос-

ле выполнения каждой команды. Пошаговый или покомандный режим предназначен для отладки программ.

Внутренние прерывания характеризуются следующими свойствами:

тип прерывания либо предопределен, либо содержится в коде команды;

циклы шины подтверждения прерывания INTA не формируются;

внутренние прерывания нельзя запрещать, кроме прерывания пошаговой работы;

любое внутреннее прерывание (за исключением прерывания пошаговой работы) имеет более высокий приоритет, чем внешние прерывания. Если запрос NMI или INT появляется при выполнении команды, которая сама генерирует внутреннее прерывание (например, ошибка деления), оно обрабатывается первым.

Приоритеты прерываний в поридке их убывания:

прерывание из-за ошибки деления,

программное прерывание, инициируемое командой INT,

команда прерывания при переполнении,

немаскируемое прерывание NMI,

маскируемое прерывание INT,

прерывание пошаговой работы.

Таблица указателей векторов прерываний. Таблица указателей векторов прерываний осуществляет связь между кодом типа прерывания и процедурой, которая обслуживает прерывание данного типа. Таблица занимает 1К байт памяти с диапазоном адресов 0—3FF и может содержать до 256 элементов (рис. 1.17). Каждый элемент *i* представляет собой полный (два слова), начальный логический адрес процедуры, которая обслуживает прерывание типа *i*. Слово с бо́льшим адресом содержит базовый адрес сегмента, а слово с меньшим адресом содержит смещение процедуры от начала сегмента. Так как каждый элемент таблицы состоит из 4 байт, МП вычисляет адрес нужного элемента таблицы путем простого умножения кода типа прерывания на 4 (двойной сдвиг влево).

Если в конкретной микропроцессорной системе не применяются прерывания, коды типов которых соответствуют старшим адресам таблицы указателей, эту область памяти допускается использовать для других целей. Однако первые 128 байт в таблице зарезервированы для фиксированных типов прерываний и других системных целей; их нельзя использовать для другого назначения.

После включения в стек содержимого регистра флажков МП инициирует процедуру прерывания, выполняя операции, эквивалентные действию команды межсегментного косвенного вызова CALL. Адресом передачи управления служит при этом полный адрес, содержащийся в найденном элементе таблицы указателей векторов прерываний. Микропроцессор запоминает адрес возврата (адрес следующей по порядку команды), включая в стек содержимое регистров CS и PC. Затем в эти регистры загружают-

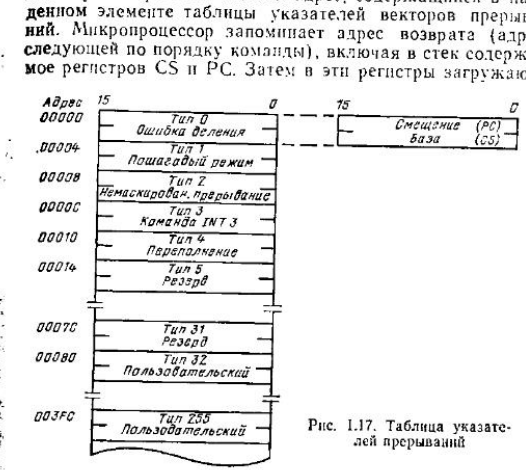


Рис. 1.17. Таблица указателей прерываний

ся второе и первое слова элемента таблицы указателей, что эквивалентно передаче управления процедуре. Принцип косвенного перехода через таблицу указателей поясняется на рис. 1.18.

Процедуры прерываний. Когда вызывается процедура прерывания, содержимое регистров флажков CS и PC включено в стек, а флажки IF и TF сброшены. Процедура может разрешить внешние прерывания командой STI, допуская прерывание самой себя запросом на входе INT. Кроме того, она может быть всегда прервана запросом на входе немаскируемых прерываний NMI. Внутренние прерывания, возникающие при выполнении процедуры, будут также прерывать ее. В каждой процедуре необходимо

тщательно следить за тем, чтобы в ней не возникло прерывание того типа, которое она обслуживает. Например, попытка деления на нуль в процедуре прерывания из-за ошибки деления (тип 0) приведет к бесконечным вызовам этой процедуры. Внешне такая ситуация вызовет непредсказуемое поведение системы. Необходимо также позаботиться и о том, чтобы стек был рассчитан на максимальную глубину вложения прерываний, которая может возникнуть в системе.

Каждая процедура прерывания должна запоминать (обычно в стеке) содержимое всех регистров, которые она использует, до их модификации, а перед завершением восстанавливать содержимое этих регистров. Целесообразно разрешать восприятие прерываний по входу INT для всех частей процедуры, за исключением критических секций, которые нельзя прерывать без риска получения ошибочных результатов. Если прерывания запрещены слишком долго, возникает потенциальная опасность потери запросов прерываний по входу INT.

Процедура прерывания должна заканчиваться командой возврата из прерывания IRET. Перед ее выполнением предполагается, что стек находится в том состоянии, в каком он был сразу после вызова процедуры. Команда IRET извлекает три верхние слова из стека в регистры PC, CS и флажков, что обеспечивает возврат к команде, которая выполнялась бы без инициирования процедуры прерывания.

Фактические действия процедуры зависят от ее назначения. Если процедура обслуживает периферийное устройство, она должна вывести в него приказ о снятии запроса прерывания. Затем она может вывести из устройства информацию о состоянии, определить причину прерывания и предпринять соответствующие действия.

Процедуры программных прерываний (инициируемые командой INT *n*) можно использовать как обслуживающие подпрограммы (вызовы супервизора) для других программ, находящихся в системе. При этом процедура прерывания инициируется в том случае, когда внимания требует не периферийное устройство, а программа. (Под вниманием может пониматься поиск записи в файле, передача сообщения в другую программу, запрос о выделении свободной памяти и др.). Другими словами, команду INT *n* следует считать эквивалентом команды вызова супервизора SVC, которая имеется в процессорах средних и

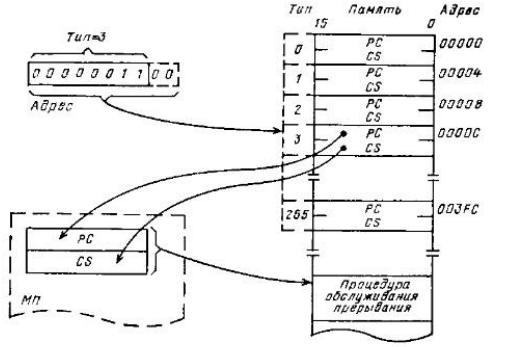


Рис. 1.18. Переход через таблицу указателей

больших ЭВМ, а также в некоторых микропроцессорах.

Процедуры программных прерываний удобно применять в тех системах, которые допускают динамическое перемещение программ при выполнении. Поскольку таблица указателей находится в фиксированной области памяти, процедуры могут вызывать друг друга через таблицу с помощью команд программных прерываний. Этим обеспечивается устойчивое взаимодействие, не зависящее от адресов процедур. Сами процедуры могут перемещаться в памяти, но с условием соответствующей модификации таблицы указателей.

Прерывание типа 0. Прерывание из-за ошибки деления возникает в том случае, когда результат операции деления превышает максимально допустимое значение, которое может быть загружено в получатель, например при делении на нуль. Если в процедуре обслуживания данного прерывания не предусматривается разрешение прерываний по входу INT, то время ее выполнения должно учитываться при расчете продолжительности команды деления в наихудшем случае. Такой расчет необходим при оценке максимального цикла команды и его влияния на задержку восприятия внешних прерываний.

Прерывание типа 1. Прерывание пошагового (покоманд-

ного) режима автоматически генерируется микропроцессором после выполнения каждой команды, если установлен флажок TF. Напомним, что при обработке прерывания МП включает в стек содержимое регистров флажков, CS и PS, а затем сбрасывает флажки IF и TF. Следовательно, после вызова соответствующей процедуры МП не работает в пошаговом режиме — он работает обычным образом. Когда процедура завершается, из стека извлекаются старые состояния флажков и МП переводится в пошаговый режим работы.

Пошаговый режим является ценным средством отладки, так как его процедура прерывания действует как «окно», через которое можно наблюдать работу системы по командам. Эта процедура может, например, индицировать содержимое внутренних регистров МП (в том числе и содержимое PC, так как оно находится в стеке), значения важных переменных, находящихся в памяти, и т. д., а также их изменения после каждой команды. Появляется возможность подробно проследить ход выполнения программы и найти место, где возникает расхождение между ожидаемыми и реально получающимися результатами.

Микропроцессор не имеет команд установки и сброса флажка TF. Однако состояние TF программист может изменять, воздействуя на состояния флажков, включенных в стек. Для прямого включения в стек содержимого регистра флажков предусмотрена команда PUSHF, а для извлечения вершины стека в регистр флажков—команда POPF. Флажок TF устанавливается путем объединения по ИЛИ состояний флажков с непосредственным операндом 0100 и сбрасывается при объединении по И с константой FEFF. Если флажок TF установлен таким образом, первое пошаговое прерывание происходит после выполнения команды, следующей за командой возврата из процедуры пошагового прерывания.

В одношаговом режиме МП реагирует на внешние и внутренние прерывания. Обычным образом (с включением в стек содержимого регистров флажков, CS и PC) управление передается процедуре обработки прерывания того типа, которое возникло. Однако до выполнения первой команды этой процедуры распознается прерывание пошагового режима и управление тем же образом (флажки, CS и PC включаются в стек) передается процедуре прерывания типа 1. Когда процедура пошагового режима завершается, управление возвращается предыдущей процедуре прерывания.

Прерывание типа 3 специализировано как прерывание контрольной точки (или контрольного останова). Контрольной точкой называется любое место в программе, где нормальное ее выполнение приостанавливается и можно реализовать некоторую специальную обработку. Контрольные точки обычно вводятся в программы при отладке как средство индикации содержимого регистров, ячеек памяти и портов ввода в критических местах программы.

Команда INT 3 имеет длину 1 байт (остальные команды INT *n* — двухбайтные), что обеспечивает простую вставку ее в любом месте программы. Эту команду можно также использовать, чтобы исправить программу, т. е. вставить дополнительные команды, без ее повторной трансляции. Для этого байт команды запоминается и заменяется командой INT 3 (код операции CC). Процедура контрольной точки должна содержать новые машинные команды, а также код для восстановления запомненного байта команды и декремента PC в стеке, чтобы после вставленных команд выполнялась замененная команда.

Отметим, что данный способ требует программирования на машинном языке и особой осторожности, так как при попытке исправить имеющиеся в программе ошибки в нее легко ввести новые ошибки. Вставку следует считать временной мерой и применять в исключительных ситуациях. Необходимо как можно скорее исправить программу и осуществить ее повторную трансляцию.

1.9. Управление микропроцессором

Микропроцессор К1810 имеет несколько входных и выходных линий, определяющих порядок его функционирования.

Запуск и сброс микропроцессора. Сигнал на входной линии сброса CLR предназначен для упорядоченного запуска МП после включения питания и приведения его внутренних узлов в известное начальное состояние. Для правильного запуска одновременно с подачей напряжения питания необходимо обеспечить действие на входе CLR сигнала высокого уровня. Минимальная его продолжительность после достижения номинального значения напряжения питания составляет 50 мкс. Чтобы привести МП в исходное состояние в процессе работы (операция повтор-

ного запуска или рестарта), на вход CLR подается сигнал высокого уровня с продолжительностью не менее четырех тактов синхронизации.

При действии активного сигнала CLR микропроцессор прекращает все свои действия и переводит линии шины адреса/данных/состояния в высокоимпедансное состояние. Одна часть выходных управляющих сигналов переводится в высокоимпедансное состояние, а другая — в пассивное.

Регистры МП инициализируются следующим образом: (PC) = 0000, (CS) = FFFF, а остальные сегментные регистры и регистр флажков сброшены. Внутренняя очередь команд сбрасывается, т. е. становится пустой.

В соответствии с начальным состоянием регистров CS и PC микропроцессор после сброса выбирает первую команду из ячейки с абсолютным адресом FFFF0. Обычно здесь находится команда прямого межсегментного перехода JMP, которая осуществляет передачу управления первой команде программы.

Поскольку после действия сигнала CLR прерывания по входу INT запрещены (регистр флажков сброшен и IF=0), программа должна разрешить прерывания, как только система инициализирована до такого состояния, в котором их можно обслуживать.

Останов микропроцессора. Когда выполняется команда HLT, микропроцессор переходит в состояние останова. Оно интерпретируется следующим образом: «прекратить все действия до появления сигнала внешнего прерывания или сигнала сброса».

В состоянии останова МП не выдает никаких управляющих сигналов, а состояние шины адреса/данных/состояния не определено. В этом состоянии МП обычным образом подтверждает запросы шины по линии HLDA (минимальный режим) или по линиям $\overline{RQ}$/E (максимальный режим) и возвращается в состояние останова после снятия сигнала запроса шины.

Состояние останова можно использовать в тех ситуациях, когда правильному функционированию системы препятствует некоторое событие. Примером такого события может служить прерывание по сбою питания. После уведомления о неизбежном выключении питания МП использует оставшееся время (пока напряжение питания не упадет ниже минимально допустимого значения) для передачи содержимого внутренних регистров и значений важных переменных в экономичную оперативную память, выполненную по КМОП-технологии и имеющую резервное аккумуляторное питание. Затем он переходит в состояние останова и остается в нем до тех пор, пока о восстановлении питания не просигнализирует внешнее прерывание или системный сброс.

Команду останова можно использовать также для синхронизации работы МП с некоторым внешним событием, инициирующим сигнал прерывания INT. Естественно, в этом случае (как и в предыдущем) необходимо позаботиться о том, чтобы до выполнения команды останова прерывания были разрешены.

Сигналы состояния микропроцессора. В максимальном режиме работы МП формирует восемь сигналов состояния, которые могут использовать внешние устройства. Сигналы $\overline{ST2}$—$\overline{ST0}$ идентифицируют тип цикла шины и состояние МП в соответствии с табл. 1.1. Обычно эти сигналы дешифрируются контроллером шины, который формирует необходимые управляющие сигналы. Сигналы ST3 и ST4 показывают, какой сегментный регистр был использован в формировании физического адреса, участвующего в данном цикле шины. Сигнал ST5 повторяет состояние флажка прерывания IF. Сигнал ST6 всегда имеет низкий уровень, а значение резервного сигнала ST7 не определено.

Сигналы состояния очереди команд. В максимальном режиме МП выдает на линии QS1 и QS0 информацию об операциях над очередью команд в соответствии с табл. 1.2.

ГЛАВА 2

РЕЖИМЫ АДРЕСАЦИИ И СИСТЕМА КОМАНД МИКРОПРОЦЕССОРА

2.1. Введение в язык ассемблера

Рассмотренная в гл. 1 программная модель микропроцессора, дополненная реализованным физически адресным пространством памяти и конкретной конфигурацией портов ввода и вывода, показывает, *чем* может распоряжаться программист, разрабатывая прикладную программу. Разумеется, не менее важный вопрос связан с тем, *каким образом* можно распоряжаться потенциальными ресурсами системы. Ответ на этот вопрос дает изучение режимов адресации и системы команд микропроцессора.

Режимы адресации показывают те способы, с помощью которых осуществляется доступ к командам программы и данным. Разнообразие и гибкость режимов адресации непосредственно влияют на длину программы и время ее выполнения. В свою очередь, система команд также оказывает существенное воздействие на разработку прикладных программ, являясь по существу «дирижерской палочкой» в руках программиста, с помощью которой он управляет всеми действиями в системе. Таким образом, для разработки эффективных программ необходимо в совершенстве знать и режимы адресации и систему команд. Поскольку изучение обоих вопросов должно сопровождаться примерами ассемблерных операторов, настоящая глава начинается с элементарного введения в язык ассемблера [3,4].

Язык ассемблера, называемый также языком символического кодирования и мнемокодом, представляет собой машинный язык в символической форме, которая более понятна и удобна человеку. Сложная внутренняя структура, разнообразные форматы команд, сегментная организация памяти и многочисленные режимы адресации МП К1810 практически исключают возможность разработки сложных и объемных программ на языке ассемблера. Однако в любых сложных и объемных программах всегда найдутся критические секции программы с интенсивным использованием аппаратных средств системы, которые придется разрабатывать не на языке высокого уровня, а на языке ассемблера.

Пользуясь языком ассемблера, программисты могут распоряжаться всеми ресурсами МП К1810, но для этого их нужно, естественно, хорошо знать. Возможности МП, недоступные в языках высокого уровня, но имеющиеся в языке ассемблера, включают в себя программные прерывания, команды WAIT и ESC, а также управление сегментными регистрами.

Ассемблерные программы обычно эффективнее (занимают меньший объем памяти и выполняются быстрее) эквивалентных программ, транслированных с тех или иных языков высокого уровня. Если программа выполняется в режиме компиляции, то компилятор не знает целиком всю программу и должен генерировать обобщенный набор команд, который работает во всех случаях, но может не быть оптимальным в конкретной ситуации. Пусть, например, необходимо просуммировать значения элементов массива и поместить результат в ячейку памяти. Машинные команды, сформированные компилятором, будут передавать

очередной элемент массива в регистр, а затем прибавлять содержимое регистра к сумме, находящейся в памяти. Программируя эту задачу на языке ассемблера, следует сложить значения всех элементов в регистре, а затем передать его содержимое в ячейку памяти. При этом экономится одна команда на каждом элементе массива.

Язык ассемблера МП К1810 является довольно необычным и сложным, что в первую очередь объясняется сегментной организацией памяти и одновременной адресацией четырех сегментов. В языке имеется более 100 базовых символических команд, в соответствии с которыми ассемблер генерирует более 3800 машинных команд. Кроме того, в распоряжении программиста имеется около 20 директив, предназначенных для распределения памяти, инициализации переменных, условного ассемблирования и т. д. Если учесть, что в любой программе имеется множество данных и адресов, то разработка ассемблерной программы превращается в кропотливый труд и требует много усилий.

В языке ассемблера предусмотрены мощные средства структурирования данных, которые обычно характерны только для языков высокого уровня. В нем использованы так называемые обобщенные или универсальные мнемоники команд, которые программисту легче запоминать и применять. Например, хотя имеется 28 типов команд передач данных, программист пользуется одной формой команды

MOV *dst*, *src* *dst*: = (*src*)

Ассемблер формирует правильную машинную команду, используя атрибуты операндов *dst* и (или) *src*. Примеры записи команды MOV:

```
MOV  CX, DX        ; Передача регистр — регистр
                   ;  (слово)
MOV  AL, DH        ; Передача регистр — регистр
                   ;  (байт)
MOV  ES, CX        ; Передача в сегментный ре-
                   ;  гистр
MOV  CL, 0FFH      ; Загрузка константы в регистр
                   ;  (байт)
MOV  [BP] [SI] 800H ; Загрузка константы в память
                   ;  (слово)
MOV  DI, [DI+4]    ; Передача память — регистр
                   ;  (слово)
```

В языках ассемблера других микропроцессоров для выполнения аналогичных операций применяются мнемоники LOAD (загрузить в регистр), STORE (запомнить в памяти), MVI (передать константу) и др.

Ассемблер осуществляет контроль совместимости определения и использования операндов, фиксируя наиболее распространенные ошибки. Такие ошибки могут иногда проникать в сложную программу, и для их локализации и исправления требуются значительные усилия и время.

Совокупность команд и директив, представляемая как функциональная единица для обработки ассемблером, называется *исходным модулем*. Небольшая программа может состоять из одного модуля, а сложная — из десятков. Исходные модули создаются с помощью *редактора текста* и обычно хранятся в виде *исходного файла* во внешней памяти. Ассемблер транслирует исходный модуль в *перемещаемый объектный* модуль, содержащий машинные команды и служебную информацию. Термин «перемещаемый» означает, что все обращения к памяти осуществляются не по абсолютным адресам, а по относительным. Модуль обычно не выполняется до тех пор, пока относительные обращения не заменяются абсолютными с учетом размещения той области памяти, в которую модуль загружается для выполнения. Процесс замены относительных обращений (адресов) абсолютными называется *перемещением*. Перемещаемые объектные модули с помощью специальной программы-библиотекаря вводятся в особый системный файл, называемый *библиотекой*. Объединение объектных модулей, разработанных программистом, и (при необходимости) библиотечных модулей осуществляет *редактор связей*, или компоновщик.

Кроме объектного модуля, хранимого во внешней памяти в виде объектного файла, ассемблер генерирует также *листинг* и *файл листинга* программы. Листинг представляет собой регистрацию исходного модуля, объектного модуля и процесса ассемблирования. Он содержит также диагностические сообщения, отмечающие ошибки программирования. Если например, программист указал 16-битное значение в команде, оперирующей по контексту программы 8-битным значением, ассемблер выдает сообщение об ошибке — значение превышает допустимый диапазон.

В некоторых версиях ассемблера формируется специальный файл ошибок. Он содержит все синтаксические и некоторые семантические ошибки, обнаруженные при ассемблировании исходного модуля.

2.1.1. Основные конструкции языка ассемблера

Алфавит. Алфавит допустимых символов языка ассемблера включает в себя прописные и строчные буквы латинского алфавита, цифры и следующие специальные символы:

-|— * /=( )[ ] < >;' . ",_: @ &¤

Специальные символы и некоторые их комбинации имеют в языке особенный смысл. Кроме того, допускаются следующие непечатаемые символы: пробел, табуляция, возврат каретки и перевод строки. Если в программе встречается какой-либо символ, не входящий в алфавит допустимых символов, ассемблер считает его пробелом. Прописные и строчные буквы не отличаются друг от друга, за исключением символьных констант. Далее все ассемблерные операторы даются прописными буквами, чтобы отличить их от пояснительного текста, который дается на русском языке.

Наименьшей конструкцией исходного модуля является *идентификатор (имя)*. Чтобы отделить друг от друга два соседних имени во избежание трактовки их как одного более длинного имени, применяются *разделители*. Другими словами, разделители только отмечают окончание имени. Наиболее часто разделителем служит пробел, но как разделитель может использоваться и горизонтальная табуляция. Любой недопустимый символ или символ, используемый в недопустимом контексте, считается разделителем.

Специальные символы, которые отмечают конец имени и имеют особенный смысл, называются *ограничителями*. Например, в ассемблерной команде

```
ADD  AX, [BX+3]
```

ограничителями будут запятая, знак плюс и квадратные скобки. При наличии ограничителей разделители не обязательны, но их введение часто улучшает читабельность программы.

В табл. 2.1 приведены специальные значения ограничителей и разделителей, принятые в большинстве версий языка ассемблера МП К1810.

Примечание. В литературе по вычислительной технике отечественных и особенно зарубежных авторов имеется неоднозначность трактовки термина «символ». В англоязычной литературе термин symbol имеет смысл только «символического имени» и почти аналогичен терминам name (имя) и identifier (идентификатор). Но, очевидно по созвучию с эквивалентным русским происхождением и по одинаковости написания,

Таблица 2.1. **Смысловое содержание специальных символов**

Обозначение	Название	Функция
SP (20H)	Пробел(ы)	Разделение и завершение имен
HT (09H)	Табуляция	Разделение и завершение имен
,	Запятая	Разделение операндов
'...'	Апострофы	Ограничение символьной цепочки
(...)	Круглые скобки	Ограничение выражения или его части
CR (0DH)	Возврат каретки	Терминатор
LF (0AH)	Перевод строки	оператора (если нет последующего символа &)
CR-LF	Возврат каретки и перевод строки	
;	Точка с запятой	Начало комментария
:	Двоеточие	Ограничитель меток, замены сегментов, спецификаторов макрокоманд, директив EXTRN и ASSUME, полей записи
.	Точка	Разделитель полей записи
&	Амперсанд	Указатель строки продолжения
<...>	Угловые скобки	Внутренние значения используются для инициализации записи в определениях макрокоманд
$	—	Текущее значение счетчика ячеек (адресов) в ассемблере
[...]	Квадратные скобки	Косвенная адресация
=	Знак равенства	Отделяет спецификацию ширины поля от начального значения по умолчанию
—	Знак минуса	Бинарное вычитание, унарный минус
+	Знак плюса	Бинарное сложение, унарный плюс
*	Звездочка	Умножение
/	Знак деления	Деление
?	Вопросительный знак	Неопределенное значение для инициализации, символ в именах
@	Знак «коммерческого эт»	Символ в именах
_	Подчеркивание	Символ в именах

термин symbol часто употребляется как «символ». В то же время термин character, обозначающий элемент алфавита, часто употребляется как «знак». Однако термин «знак» (sign) зарезервирован для обозначения знаков плюс и минус. С учетом сказанного в данной книге термин «символ» обозначает только элемент алфавита, т. е. букву, цифру, или специальный символ.

Операторы. Исходный модуль представляет собой последовательность *операторов* (или *предложений*) языка ассемблера. Операторы обычно занимают по одной строке, под которой понимается последовательность символов, заканчивающаяся возвратом каретки, переводом строки или их комбинацией. Исключение составляет только так называемая *строка продолжения*, в первом столбце (символьной позиции) которой находится символ &. Такую строку ассемблер считает частью (своеобразным переносом) предыдущей строки.

Ассемблер воспринимает операторы в свободном формате, т. е. элементам операторов не назначены фиксированные столбцы и между ними может быть любое число пробелов там, где может быть один пробел. Другими словами, любая непрерывная последовательность пробелов рассматривается как один пробел. Свободный формат сокращает число ошибок и улучшает читабельность программ. Ниже приведены примеры операторов языка ассемблера.

```
        MOV     CX, ARRAY[SI]   ; Типичный командный оператор
MOV             CX, ARRAY[SI]   ; Эквивалентен предыдущему
        MOV     CX,             ; Допустимый, но необычный
&               ARRAY[SI]       ; формат оператора
TRUE    EQU     0FFH            ; Директива ассемблера
CODE    SEGMENT                 ; Директива ассемблера
WVAL    DW      0               ; Инициализация памяти
NEXT:   ADD     AL, DH          ; Отмеченный оператор
        MOV     ES:ROW[DI], BL  ; Замена сегмента
WAIT:   LOCK    XCHG AL,SEMA    ; Префикс блокировки

; Это примеры операторов языка ассемблера
```

Операторы в исходном модуле классифицируются как *командные операторы, операторы распределения данных* и *директивы ассемблера*, но иногда операторы распределения данных относят к директивам.

Командные операторы определяют генерируемые ассемблером машинные команды; они содержат *мнемонику* и (необязательно) один или два *операнда*. Каждый командный оператор порождает одну машинную команду, формат которой зависит от способа задания операнда (или операндов). Ассемблер допускает изменение, добавление и переопределение мнемоник.

Операторы распределения данных резервируют ячейки памяти для программных данных.

Директивы ассемблера, называемые также *псевдокомандами*, содержат специальные указания для ассемблирующей программы. Директивы отличаются от операторов первых двух классов тем, что они не определяют содержимого памяти.

2.1.2. Формат операторов

Командные операторы записываются в формате, который стал стандартным для языков ассемблера большинства современных ЭВМ, за исключением поля префикса, характерного только для языка ассемблера МП К1810:
{Метка:} {Префикс} Мнемоника {Операнд(ы)} {;Комментарий} Здесь фигурные скобки обозначают необязательные поля. Рассмотрим содержание отдельных полей данного формата.

Метка. *Метка* представляет собой определяемое пользователем имя, заканчивающееся двоеточием. Значением метки является текущее значение *счетчика ячеек (адресов)* в текущем сегменте кода. Счетчик ячек — это главная переменная процесса ассемблирования, значение которой в известном смысле эквивалентно содержимому программного счетчика PC при выполнении программы. Поэтому значением метки является адрес отмеченной команды. Метки как операнды фигурируют только в командах передачи управления.

Префикс. Задание префикса заставляет ассемблер сформировать один из префиксных байт — блокировки LOCK или повторения REP, который непосредственно предшествует команде. Префикс замены сегмента определяется несколько иначе (см. § 3.5).

Мнемоника или код операции (КОП). Мнемоника представляет собой заранее определенное и неизменяемое имя, которое является частью словаря языка ассемблера и идентифицирует тип генерируемой машинной команды. В качестве мнемоник используются сокращенные или полные английские слова, передающие смысл основной функции команды: ADD — сложить, SUB (*tract*) — вычесть, XCHG (*eXCHanGe*) — обменять и т. д.

Операнд(ы). Мнемоника команды может потребовать следования за нею других имен, являющихся объектами операции, т. е. операндов. В зависимости от функции команды она требует одного, двух или не требует ни одного операнда, например:

```
MOV     AX, STRING [SI]     ; Два операнда
NOT     DL                  ; Один операнд
XLAT                        ; Нет операндов
```

При необходимости указания более двух операндов программист пользуется *макрокомандами*. Все операнды, следующие за первым, должны отделяться от предшествующих запятыми.

Комментарий. Точка с запятой всегда (за исключением случая использования ее в символьной цепочке) идентифицирует начало комментария. Заканчивается комментарий символом завершения строки. Комментарии не влияют на выполнение программы, так как при трансляции ассемблер игнорирует это поле, но сохраняет комментарии в листинге. Комментарии предназначены только для целей документирования программ и пояснения намерений программиста, если они не очевидны из команды.

Директивы ассемблера и операторы распределения данных имеют несколько иной формат:

{Имя} Директива {Операнд(ы)} {;Комментарий}

Рассмотрим назначение отдельных полей этого формата.

Имя директивы имеет совершенно другой смысл по сравнению с меткой, поэтому оно никогда не заканчивается двоеточием. Некоторые директивы требуют обязательного наличия имени, например SEGMENT, ENDS, PROC, GROUP и др. В других директивах поле имени должно быть пустым, например NAME, ASSUME, ORG, PUBLIC и др. В операторах распределения данных DB, DW и DD имя вводится программистом по его усмотрению, т. е. является необязательным. Директива определения макрокоманды представляет собой исключение из приведенного формата (см. ниже).

Директива. Поле директивы, аналогичное полю мнемоники в командных операторах, содержит одно из ключевых неизменяемых слов ассемблера и определяет его действия

в процессе ассемблирования. Директивы ассемблера используются программистом для распределения памяти, обеспечения связей между модулями манипуляций с символическими именами.

Операнд(ы). Операнды директив аналогичны операндам командных операторов. В некоторых директивах, например DB, DW, PUBLIC, допускаются списки операндов. Операнды других директив, например SEGMENT, PROC, являются ключевыми словами, назначающими определенные атрибуты определяемым объектам. В этом случае операнды необязательны, т. е. допускается задание атрибутов по умолчанию.

Комментарий. Поле комментария в директивах полностью соответствует такому же полю в командных операторах.

Директива определения макрокоманды, как уже отмечалось, имеет формат, отличающийся от формата остальных директив:

CODEMACRO Имя {Операнд(ы)} {;Комментарий}

Отметим, что имя макрокоманды находится после ключевого слова CODEMACRO.

В языке имеются три пары взаимосвязанных директив, требующих обязательно согласованного употребления:

директивы SEGMENT (сегмент) и ENDS (конец сегмента);

директивы PROC (процедура или подпрограмма) и ENDP (конец процедуры);

директивы CODEMACRO (начало определения макрокоманды) и ENDM (конец определения макрокоманды).

В парных директивах SEGMENT/ENDS и PROC/ENDP имя, находящееся в первой директиве пары, должно быть указано и во второй директиве. Для директив CODEMACRO/ENDM это требование необязательно.

2.1.3. Элементы операторов

Операторы исходного модуля состоят из ключевых слов, идентификаторов, численных констант, символьных цепочек, специальных символов и комментариев.

Ключевые слова. Ключевые или зарезервированные слова представляют собой имена, имеющие для ассемблера вполне определенный смысл. Примерами ключевых слов служат, например, мнемоники команд и директив — ADD, MOV, SEGMENT, PROC и др.

Идентификаторы. Идентификатор как общий термин для **меток** и имен переменных — это определяемая программистом последовательность символов. Первым символом в последовательности должна быть буква или один из символов

@ , _ и ?, но один вопросительный знак не может быть

идентификатором. Каждым последующим символом может быть любой из указанных выше символов или цифра. Ассемблер обычно допускает идентификаторы произвольной длины, но различает их только по *n* первым символам, причем *n* зависит от версии языка ассемблера. Целесообразно придавать идентификаторам содержательный смысл. Ниже приведены примеры допустимых и недопустимых идентификаторов:

ARRAY TOP_OF_STACK MAIN_MODULE	— допустимые идентификаторы
FIRST [STREET]	— скобки не разрешены
'STRING'	— это символьная цепочка
5ABC	— начинается с цифры

Язык ассемблера МП К1810 относится к классу жестко типизированных языков. Это означает, что операнды команд (регистры, переменные, метки, константы) имеют ассоциированный с ними атрибут типа, который сообщает ассемблеру некоторую информацию об операнде. Например, операнд 20 имеет тип NUMBER (число) и сообщает ассемблеру, что это численная константа, а не регистр или адрес ячейки памяти.

Регистры. Ассемблер определяет для программиста набор внутренних регистров и один из сегментов. При желании имена регистров можно переопределить, хотя почти всегда используются имена, приведенные в программной модели микропроцессора. Напомним, что в МП имеются следующие регистры:

общие

тип BYTE — регистры AL, AH, BL, BH, CL, CH, DL, DH;

тип WORD — регистры AX, BX, CX, DX, BP, SP, SI, DI;

сегментные

тип WORD — регистры CS, DS, SS, ES.

Триггеры регистра флажков представляют собой однобитные регистры.

Переменные. Переменная — это единица программных данных, имеющая символическое имя. Переменные подробно рассматриваются в § 3.2, а ниже приведены только общие положения.

Большинство ассемблерных программ начинается с определения данных, которыми они будут оперировать. Распределение и именование ячеек памяти осуществляется с помощью директив DB, DW и DD, которые определяют соответственно переменные длиной в байт, слово и двойное слово. Операнды этих директив сообщают ассемблеру, сколько единиц памяти необходимо распределить и как их инициализировать, если заданы начальные значения. Примеры определения перемеиных приведены ниже:

```
G_ SEG          SEGMENT
   ALPHA        DB      ?           ; Не инициализируется, тип BYTE
   BETA         DW      ?           ; Не инициализируется, тип WORD
   GAMMA        DD      ?           ; Не инициализируется, тип DWORD
   DELTA        DB      ?           ; Не инициализируется, тип BYTE
   EPS          DW      0AH         ; Значение 000A, тип WORD
G_ SEG          ENDS
R_ SEG          SEGMENT AT 100H     ; Базовый адрес сегмента
   IOTA         DB      'GORKY'     ; Символьная цепочка, тип BYTE
   KAPPA        DW      'AB'        ; Инициализируются 2 байта
   LAMBDA       DD      R_SEG       ; Содержит 0000 0100, тип DWORD
   MU           DB      100 DUP(?)  ; 100 байт без значений
R_ SEG          ENDS
```

Любая перемеиная имеет три атрибута: сегмент, смещение и тип. Сегмент (SEG) идентифицирует сегмеит, содержащий переменную, т. е. тот сегмент, который ассемблируется при определении переменной. Смещение (OFFSET) представляет собой расстояние или дистанцию в байтах переменной от начала (базы) содержащего его сегмента. Диапазон смещений составляет 0 — 65535. Тип (TYPE) идентифицирует единицу памяти, выделяемую для хранения переменной. Допустимые значения типа (число байт) 1 (байт), 2 (слово) и 4 (двойное слово). Атрибуты переменных приведенного выше примера показаны в табл. 2.2.

Таблица 2.2. Атрибуты переменных

Переменная	Атрибуты			Операторы	
	Сегмент	Смещение	Тип	Длина	Размер
ALPHA	G_SEG	0	1	1	1
BETA	G_SEG	1	2	1	2
GAMMA	G_SEG	3	4	1	4
DELTA	G_SEG	7	1	1	1
EPS	G_SEG	8	2	1	2
IOTA	R_SEG	0	1	5	5
KAPPA	R_SEG	5	2	1	2
LAMBDA	R_SEG	7	4	1	4
MU	R_SEG	11	1	100	100

Смысл операторов LENGTH (длина) и SIZE (размер) рассматривается в § 3.2.

Когда переменная находится в поле операнда командного оператора, ассемблер использует ее атрибуты для определения формата генерируемой машинной команды. При несоответствии атрибутов переменной ее использованию в команде ассемблер фиксирует ошибку. Например, попытка прибавить переменную типа WORD к байтному регистру будет отмечена как ошибка.

Метки. Метка представляет собой имя, относящееся к ячейке программной памяти, и предназначена для использования как операнда в командах передачи управления. Каждая метка имеет четыре атрибута. Атрибуты сегмента и смещения аналогичны соответствующим атрибутам переменной. Расстояние (дистанция) определяет возможность «достижения» метки командой передачи управления с помощью двухбайтного смещения (этот атрибут похож на тип переменной). Если это возможно, то говорят, что метка имеет тип NEAR (близкая, т. е. находящаяся внутри текущего сегмента кода), и команда должна модифицировать только содержимое программного счетчика PC. В противном случае, т. е. когда метка находится вне текущего сегмента кода, значением типа метки является FAR (далекая), и передающая ей управление команда должна иметь 4-байтный указатель сегмент:смещение и модифицировать содержимое регистров PC и CS. Наконец, последним ат-

рибутом метки, тесно связанным с предыдущим, является предположение о сегментном регистре CS. Оно показывает, что ожидалось найти в регистре CS, когда была определена метка. Если при всех обращениях к данной метке используется одно и то же предположение о регистре CS, то атрибут расстояния метки должен указывать тип NEAR, а в противном случае — тип FAR. По умолчанию принимается тип NEAR.

Тип NEAR сообщает ассемблеру, что все передачи управления данной метке могут достигать ее с помощью 16-битного смещения. Наряду с этим тип FAR означает, что вместе со смещением необходимо указывать еще и второе слово — сегмент ячейки. При обнаружении обращения, которое требует далекой (FAR) передачи управления метке, имеющей тип NEAR, ассемблер формирует сообщение об ошибке.

Идентификация меток производится по следующим правилам:

наличие перед командой имени, заканчивающегося двоеточием, например:

```
NEXT:   SUB     CX, SI
AGAIN:  MOV     AX, ALPHA[DI]
```

использование в поле имени директивы LABEL:

```
REPEAT  LABEL   FAR
```

указание в поле имени директивы EQU:

```
MORE    EQU     THIS NEAR
```

использование в поле имени пары директив PROC/ENDP:

```
SUMMA   PROC    NEAR
        .
        .
        .
SUMMA   ENDP
```

Числа. В языке ассемблера можно определять имена для представления численных значений, а не указателей регистров или ячеек памяти. При использовании таких имен они интерпретируются ассемблером так, как будто программист явно указывает представляемые ими числа. Например, конструкция

```
CR      EQU     0DH
        MOV     AL, CR
```


эквивалентна записи команды

```
        MOV     AL, 0DH
```

Таким образом, CR определяется как имя, заменяющее 16-ричное число 0D.

Отметим, что значением числа могут быть цепочки из одного или двух символов. В первом случае вместо символа подставляется 1 байт, соответствующий кодированию символа в применяемом алфавите, а во втором случае — 2 байта. Например, фрагмент

```
INIT    EQU     'AA'
        MOV     CX, INIT
```

вызовет загрузку 16-ричного числа 4141 в регистр CX (буква A имеет двоичный код 01000001).

Другие имена. В языке ассемблера имеется ряд директив, которые требуют наличия имени в соответствующем поле. К ним относятся следующие директивы:

SEGMENT и ENDS, определяющие имя сегмента;

GROUP, определяющая имя группы и объединяемые в ней сегменты;

RECORD, определяющая имя записи и имена всех составляющих ее полей;

CODEMACRO, определяющая имена макрокоманды и всех ее формальных параметров; отметим, что определенное таким образом имя макрокоманды не является обычным именем, а превращается в мнемокод команды;

EQU, определяющая переменные и метки и используемая также для присваивания имен выражениям.

Численные константы. Константа — это численное значение, вычисляемое во время ассемблирования из имеющегося выражения. Константа отличается от адреса памяти (переменная или метка) тем, что она определяет чистое число. Константы имеют тип NUMBER (число).

Численные константы допускается представлять в системах счисления с основаниями 2, 8, 10 и 16. За младшей цифрой должен находиться однобуквенный дескриптор системы счисления: B — двоичная, O или Q — восьмеричная, D (необязательно) — десятичная, H — 16-ричная. В последнем случае число, начинающееся с «буквенной цифры» A—F, должно быть дополнено слева незначащим нулем. Диапазон всех чисел соответствует диапазону 16-битных двоичных чисел (включая знаковый бит). Отрицательные

числа представляются в дополнительном коде. Ниже приведены примеры задания численных констант:

```
MASK    EQU     00001111B       ; Двоичная константа
OCTAL   EQU     17O             ; Две восьмеричные константы
OCTAL   EQU     17Q

DECIM   EQU     15D             ; Две десятичные константы
DECIM   EQU     -15
HEX     EQU     0FFH            ; 16-ричная константа
```

Напомним, что в качестве численных констант допускается использовать одно- и двухсимвольные цепочки. Первая из них имеет тип BYTE, а вторая тип WORD.

Символьные цепочки. Символьные цепочки или константы заключаются в апострофы (одинарные кавычки) и обычно имеют максимальную длину до 255 символов. В них допускаются символы пробела и горизоитальной табуляции, однако символы возврата каретки и перевода строки использовать нельзя. При необходимости включения в цепочку апострофа он представляется двумя соседними символами. Примеры символьных цепочек приведены ниже.

```
MESSAGE     DB      'RANGE ERROR'
REPLY       DW      'NO'
HEADING     DB      'OS CP/M VERSION 2.2'
```

Символьные цепочки, содержащие более двух символов, применяются только для инициализации памяти. Ассемблер представляет цепочку в виде последовательности байт, соответствующих кодированию символов цепочки.

2.2. Режимы адресации

При выполнении любой программы процессор обращается к памяти, в которой хранятся команды и данные. В командах преобразований данных определяются адреса, которые указывают местоположение необходимых данных, а в командах передачи управления определяются адреса команд, которым передается управление, т.е. адреса переходов. Способ, или метод определения в команде адреса операнда или адреса перехода, называется режимом адресации или просто адресацией.

В наиболее простом режиме адресации, называемом прямой адресацией, адрес находится в самой команде. Однако использование этого режима, хотя и предусмотрено в

большинстве современных процессоров, приводит к чрезмерной длине команд, особенно в условиях постоянно увеличивающейся емкости памяти. Поэтому в настоящее время в процессорах применяется много других режимов адресации, направленных на достижение следующих целей:

определение адреса памяти наименьшим числом бит, что сокращает длину команд;

вычисление адреса относительно текущей команды (так называемая относительная адресация), обеспечивающее загрузку программ без модификаций в любую область памяти;

осуществление доступа к ячейкам памяти, адреса которых вычисляются при выполнении программы, что упрощает доступ к регулярным структурам данных;

оперирование адресами в такой форме, которая наиболее удобна для таких структур данных, как массивы и сетки [1].

Для современных процессоров разработано более двух десятков режимов адресации, которые в той или иной степени удовлетворяют приведенным выше требованиям. Режимы адресации значительно расширяют гибкость и удобство пользования системой команд.

Назначением режима адресации является указание способа формирования эффективного (или исполнительного) адреса ЕА. Этот адрес является либо адресом операнда (в командах, оперирующих данными), либо адресом перехода (в командах передачи управления). В МП К1810 эффективный адрес памяти представляет собой 16-битное беззнаковое целое, являющееся смещением относительно базы некоторого сегмента. Полный (физический) адрес памяти формируется с привлечением одного из сегментных регистров, как это было рассмотрено в § 1.5.

Режимы адресации подразделяются на прямые и косвенные. При прямой адресации эффективный адрес либо содержится в команде, либо вычисляется с использованием значения, находящегося в команде, и содержимого, указанного в команде регистра (или двух регистров). При косвенной адресации эффективный адрес в команде определяет регистр или ячейку памяти, содержащую окончательный эффективный адрес. Большинство современных процессоров допускает только один уровень косвенности, хотя принципиально этот уровень может быть любым.

Как уже отмечалось, МП К1810 имеет организацию типа «регистр — память». С точки зрения адресации это ор-

начает, что его команды адресуют максимум два операнда и что не допускается одновременная адресация двух ячеек памяти. Первым операндом в двухоперандной (или двухадресной) команде обычно является содержимое регистра или ячейки памяти, а вторым — содержимое регистра или непосредственный операнд в команде. Ниже будет показано, что приведенная нумерация («первый» и «второй» операнды) в МП К1810 является довольно условной и при желании может быть изменена.

Наиболее общий формат двухоперандной команды приведен на рис. 2.1 (штриховыми линиями обозначены необя-

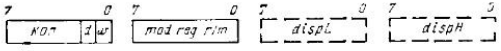


Рис. 2.1. Двухоперандная команда (второй операнд — регистр)

зательные байты команды). Первый байт команды содержит код операции КОП и два однобитных поля — направления d и слова w. Поле d определяет направление передачи: если $d=1$, то направление в МП, а если $d=0$, то направление из МП. Само направление относится ко второму операнду — регистру, определяемому полем *reg* второго байта команды. Этот байт называется постбайтом (или просто байтом) режима адресации. Поле w идентифицирует тип (размер) операнда: если $w=1$, команда оперирует словом, а если $w=0$, — байтом. Таким образом, в зависимости от значений полей d и w имеются четыре возможности:

d	w	Передача или операция
0	0	Байт из регистра в память или регистр
0	1	Слово из регистра в память или регистр
1	0	Байт в регистр из памяти или регистра
1	1	Слово в регистр из памяти или регистра

Участвующие в операции регистры или регистр и ячейку памяти идентифицирует постбайт, состоящий из трех полей: *mod* — режим, *reg* — регистр и *r/m* — регистр/память. Поле *reg* определяет второй операнд, обязательно находящийся в регистре. Способ кодирования внутренних регистров МП в поле *reg* представлен в табл. 2.3.

Из табл. 2.3 видно, что регистры данных AX — DX, а также указательные и индексные регистры SP, BP, SI, DI адресуются одинаковым образом. Данное обстоятельство

Таблица 2.3. **Адресация внутренних регистров МП**

Поле *reg* (и *r/m*)	8-битные регистры	16-битные регистры
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

подчеркивает правомерность объединения всех этих регистров в группу общих регистров.

Поле *mod* определяет используемый режим адресации. В частности, оно показывает, как интерпретируется поле *r/m* для нахождения первого операнда. Если *mod*=11, операнд содержится в регистре, а в остальных случаях первый операнд находится в памяти.

Когда поле *mod* содержит 11 (регистровая адресация), поле *r/m* определяет 8- или 16-битный регистр в соответствии с табл. 2.3.

В остальных случаях адресуется память, и поле *mod* определяет, как используется (необязательное) смещение *disp*, находящееся во втором и третьем байтах команды:

mod=
- 00, *disp*=0 — смещение отсутствует;
- 01, *disp*=*dispL* — команда содержит 8-битное смещение, которое расширяется со знаком до 16 бит;
- 10, *disp*=*displH*, *dispL* — команда содержит два байта смещения.

В этом же случае (*mod*≠11) косвенной адресации поле *r/m* определяет, каким образом формируется эффективный адрес операнда. Кодирование поля *r/m* представлено в табл. 2.4.

Приведенные правила имеют только одно исключение: если *mod*=00 и *r/m*=110, то EA=*dispH*, *displ*. Здесь в команде содержится абсолютный адрес памяти.

Таким образом, операнд в памяти можно адресовать прямо (16-битное смещение) или косвенно (с 8- или 16-битным смещением). Во втором случае память можно адресовать через базовый регистр (BP или BX), через индексный регистр (SI или DI), а также через комбинацию базового и индексного регистров. Всего получается 24 ре-

Таблица 2.4. **Формирование эффективного адреса памяти**

Поле *r/m*	Эффективный адрес
000	EA=(BX)+(SI)+*disp*
001	EA=(BX)+(DI)+*disp*
010	EA=(BP)+(SI)+*disp*
011	EA=(BP)+(DI)+*disp*
100	EA=(SI)+*disp*
101	EA=(DI)+*disp*
110	EA=(BP)+*disp*
111	EA=(BX)+*disp*

жима адресации памяти — три способа интерпретации поля *mod* и восемь способов интерпретации поля *r/m*.

Суммарные сведения о постбайтных режимах адресации МП К1810 сведены в табл. 2.5 и представлены на рис. 2.2.

Таблица 2.5. **Постбайтные режимы адресации**

r/m	*mod*					
	00	01	10	11		
				w=0	*w*=1	
000	(BX)+(SI)	(BX)+(SI)+D8	(BX)+(SI)+D16	AL	AX	
001	(BX)+(DI)	(BX)+(DI)+D8	(BX)+(DI)+D16	CL	CX	
010	(BP)+(SI)	(BP)+(SI)+D8	(BP)+(SI)+D16	DL	DX	
011	(BP)+(DI)	(BP)+(DI)+D8	(BP)+(DI)+D16	BL	BX	
100	(SI)	(SI)+D8	(SI)+D16	AH	SP	
101	(DI)	(DI)+D8	(DI)+D16	CH	BP	
110	(D16)	(BP)+D8	(BP)+D16	DH	SI	
111	(BX)	(BX)+D8	(BX)+D16	BH	DI	

D8 — *dispL* (однобайтное смещение)

D16 — *dispH*, *dispL* (двухбайтное смещение)

Примечание. Необходимо отчетливо представлять смысловое различие двух употреблений термина «смещение». Смещение (*displacement*), содержащееся в команде, интерпретируется как знаковое целое, которое участвует в формировании операционным устройством эффективного адреса EA. С другой стороны, из-за сегментной организации памяти весь эффективный адрес является смещением (*offset*) относительно базового адреса сегмента и интерпретируется как беззнаковое целое. Физический адрес памяти формирует шинный интерфейс.

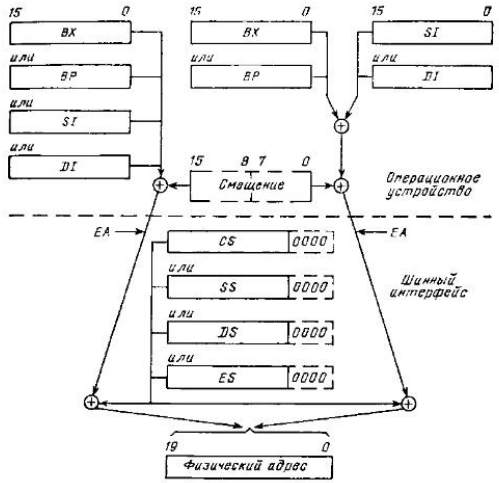


Рис. 2.2. Вычисление адреса памяти

В двухоперандной команде с непосредственным операндом, формат которой приведен на рис. 2.3, второй операнд адресовать не нужно, поэтому поле *reg* постбайта режима адресации используется как расширение кода операции.

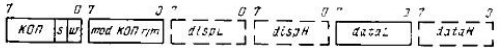


Рис. 2.3. Двухоперандная команда (второй операнд — константа)

Кроме того, здесь не нужен бит *d*, так как результат можно поместить только на место первого операнда. Но зато в этом формате необходимо определить размер (тип) непосредственного операнда. Для этой цели служат биты *s* и *w*, интерпретируемые следующим образом:

$$s:w=\begin{cases} \text{X0, один байт данных } dataL; \\ \text{01, два байта (слово) данных } dataH,\ dataL; \\ \text{11, один байт данных, который расширяется со знаком до 16 бит.} \end{cases}$$

Поля *mod* и *r/m* интерпретируются так же, как в предыдущем формате.

Наконец, на рис. 2.4 показан типичный формат однооперандной команды. Ничего нового в этом формате по сравнению с рассмотренными выше нет.

Следует отметить, что в МП К1810 есть специальные (хотя и избыточные) форматы, которые позволяют сокра-

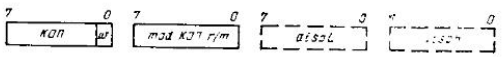


Рис. 2.4. Однооперандная команда

тить на 1 байт длину часто используемых команд. В основном они относятся к операциям с регистрами и особенно с аккумуляторами AL и AX. Примерами таких операций являются включение в стек содержимого регистра, извлечение из стека в регистр, передача непосредственных данных в регистр, инкремент и декремент содержимого регистра, арифметические операции непосредственных данных с аккумулятором. На рис. 2.5, *а* показан стандартный формат

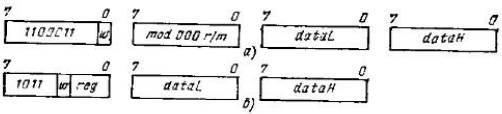


Рис. 2.5. Стандартный и специальный форматы команд

команды передачи непосредственного операнда в регистр или память, а на рис. 2.5, *б* — специальный, более короткий формат передачи непосредственного операнда в регистр. Необязательные байты смещения *disp* на рис. 2.5 не приведены. Программа-ассемблер автоматически выбирает более короткую команду, и программист может не помнить о специальных форматах команд. Однако знание их необ-

ходимо, когда при отладке программ приходится разбираться в распечатках (дампах) памяти. Более подробно форматы команд описаны в § 2.3.

Рассмотрим стандартные режимы адресации МП К1810 с учетом приведенных выше особенностей формирования эффективного адреса.

Регистровая адресация. При использовании регистровой адресации операнд находится в одном из общих регистров МП, а в некоторых командах — в одном из сегментных регистров. Регистр может быть определен в байте кода операции или в постбайте режима адресации. В двухоперандных командах определяются два регистра. Регистры идентифицируются 3-битными полями *reg* и *r/m* (когда *mod*==11). В зависимости от значения бита *w* в операции участвует 8- или 16-битный регистр (или регистры).

Команды, оперирующие содержимым регистров, оказываются наиболее короткими и выполняются за минимальное время, так как не требуют цикла шины для обращения к памяти.

В ассемблерных программах регистры обозначаются зарезервированными именами: AL, AH, AX, BL, BH, BX и т. д.

Примеры ассемблерных команд с регистровой адресацией приведены ниже.

```
MOV     AX, SI      ; Передать содержимое SI в AX
ADD     DI, BX      ; Прибавить содержимое BX к DI
AND     CL, AX      ; Ошибка — несоответствие разме-
                    ; ров
XOR     AL, AH      ; Исключающее ИЛИ регистров
                    ;   AL и AH
```

Непосредственная адресация. Непосредственные операнды (называемые иногда «литералами») представляют собой константы длиной 8 или 16 бит, содержащиеся в командах. Доступ к таким операндам осуществляется очень быстро, так как они находятся во внутренней очереди команд и циклов шины для считывания операндов из памяти не требуется. Непосредственные операнды изменить в ходе выполнения команды невозможно.

Данный режим предусмотрен для большинства двухоперандных команд. Наличие в командах постбайта режима адресации позволяет манипулировать непосредственными операндами и содержимым регистров или ячеек памяти. Однако в МП нет команд непосредственной загрузки сегментных регистров и включения константы в стек, что вы-

зывает некоторые неудобства при программировании. Стандартный непосредственный операнд имеет длину 16 бит, а короткий — 8 бит (при необходимости он расширяется со знаком до 16 бит).

Константы в командах применяются для нескольких целей, например для инициализации регистров и переменных в памяти, в качестве масок для манипуляций отдельными битами, для сравнения переменных с граничными значениями и т. д.

Непосредственные операнды находятся в конце формата команд после смещения, если оно имеется, причем первым следует младший байт константы.

Примеры записи команд с непосредственными операндами на языке ассемблера приведены ниже.

```
SUB     AL, 30H         ; Вычесть из содержимого AL число 48
MOV     CL, 10          ; Загрузить в регистр CL число 10
AND     AX, 0F000H      ; Выделить старшие 4 бита AX
XOR     DH, 1           ; Инвертировать младший бит DH
CMP     BL, 40H         ; Сравнить содержимое BL с числом 64
```

Прямая адресация. Прямая адресация представляет собой простейший режим адресации — эффективный адрес EA берется прямо из поля смещения команды, и никакие регистры для его вычисления не привлекаются. Этот режим применяется для обращения к простым переменным, которые называются также скалярами.

Примеры ассемблерных комаид с прямой адресацией приведены ниже.

```
MOV     AX, GAMMA       ; Загрузить в AX переменную GAMMA
ADD     TEMP, BL        ; Прибавить (BL) к переменной TEMP
```

Разновидностью данного режима является длинная прямая адресация, в которой команда содержит базовый адрес сегмента и смещение. В этом случае обеспечивается прямой доступ к операнду с любым логическим (и физическим) адресом. Однако длинная прямая адресация допускается только в командах межсегментных переходов и вызовов и не может применяться в командах, оперирующих данными. Метки, которым передают управление команды с данным режимом адресации, должны иметь тип FAR.

Косвенная регистровая адресация. В этом режиме эффективный адрес EA операнда находится в одном из базовых или индексных регистров (рис. 2.6). Из четырех адресных регистров BP, BX, SI и DI в косвенной адресации могут использоваться только регистры BX, SI и DI. Косвенная адресация через указатель базы BP моделируется при помощи базовой адресации с нулевым смещением (см. ниже).

Данный режим с некоторыми вариантами применяется во всех современных процессорах. Он позволяет вычислять адреса во время выполнения программы, что часто требуется, например, для обращения к различным элементам ре-

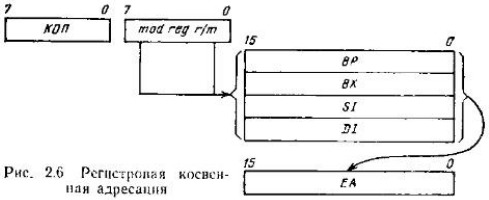


Рис. 2.6 Регистровая косвенная адресация

гулярных структур данных. При модификации содержимого регистра одна и та же команда оперирует различными ячейками памяти. Для изменения содержимого регистра применяется команда загрузки эффективного адреса LEA, а также все арифметические и логические команды. Отметим, что в командах безусловного перехода и вызова подпрограммы CALL с регистровой косвенной адресацией допускается указывать любой 16-битный общий регистр.

Примеры записи команд с косвенной регистровой адресацией на языке ассемблера приведены ниже.

```
ADD     AX, [DI]        ; Прибавить к AX содержимое ячейки
                        ; памяти, адресуемой DI
MOV     [SI], CL        ; Передать (CL) в ячейку памяти, адресуемую SI
MOV     DI, [DI]        ; Передать в DI содержимое ячейки памяти, адресуемой DI
JMP     AX              ; Перейти по адресу из AX
CALL    BX              ; Вызвать подпрограмму по адресу из BX
```

Базовая адресация. При базовой адресации (называемой также адресацией по базе или адресацией типа «база плюс смещение») эффективный адрес операнда является суммой значения смещения и содержимого регистров BX

или BP (рис. 2.7). Напомним, что при определении BP в качестве базового адреса шинный интерфейс обращается к операнду в текущем сегменте стека (если нет префикса замены сегмента). Это делает базовую адресацию с регистром BP очень удобным средством обращения к данным, находящимся в стеке, что требуется, например, при передаче подпрограммам параметров через стек.

Основное применение базовой адресации связано с доступом к элементам структур данных, когда смещение (номер) элемента известно при ассемблировании программы, а базовый (начальный) адрес структуры должен вычисляться при выполнении программы. Таким образом, структура данных может размещаться в различных областях памяти, а модификация содержимого базового регистра обеспечивает доступ к этим областям. Базовый регистр указывает на начало структуры данных, а требуемый элемент адресуется с помощью смещения (расстояния) от базы (рис. 2.8). Смещения, содержащиеся в команде, могут иметь длину 8 или 16 бит и интерпретируются как знаковые целые. Размер смещения программа-ассемблер выбирает автоматически с учетом атрибутов операндов.

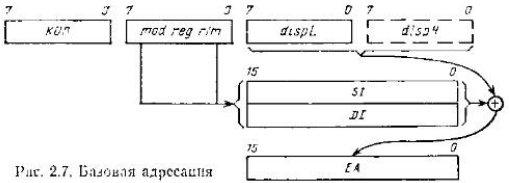


Рис. 2.7. Базовая адресация

В языке ассемблера используются два обозначения базовой адресации. Первое обозначение имеет вид [BREG] DISP и соответствует адресации структуры данных типа «запись» в языке ПАСКАЛЬ. Второе обозначение имеет вид [BREG+DISP] и явно указывает на необходимость суммирования содержимого регистра и смещения при вычислении эффективного адреса. В обоих случаях обозначение BREG подразумевает один из базовых регистров.

Примеры записи команд с базовой адресацией на языке ассемблера приведены ниже.


```
MOV     AX, [BP]10      ; Обе команды передают шестое
MOV     AX, [BP+10]     ;     слово массива, адресуемо-
                        ;     го BP
ADD     [BX]TEMP, CX    ; Прибавить CX к слову TEMP
                        ;     в массиве, адресуемом BX
```

Индексная адресация. В режиме индексной адресации (называемой также адресацией с индексированием, адресацией типа «база плюс индекс») эффективный адрес вычисляется как сумма смещения, находящегося в команде, и содержимого регистров SI или DI (рис. 2.9). Данный режим обычно применяется для обращения к элементам одномерных массивов. Смещение определяет фиксированный при ассемблировании начальный адрес массива, а значение

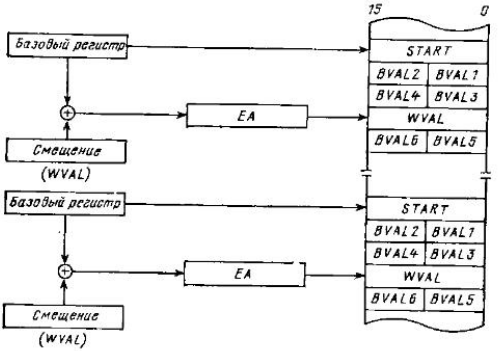


Рис. 2.8. Применение базовой адресации

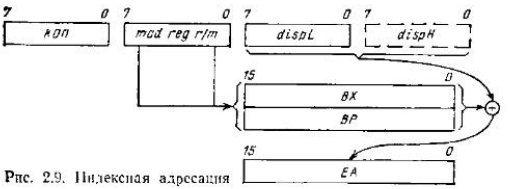


Рис. 2.9. Индексная адресация

в индексном регистре определяет нужный элемент. Так как все элементы массива имеют одинаковый размер, простые манипуляции над содержимым индексного регистра позволяют обращаться к любому элементу массива.

По существу режимы базовой и индексной адресации в МП К1810 аналогичны. Это объясняется тем, что базовые адреса и индексные значения имеют одинаковую длину 16 бит.

В языке ассемблера индексная адресация обозначается в виде ADDR16 [IREG], как это принято в языках высокого уровня. Здесь ADDR16 — 16-битное смещение, а IREG обозначает один из индексных регистров SI или DI.

Примеры команд с индексной адресацией на языке ассемблера приведены ниже.

```
MOV     ARRAY[SI], AX   ; Передать AX в элемент массива
ADD     CX, ROW[DI]     ; Прибавить к CX элемент массива
XOR     DL, MATRIX[SI]  ; Операция с элементом массива
```

Базовая индексная адресация. Данный режим называется также базово-индексной адресацией или адресацией типа «база плюс индекс плюс смещение». Здесь эффективный адрес равен сумме содержимого базового регистра, индексного регистра и смещения, находящегося в команде (рис. 2.10). Базовая индексная адресация является наиболее гибким режимом, так как два компонента адреса можно определять и варьировать при выполнении программы.

Регистры BP и BX используются как базовые, а регистры SI и DI — как индексные, т. е. всего получается четыре

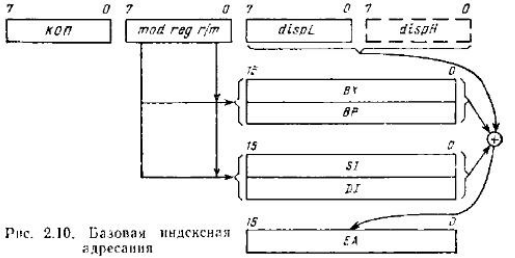


Рис. 2.10. Базовая индексная адресация

комбинации регистров. В МП К1810 данный режим расширен и допускает в командах 8- или 16-битные смещения, которые также суммируются при вычислении эффективного адреса. Смещение считается знаковым целым, представленным в дополнительном коде. При просмотре исходной программы ассемблер по выражению, находящемуся в поле операнда, определяет необходимость задания смещения и его размер.

Данный режим обеспечивает подпрограммам удобный способ адресации массива, находящегося в стеке (рис. 2.11). Регистр BP обычно адресует некоторую отсчетную

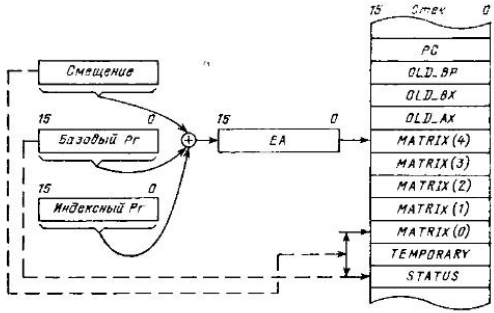


Рис. 2.11. Обращение к массиву в стеке

точку в стеке после того, как подпрограмма включила в стек содержимое внутренних регистров МП и выделила область стека для локальных переменных. В этом случае регистр BP выполняет функцию указателя стекового кадра. Расстояние начала массива от отсчетной точки представляется значением смещения, а индексный регистр предназначен для адресации отдельных элементов массива.

С помощью базовой индексной адресации возможно также обращение к элементам массива, содержащегося в матрице, т. е. двумерном массиве.

В ассемблерных программах базовая индексная адресация обозначается в виде [BREG]ADDR16[IREG], т. е. как комбинация базовой и индексной адресаций.

Примеры использования данного режима в ассемблерных программах приведены ниже.

```
ADD     [BX]ALPHA[SI], DI
CMP     [BP]BETA[DI], 4000H
SUB     [BX]GAMMA[DI], 1000
```

Примечание. Для альтернативных обозначений базовой, индексной и базовой индексной адресаций в некоторых ассемблерах следуют соглашению: если в адресе памяти фигурирует переменная (смещение), она указывается первой. В этом случае запись ассемблерных команд принимает следующий вид:

```
ADD     TEMP[BX], CX        ; Базовая адресация
MOV     AX, ARRAY[SI]       ; Индексная адресация
AND     BL, GAMMA[BP][DI]   ; Базовая индексная адре-
                            ;   сация
```

Относительная адресация. В режиме относительной адресации эффективный адрес вычисляется как сумма фиксированного смещения, находящегося в команде, и текущего значения программного счетчика PC. При этом значение PC равно адресу байта, следующего за текущей командой (предполагается, что команда с относительной адресацией считана из памяти и PC адресует следующую по порядку команду). В МП К1810 относительная адресация применяется только в командах условных и безусловных переходов, вызовов программ и управления итерациями (или циклами).

Команда содержит короткое 8-битное смещение, которое расширяется со знаком до 16 бит, а затем прибавляется к содержимому PC (рис. 2.12). Следовательно, этот режим обеспечивает передачу управления в диапазоне $-128 \div +127$ байт от текущей команды. Короткие (SHORT) переходы часто требуются в циклических программах, и на них приходится до 80 % команд передачи управления.

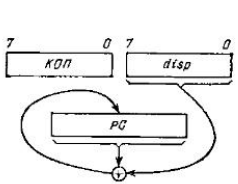


Рис. 2.12. Относительная адресация

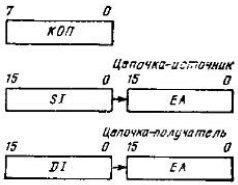


Рис. 2.13. Адресация цепочек

Следует отметить, что программист в ассемблерных программах указывает не значение смещения, а абсолютный адрес перехода, т. е. метку команды, которой необходимо передать управление. Значение смещения автоматически вычисляет программа-ассемблер.

Значение смещения не зависит от размещения программы в адресном пространстве памяти, поэтому относительная адресация обеспечивает позиционную независимость программ.

Адресация цепочек. Для обращения к операндам цепочечных команд обычные режимы адресации памяти не применяются. Вместо этого неявно используются индексные регистры, как это показано на рис. 2.13. При выполнении цепочечной команды предполагается, что регистр SI адресует первый байт (слово) цепочки-источника, а регистр DI — первый байт (слово) цепочки-получателя. В повторяющихся цепочечных операциях МП автоматически корректирует содержимое регистров SI и DI по мере перехода к следующим элементам цепочек.

Адресация портов ввода-вывода. Если порты ввода-вывода отображены на память, то для обращения к ним применяются любые режимы адресации операндов, находящихся в памяти. В этой же ситуации возможно использование цепочечных команд при наличии соответствующего аппаратного интерфейса.

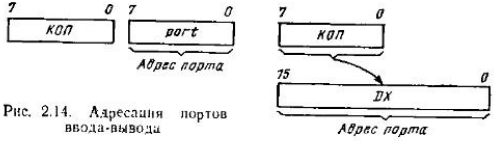


Рис. 2.14. Адресация портов ввода-вывода

Для доступа к портам, находящимся в адресном пространстве ввода-вывода, используются два режима адресации, представленные на рис. 2.14. В прямой адресации номер порта представляет собой 8-битный непосредственный операнд, находящийся во втором байте команды, что обес-

печивает обращение к фиксированным портам 0—255. Косвенная адресация портов ввода-вывода аналогична регистровой косвенной адресации операндов в памяти: номер порта находится в регистре DX и имеет диапазон 0—65 535. С помощью предварительной инициализации регистра DX одна и та же команда может обращаться к любому порту в адресном пространстве ввода-вывода. К группе соседних портов возможно обращаться с помощью простого программного цикла, который модифицирует содержимое регистра DX.

Примеры использования команд ввода-вывода в ассемблерных программах приведены ниже.

```
IN    AL, 40H   ; Ввод байта из фиксированного порта
OUT   DX, AX    ; Вывод слова по адресу, хранимому в DX
IN    AX, DX    ; Ввод слова из устройства, адрес которого
                ;     хранится в DX
```

Режим адресации и время выполнения команды. Как следует из предыдущего материала, эффективный адрес памяти получается в результате некоторых вычислений. Время их выполнения, а следовательно, и время выполнения команды будет варьироваться в зависимости от сложности вычислений. Время вычисления эффективного адреса зависит от режима адресации. Оно равно числу тактов синхронизации, необходимому для определения эффективного адреса при условии, что команда находится во внутренней очереди команд. Зависимость времени вычисления эффективного адреса от используемого режима адресации приведена в табл. 2.6. Наличие перед командой префикса замены сегмента увеличивает находящиеся в таблице величины на два такта синхронизации.

Таблица 2.6. **Время вычисления эффективного адреса**

Адресация	Обозначение	Число тактов
Прямая	*disp*	6
Косвенная регистровая	[BX], [BP], [SI], [DI]	5
Базовая или индексная	[BX, BP, SI, DI]+*disp*	9
Базовая индексная (без смещения)	[BP]+[DI], [BX]+[SI]	7
	[BP]+[SI], [BX]+[DI]	8
Базовая индексная (со смещением)	[BP]+[DI]+*disp*	11
	[BX]+[SI]+*disp*	
	[BP]+[SI]+*disp*	12
	[BX]+[DI]+*disp*	

2.3. Система команд

Систему или набор (и даже репертуар) команд МП К1810 образуют 113 базовых команд, многие из которых допускают использование разнообразных режимов адресации. В изучении системы команд МП помогает группирование команд по функциональному назначению. Кроме того, значительную помощь читателю окажут и подробные примеры выполнения команд.

2.3.1. Команды передачи данных

Как любой современный процессор, МП К1810 имеет обширный набор команд, предназначенных для пересылок данных между регистрами, а также между регистрами и памятью. Команды передач данных удобно разделить на четыре подгруппы:

общие команды передачи данных;

команды передачи данных с привлечением стека (стековые команды);

команды ввода-вывода;

команды передачи цепочек байт или слов.

В данном параграфе сравнительно подробно рассмотрены команды первых трех подгрупп, а команды манипуляций цепочками данных в силу их специфики рассмотрены в п. 2.3.5.

Отличительной особенностью команд передач данных является то, что они не модифицируют состояния флажков. Исключение составляют только команды POPF и SAHF, которые прямо воздействуют на регистр флажков.

Общие команды передачи данных. Данная подгруппа включает в себя команды, осуществляющие передачи регистр — регистр, регистр — память и память — регистр. Наиболее мощной среди них является команда MOV (передать, переслать).

Команда MOV. Эта команда имеет следующее обобщенное представление:

MOV *dst*, *src* *dst*:=(*src*)

и осуществляет передачу содержимого источника *src* в получатель *dst*. Команда MOV имеет несколько форматов, показанных на рис. 2.15.

Самой гибкой и универсальной является команда MOV *mem*/*reg*$_1$, *mem*/*reg*$_2$. Она содержит постбайт режима

адресации, с помощью которого можно задавать любой допустимый режим адресации из рассмотренных в § 2.2. Наличие бита *w* показывает возможность передач байт и слов, а бит *d* определяет направление передачи. Таким образом, с помощью одной этой команды осуществляются передачи регистр — регистр/память и память — регистр, причем ре-

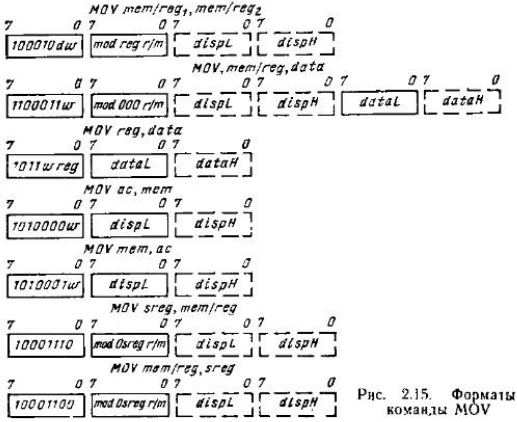


Рис. 2.15. Форматы команды MOV

гистром может быть любой общий регистр. Длина команды составляет 2, 3 или 4 байта, а время выполнения равно двум тактам в формате регистр — регистр, (8+EA) тактам в формате память — регистр и (9+EA) тактам синхронизации в формате регистр — память.

Пример 1.

(SI) = 3456 (DS) = ABCD ([AF126] = 3AAD	MOV	CX, [SI]	(CX) = 3AAD

Пример 2.

(BX) = 789A	MOV	DI, BX	(DI) = 789A

Команда MOV *mem/reg*$_1$, *mem/reg*$_2$ выполняет функцию команды MOV r_1, r_2 микропроцессора К580, но является более универсальной.

Примеры записи команды на языке ассемблера приведены ниже.

```
MOV   DX, CX          ; Слово, регистр — регистр
MOV   BP, GAMMA       ; Слово, память — регистр
MOV   TEMP, BL        ; Байт, регистр — память
MOV   [DI], BX        ; Слово, регистр — память
MOV   DX, [BP][SI]    ; Слово, память — регистр
```

Примечание. Для лучшего усвоения копирования команд рекомендуется самостоятельно построить машинные команды, реализующие приведенные выше операторы языка ассемблера. Символическим именам, например GAMMA, TEMP, можно присваивать произвольные 16-битные значения. Такие же упражнения целесообразно выполнять для всех ассемблерных операторов, которые приводятся в качестве примеров.

Команда MOV *mem/reg*, *data* позволяет передать в регистр или ячейку памяти непосредственные данные. Она также допускает большинство режимов адресации и передачу байт или слов.

Длина команды в зависимости от режима адресации варьируется от 3 до 6 байт, а выполняется за (10+EA) тактов синхронизации.

Пример 3.

(DI) = 1000 (DS) = A345	MOV	[DI], 8400H	([A4450]) = 8400

Примеры записи на языке ассемблера приведены ниже.

```
MOV   [BX], 3450H     ; Слово в память
MOV   [BP][SI], 2CH   ; Байт (или слово) в память
MOV   CX, 1000H       ; Слово в регистр
```

В этой команде нельзя определять сегментные регистры. Отсутствие команды загрузки непосредственных данных в сегментные регистры считается недостатком системы команд МП К1810, так как при инициализации системы в сегментные регистры приходится загружать константы. Решение этой проблемы приведено ниже.

Команда MOV *reg*, *data* представляет собой более короткий вариант предыдущей команды и осуществляет загрузку констант в общие регистры МП. Длина этой команды составляет 2 или 3 байта, а время выполнения равно четырем тактам синхронизации. Она эквивалентна командам MVI *r*, *data* и LXI *rp*, *data* микропроцессора К580.

Пример 4.

(CL) = 63 MOV CL, 0FFH (CL) = FF

Примечание. Выше в примерах записи команды MOV *mem,reg, data* на языке ассемблера была приведена команда MOV CX, 1000H. Возможно, естественно, закодировать ее 4 байтами в формате команды MOV *mem,reg, data*, однако программа-ассемблер выберет более короткий 3-байтный вариант команды в формате MOV *reg, data*.

Команды MOV *ac, mem* и MOV *mem, ac* предназначены для загрузки и запоминания содержимого аккумуляторов. Например, команда MOVAL, BETA передает в аккумулятор AL байт с адресом BETA, а команда MOV ALPHA, AX запоминает содержимое аккумулятора AX в слове памяти с адресом ALPHA. При выполнении последней команды содержимое регистра AL оказывается в байте с адресом ALPHA, а содержимое регистра AH — в байте с адресом ALPHA+1. Напомним, что оба байта находятся в текущем сегменте данных (если специально не определен другой сегмент) и указанные адреса представляют собой смещения в этом сегменте.

Обе команды имеют длину 3 байта и выполняются за десять тактов синхронизации.

Команды MOV *sreg, mem/reg* и MOV *mem/reg, sreg* осуществляют передачи между сегментными регистрами и регистрами/памятью. В них передаются только слова, а ячейка памяти может быть определена с помощью любого допустимого режима адресации. Отметим, что в команде MOV *sreg, mem/reg* нельзя указывать сегментный регистр кода CS, так как при этом результат операции неопределен.

Обе команды имеют длину 2—4 байта в зависимости от режима адресации. Время выполнения в формате регистр—регистр составляет два такта синхронизации; при задании памяти первая команда выполняется за (8+EA), а вторая — за (9+EA) тактов синхронизации.

Пример 5.

CX = 5F00 MOV ES, CX (ES) = 5F00

Примеры записи команд на языке ассемблера приведены ниже.

```
MOV   SS, BP    ; Регистр — сегментный регистр
MOV   [BX], ES  ; Сегментный регистр — память
```

Команда MOV *sreg, mem/reg* применяется для инициализации сегментных регистров, т. е. для определения сегментов. Если, например, в сегментный регистр DS необходимо загрузить F000, то потребуются команды

```
MOV   AX, 0F000H  ; Инициализация
MOV   DS, AX      ;   регистра DS на F000
```

При такой инициализации почти всегда как промежуточный используется регистр AX, так как команда MOV *ac, data* короче более общей команды MOV *mem/reg, data*.

Команда XCHG. Команда обмена XCHG имеет два формата, показанные на рис. 2.16. Первый формат позволя-

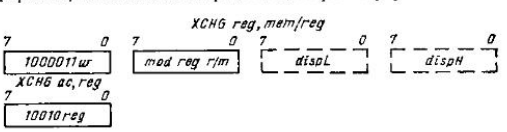


Рис. 2.16. Команда обмена

ет обменять содержимое любого общего регистра и ячейки памяти, а также любой пары общих регистров. Память адресуется в любом допустимом режиме адресации. Однобайтная команда XCHG предназначена для обмена любого общего регистра и аккумулятора AX. Первая из команд имеет длину 2—4 байта и выполняется за четыре (регистр—регистр) или (17+EA) (регистр—память) тактов синхронизации.

Пример 6.

(CX) = 37AA (BP) = 2346 (SS) = A000 ([A2346]) = DCBA	XCHG CX, [BP]	([A2346]) = 37AA (CX) = DCBA

Пример 7.

(DL) = FF (AH) = A8	XCHG AH, DL	(DL) = A8 (AH) = FF

Примеры использования команд обмена в ассемблерных операторах приведены ниже.

```
XCHG   AL, SEMA  ; Обмен байтами память — регистр
XCHG   AL, BL    ; Обмен байтами регистр — регистр
XCHG   AX, CX    ; Обмен словами регистр — аккумулятор
```

Отметим следующее:

в команде XCHG нельзя указывать сегментные регистры,

команда XCHG AX,AX используется как команда пустой операции NOP.

Команда XLAT. Однобайтная команда преобразования XLAT с кодом операции D7 заменяет содержимое аккумулятора AL на байт из 256-байтной таблицы, начальный адрес которой находится в регистре BX (рис. 2.17). Другими словами, содержимое AL используется как индекс таблицы, адресуемой регистром BX. Алгоритм выполнения команды XLAT состоит из двух шагов:

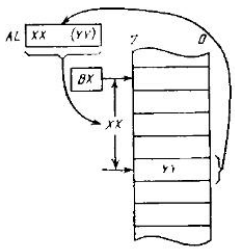


Рис. 2.17. Действие команды преобразования

1) прибавить содержимое регистра AL к содержимому регистра BX;

2) использовать результат как смещение в сегменте данных (относительно DS) и поместить адресуемый байт из памяти в регистр AL.

Команда XLAT обычно применяется для быстрого преобразования символов из одного символьного кода в другой. Время ее выполнения составляет 11 тактов синхронизации.

Пример 8.

```
(AL) = 47          XLAT    (AL) = E1
(BX) = 1200
(DS) = F000
([F1247]) = E1
```

Команды LEA, LDS, LES. Рассматриваемые команды отличаются от других команд передачи тем, что при их выполнении в адресуемый регистр (регистры) передаются не собственно данные, а адреса. Форматы этих команд приведены на рис. 2.18.

Основное применение команд LEA, LDS и LES — это инициализация регистров перед выполнением цепочечных команд.

При выполнении команды загрузки эффективного адреса LEA *reg, mem* вычисляется эффективный адрес EA памяти (в соответствии с указанным режимом адресации) и его значение (а не адресуемое им слово в памяти!) загружается в указанный общий регистр. Такая операция может потребоваться, например, для загрузки в регистр BX начального адреса таблицы, которая необходима для выполнения команды XLAT.

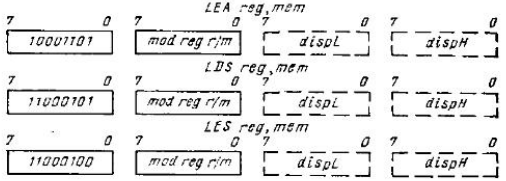


Рис. 2.18. Команды загрузки адресов

Пример 9.

```
(BX) = 0400     LEA BX, [BX][SI]     (BX) = 043C
(SI) = 003C
```

Команды LDS и LES выполняют почти один и те же действия: вычисляется эффективный адрес EA памяти, который суммируется с содержимым регистра DS; затем слово из памяти по полученному адресу загружается в адресуемый командой общий регистр, а следующее слово из памяти загружается в регистр DS (команда LDS) или ES (команда LES).

Пример 10.

```
(DS) = C000          LDS SI, [DI]    (SI) = 0180
(DI) = 0010                          (DS) = 2000
([C0010]) = 0180
([C0012]) = 2000
```

Длина всех рассматриваемых команд может быть 2—4 байта, а время выполнения составляет: команды LEA (2+EA), команд LDS и LES (16+EA) тактов синхронизации. Если поле *mod*=11 (регистровая адресация), действия команд не определены. Отметим, что обычно в команде LDS указывается регистр SI, а в команде LES регистр DI,

так как в цепочечных операциях регистр SI ассоциируется с регистром DS, а регистр DI с регистром ES.

Примеры записи команд на языке ассемблера приведены ниже.

```
LEA   BP, ALPHA
LDS   SI, [DI]
LES   DI, [DI+8]
```

Команды LAHF и SAHF. Однобайтные команды LAHF (код операции 9F) и SAHF (код операции 9E) введены для упрощения программной совместимости микропроцессоров К1810 и К580. Команда LAHF осуществляет передачу младшего байта регистра флажков в регистр AH (состояния самих флажков при этом не изменяются), а команда SAHF реализует обратную передачу — содержимое регистра AH передается в младший байт регистра флажков (старший байт регистра флажков не изменяется). Обе команды выполняются за четыре такта синхронизации.

В микропроцессоре К580 двухбайтное слово состояния процессора PSW образовано содержимым регистра флажков и аккумулятора. Оно фигурирует в двух командах: команда PUSH PSW включает слово состояния в стек, а команда POP PSW извлекает PSW из стека. Наличие в МП К1810 команд LAHF и SAHF позволяет легко эмулировать эти действия следующими командами:

```
LAHF              ;Эквивалент
PUSH  AX          ;    команды  PUSH  PSW
POP   AX          ;Эквивалент
SAHF              ;    команды  POP   PSW
```

Стековые команды. Форматы стековых команд МП К1810 представлены на рис. 2.19. Каждая команда включения в стек PUSH имеет соответствующую ей команду извлечения из стека POP. Для адресации вершины стека (мнемоническое обозначение TOS), находящейся в текущем сегменте стека и содержащей последний включенный в стек элемент данных, предназначен указатель стека SP. Все стековые команды манипулируют только словами и сопровождаются автоматической модификацией указателя стека: при включении в стек производится декремент, а при извлечении из стека — инкремент SP. До выполнения стековых команд необходимо инициализировать регистры SP и SS. Кроме того, каждой команде POP должна предшествовать команда PUSH. Отметим, что команды вызова CALL и возврата RET используют стек автоматически. Команда

PUSH часто используется для передачи через стек параметров подпрограммам.

Команда PUSH *mem/reg* включает в стек содержимое адресуемого регистра или ячейки памяти, а команда POP *mem/reg* извлекает содержимое вершины стека и передает его в общий регистр или ячейку памяти. Эти команды име-

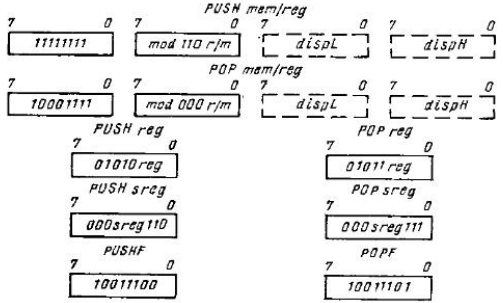


Рис. 2.19. Стековые команды

ют длину 2—4 байта и выполняются за 11 (PUSH) и 8(POP) тактов синхронизации при указании общего регистра и за (16+EA) и (17+EA) тактов соответственно при адресации памяти.

Пример 11.

```
(DS) = 2800                PUSH [BX]    ([2FFFE]) = A020
(BX) = 0400                             (SP) = 0FFE
(SP) = 1000
(SS) = 2F00
([28 4000]) = A020
```

Однобайтные команды PUSH *reg* и POP *reg* производят соответствующие стековые операции с общими регистрами. Время их выполнения составляет 10 (PUSH) и 8(POP) тактов синхронизации.

Пример 12.

```
(SP) = 3456        PUSH CX      ([AE454]) = 7777
(SS) = AB00                     (SP) = 3454
(CX) = 7777
```

Команды PUSH *sreg* и POP *sreg* выполняют стековые операции с сегментными регистрами. Отметим, что указание в них регистра CS не допускается.

Наконец, одиобайтные команды PUSHF (код операции 9C) и POPF (код операции 9D) предназначены для временного запоминания в стеке и последующего восстановления из стека содержимого 16-битного регистра флажков. С их помощью можно изменять состояние флажка TF, так как команд прямого воздействия на этот флажок нет. Изменение осуществляется с помощью включения регистра флажков в стек, необходимого изменения бита 8 копии содержимого регистра флажков в памяти и последующего ее извлечения из стека.

Последние четыре команды выполняются за то же время, что и команды PUSH *reg* и POP *reg*.

Примеры записи стековых команд на языке ассемблера приведены ниже.

```
PUSH    SI          ; Включение в стек регистра
PUSH    ES          ; Включение в стек сегментного регистра
PUSH    [DI]        ; Включение в стек слова из памяти
PUSHF               ; Включение в стек флажков
POPF                ; Извлечение из стека в регистр флажков
POP     ALPHA       ; Извлечение из стека в память
POP     DS          ; Извлечение из стека в сегментный регистр
POP     DX          ; Извлечение из стека в регистр
```

Команды ввода-вывода. Форматы команд ввода-вывода показаны на рис. 2.20. В байте кода операции всех команд

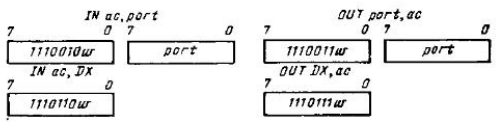


Рис. 2.20. Команды ввода-вывода

находится бит *w*; следовательно, каждая команда может передавать байты и слова. Ввод и вывод осуществляются через аккумуляторы AL (байты) и AX (слова).

Двухбайтные команды IN *ac*, *port* и OUT *port*, *ac* содержат во втором байте прямой адрес порта. Команда IN загружает данные из адресуемого порта в аккумулятор, а команда OUT выполняет передачу данных в противоположном направлении. Эти команды выполняются за десять тактов синхроизации и адресуют порты в диапазоне 00—FF.

Одиобайтные команды IN *ac*, DX и OUT DX, *ac* также допускают передачи байт и слов, но теперь 16-битный адрес порта находится в регистре DX и максимальный адрес порта равен FFFF. Косвенная адресация через регистр DX удобна в тех случаях, когда адрес нужного порта вычисляется по ходу выполнения программы. Обе команды выполняются за восемь тактов синхронизации.

В технической литературе ввод-вывод с прямыми адресами портов иногда называют статическим, а с переменными адресами — динамическим.

Примеры записи команд ввода-вывода на языке ассемблера приведены ниже.

```
IN      AX, 40H     ; Ввод слова (статический)
IN      AL, DX      ; Ввод байта (динамический)
OUT     80H, AL     ; Вывод байта (статический)
OUT     DX, AX      ; Вывод слова (динамический)
```

Для иллюстрации использования команд передач данных приведем несколько простых примеров [5].

1. Пусть необходимо передать содержимое одной области памяти с начальным адресом SRC в другую область с начальным адресом DST, причем обе области находятся в текущем сегменте данных. Распределим регистры МП следующим образом: регистр SI адресует текущий элемент области-источника, регистр DI — текущий элемент области-получателя, а регистр CX содержит число пересылаемых элементов данных N. Следующая циклическая программа пересылает байты, причем в качестве буферного регистра используется регистр AL.

Программа 1. Пересылка байт

```
MOVEB:  MOV     AL,[SI]     ; Пересылка байта в AL из источника
        MOV     [DI],AL     ; Пересылка в получатель
        INC     SI          ; Продвижение указателей
        INC     DI
        DEC     CX
        JNZ     MOVEB       ; Проверка окончания цикла
```

Если необходимо передавать слова, программа несколько модифицируется.

Программа 2. Пересылка слов

```
MOVEW:  MOV     AX,[SI]     ; Пересылка слова в AX из источ-
                            ;     ника
        MOV     [DI],AX     ; Пересылка в получатель
        INC     SI          ; Продвижение указателей
        INC     SI
        INC     DI
        INC     DI
        DEC     CX          ; Проверка окончания
        JNZ     MOVEW       ;     цикла и повторение
```

Отметим, что приведенные программы 1 и 2 могут быть составлены более эффективно с использованием цепочечных команд и префикса повторения.

2. Для правильной работы приведенных выше программ необходима соответствующая инициализация регистров SI, DI, CX и (если области не находятся в текущем сегменте данных) регистра DS. Требующиеся исходные данные удобно хранить в памяти в виде блока, адресуемого регистром (рис. 2.21, *а*). Такая конструкция называется блоком параметров, а его элементы — параметрами. Если этот блок адресуется регистром DI, то инициализация необходимых регистров осуществляется следующими командами:

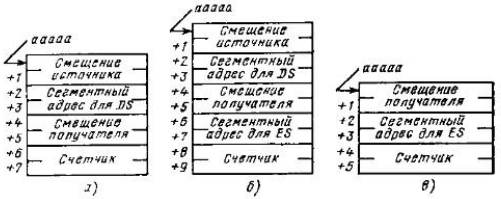


Рис. 2.21. Блоки параметров

```
        LDS     SI,[DI]     ; Инициализация SI и DS
        MOV     CX,[DI+6]   ; Инициализация CX
        MOV     DI,[DI+4]   ; Инициализация DI
```

Если область-получатель находится в сегменте, адресуемом регистром ES, целесообразно расширить блок параметров, включив в него сегментный адрес для ES (рис. 2.21, *б*). Когда блок параметров адресуется регистром DI, последовательность инициализации принимает следующий вид:

```
        LDS     SI,[DI]     ; Инициализация SI и DS
        MOV     CX,[DI+8]   ; Инициализация CX
        LES     DI,[DI+4]   ; Инициализация DI и ES
```

При такой инициализации размещение областей источника и получателя в адресном пространстве памяти может быть произвольным.

Если положение области-источника в памяти фиксировано, ее начальный адрес можно задать непосредственным операндом. Пусть OFFADDR — смещение области-источника, SADDR — ее сегментный адрес, а блок параметров имеет вид, приведенный на рис. 2.21, *в*, и находится в сегменте данных. Необходимую инициализацию реализуют следующие команды:

```
        MOV     SI,OFFADDR  ; Инициализация SI
        MOV     AX,SADDR    ; Инициализация DS
        MOV     DS,AX
        MOV     CX,[DI+4]   ; Инициализация CX
        LES     DI,[DI]     ; Инициализация DI и ES
```

Из-за отсутствия команд загрузки непосредственных данных в сегментные регистры инициализация регистра DS производится через аккумулятор AX.

3. Предположим, что при передаче байт из области-источника в область-получатель необходимо осуществить их преобразование с помощью таблицы. Для этой цели удобно воспользоваться командой преобразования XLAT. Пусть регистр BX содержит начальный адрес таблицы, а регистры SI, DI и CX выполняют такие же функции, что и в программе 1.

Программа 3. Передача байт с преобразованием

```
TRANS:  MOV     AL,[SI]     ; Передача  байта в AL из ис-
        XLAT                ;     точника
        MOV     [DI],AL     ; Преобразование
        INC     SI          ; Пересылка в получатель
        INC     DI          ; Продвижение указателей
        DEC     CX
        JNZ     TRANS       ; Проверка окончания цикла
```

Для этой программы в блок параметров необходимо включить смещение для BX.

4. Если микро-ЭВМ работает с операционной системой, то обычно все манипуляции сегментными регистрами, в том

числе их инициализация, осуществляются операционной системой. Когда операционной системы нет, сегментные регистры должна инициализировать прикладная программа. При задании фиксированных начальных адресов сегментов инициализацию сегментных регистров выполняют по программе, приведенной ниже.

Программа 4. Инициализация сегментных регистров

```
MOV     AX,IMMDS        ; Инициализация DS
MOV     DS,AX
MOV     AX,IMMES        ; Инициализация ES
MOV     ES,AX
MOV     AX,IMSS         ; Инициализация SS
MOV     SS,AX
```

Возможен и другой способ инициализации сегментных регистров.

Программа 5. Инициализация сегментных регистров из программной памяти

```
MOV     DS, CS:ADS      ; Инициализация DS
MOV     ES, CS:AES      ; Инициализация ES
MOV     SS, CS:ASS      ; Инициализация SS
```

Конструкция CS: вызывает включение в объектную программу байта префикса замены сегмента перед каждой командой. Наличие префикса приводит к тому, что в процессе формирования физического адреса памяти участвует не регистр DS, как обычно, а регистр CS.

Микропроцессор К1810 обеспечивает определенную защиту при инициализации регистров SS и SP. Когда они инициализируются командами

```
MOV     SS, CS:ASS
MOV     SP, CS:ASP
```

МП не воспринимает прерывания до завершения обеих команд.

5. Реагируя на внешнее прерывание, МП автоматически запоминает в стеке содержимое регистров флажков, PC и CS (см. § 1.8). Подпрограмма обработки прерывания в общем случае должна временно сохранить в стеке содержимое всех остальных регистров, а по окончании — восстановить его. Эти функции реализуют команды, приведенные ниже.

Программа 6. Запоминание состояния МП в стеке

```
PUSH    ES
PUSH    DS
PUSH    SI
PUSH    DI
PUSH    BP
PUSH    DX
PUSH    CX
PUSH    BX
PUSH    AX
```

Программа 7. Восстановление состояния МП из стека

```
POP     AX
POP     BX
POP     CX
POP     DX
POP     BP
POP     DI
POP     SI
POP     DS
POP     ES
```

Определенного порядка включения в стек содержимого регистров нет, не необходимо производить восстановление в порядке, обратном включению в стек.

2.3.2. Команды арифметических операций

Микропроцессор К1810 имеет широкий набор команд, реализующих арифметические операции, что позволяет применять его в сложных системах обработки данных.

Арифметические операции выполняются над целыми числами четырех типов: беззнаковыми двоичными, знаковыми двоичными, упакованными десятичными и неупакованными десятичными. Сначала рассмотрим операции над двоичными числами, а десятичную арифметику рассмотрим отдельно.

Длина беззнаковых двоичных чисел составляет 8 или 16 бит, и все они считаются значащими, т. е. учитываются при определении значения числа. Диапазон значений 8-битных чисел составляет 0—255, а 16-битных 0—65 535. Имеются команды сложения, вычитания, умножения и деления чисел, представленных в таких форматах.

Знаковые двоичные числа также могут быть 8- и 16-битными. Старший (левый) бит определяет знак числа: 0 — число положительное, 1 — число отрицательное. Числа представляются в стандартном дополнительном коде. Диа-

пазон значений составляет —128÷+127 для 8-битных чисел и —32 768÷+32 767 для 16-битных чисел. Число нуль содержит во всех разрядах нули и считается положительным. Для знаковых двоичных чисел имеются специальные команды умножения и деления, а сложение и вычитание благодаря применению дополнительного кода реализуются теми же командами, что и для беззнаковых чисел.

При выполнении арифметических операций особенности (или признаки) полученного результата фиксируются в 6 битах регистра флажков. Состояния большинства флажков проверяются командами условных переходов; имеется также специальная команда INTO прерывания при переполнении. Команды арифметических операций влияют на флажки по-разному, но справедливы следующие общие правила:

флажок **CF** фиксирует значение переноса (при сложении) и заема (при вычитании) из старшего бита. Его можно использовать для обнаружения переполнения при сложении беззнаковых двоичных чисел;

флажок **AF** фиксирует значение переноса (при сложении) и заема (при вычитании) из младшей тетрады. Этот флажок введен только для команд десятичной коррекции (см. ниже) и для других целей не используется;

флажок **SF** соответствует значению старшего бита (бит 7 или бит 15) результата операции. В знаковой арифметике отражает знак результата; в беззнаковой — интерпретируется как цифра, а не как знак;

флажок **ZF** фиксирует получение нулевого результата;

флажок **PF** устанавливается в 1, если младшие 8 бит результата операции содержат четное число единиц; в противном случае он сбрасывается в 0. Наличие флажка PF упрощает реализацию функций контроля символьных данных;

флажок **OF** в операциях со знаковыми числами фиксирует арифметическое переполнение, т. е. выход результата за диапазон представимых значений (при использовании дополнительного кода переполнение определяется сложением по модулю 2 значений переносов из знакового бита и старшего значащего бита).

Команды сложения. Форматы команд сложения МП К1810 приведены на рис. 2.22. Как видно из рисунка, все команды, кроме команды INC *reg*, имеют в коде операции бит *w* и, следовательно, могут оперировать байтами и словами.

Общее представление команды сложения имеет вид

ADD *dst*, *src* *dst*: = (*dst*) + (*src*)

Команда производит сложение операндов *dst* и *src* и помещает сумму на место *dst*.

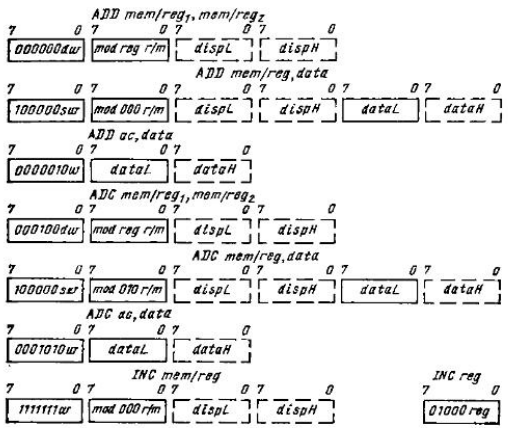


Рис. 2.22. Форматы команд сложения

Команда ADD $mem/reg_1, mem/reg_2$, содержащая постбайт режима адресации, позволяет производить сложение регистр—регистр/память и память—регистр, причем адресация памяти осуществляется в любом допустимом режиме. Длина команды составляет 2—4 байта, а время выполнения равно трем (регистр—регистр), (9+EA) (регистр—память) или (16+EA) (память—регистр) тактам синхронизации.

Пример 13.

(CX)=0049
(SI)=09EA

ADD SI,CX

(SI)=0A33
OF,SF,ZF,CF=0
AF,PF=1

Пример 14.

(AX)=F0F0 (DI)=3000 (DS)=1000 ([13008])=B456	ADD AX,[DI+8]	AX=A546 OF,ZF,AF,PF=0 SF,CF=1

Команда ADD *mem/reg, data* позволяет прибавить к содержимому регистра или ячейки памяти 8/16-битную константу, находящуюся в команде. Бит *s* в коде операции определяет размер константы: если s=0, команда содержит 16-битную константу, а если *s*=1, в команде находится 8-битная константа, но с помощью расширения знака она преобразуется в 16-битное число (напомним, что операция расширения знака для целого числа в дополнительном коде не изменяет его значения).

Длина команды варьируется от 3 до 6 байт, а время выполнения составляет четыре такта синхроизации, если определен регистр, и (17+EA) тактов, если определена ячейка памяти.

Пример 15.

(CX)=4567	ADD CX, 0ABCDH	(CX)=F134 OF, ZF, PF, CF=0 SF, AF=1

Команда ADD *ac,data* представляет собой более короткий вариант предыдущей команды и производит прибавление непосредственных данных к аккумуляторам AL (8 бит) или AX (16 бит). Она имеет длину 2 или 3 байта и выполняется за четыре такта синхронизации.

Пример 16.

(AL)=4E	ADD AL, 0C5H	(AL)=13 OF,SF,ZF,PF=0 AF,CF=1

Следующие три команды ADC сложения с переносом имеют обобщенное представление в виде

$$\text{ADC}\ \ dst,src \qquad dst := (dst) + (src) + (\text{CF})$$

Они имеют такие же форматы, как и команды ADD, и выполняются за такое же время. Единственное отличие их заключается в том, что в сложении наряду с операндами участвует флажок CF, значение которого прибавляется к младшему биту результата сложения операндов.

Пример 17.

(AX)=89AB (DI)=0300 (DS)=1200 ([12300])=3748 (CF)=1	ADC [DI], AX	([12300])=COF4 OF,ZF,PF,CF=0 SF,AF=1

Пример 18.

(AL)=F4 (CF)=0	ADC AL,35H	(AL)=29 OF,SF,ZF,AF,PF=0 CF=1

Команда инкремента имеет обобщенное представление:

$$\text{INC}\ \ dst \qquad dst := (dst) + 1$$

Она позволяет увеличить на единицу содержимое любого общего регистра или ячейки памяти. Команда INC воздействует на все арифметические флажки, за исключением флажка CF, — его состояние не изменяется. Операнд *dst* считается беззнаковым двоичным числом.

В команде INC *mem/reg* могут быть указаны ячейка памяти и любой общий регистр (8- и 16-битный). Она состоит из 2—4 байт, а время выполнения составляет три такта, если производится инкремент регистра, и (15+EA) тактов, если производится инкремент ячейки памяти.

Пример 19.

(DI)=FF00 (DS)=A000 ([AFF00])=FF (CF)=0	INC [DI]	([AFF00])=00 OF,SF,CF=0 ZF,AF,PF=1

Однобайтная комаида INC *reg* предназначена для инкремента общих 16-битных регистров и выполняется за два такта синхронизации.

Пример 20.

(SI)=00FF (CF)=1	INC SI	(SI)=0100 OF,SF,ZF=0 AF,PF,CF=1

Примеры записи команд сложения на языке ассемблера приведены ниже.

```
ADD     CX, DX          ; Регистр — регистр
ADD     DI, ALPHA       ; Регистр — память
ADD     TEMP, CL        ; Память — регистр
ADD     BL,02H          ; Регистр — константа
ADD     BETA,0FFH       ; Память — константа
```

Продолжение

```
ADC     DI,SI          ; Регистр — регистр
ADC     AX,7777H       ; Аккумулятор — константа
INC     BL             ; Регистр (8 бит)
INC     CX             ; Регистр (16 бит)
INC     GAMMA [DI]     ; Ячейка памяти
```

Команды вычитания. Команды вычитания, форматы которых приведены на рис. 2.23, отличаются от соответствующих команд сложения только выполняемой операцией.

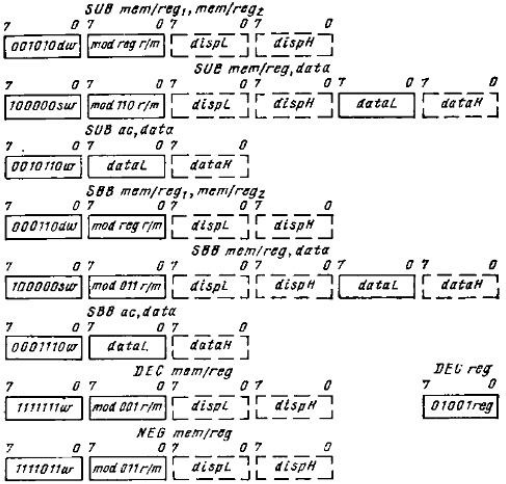


Рис. 2.23. Форматы команд вычитания

В связи с этим они рассмотрены довольно конспективно, но с таким же числом примеров. Специально подчеркнем, что флажки CF и AF в операциях вычитания становятся флажками заема и устанавливаются в единицу, когда уменьшаемое меньше вычитаемого.

Общее представление команды вычитания имеет вид

SUB *dst,src* *dst*:= (*dst*) — (*src*)

Команда осуществляет вычитание операндов *dst* и *src* и запоминает разность в *dst*.

Пример 21.

```
(CL)=83          SUB CL, [BP+8]      (CL)=E9
(SS)=1000                            OF, ZF, PF=0
(BP)=FFFE                            SF, AF, CF=1
([20006])=9A
```

Пример 22.

```
(SI)=A000        SUB[SI+20H], 2AAH   ([2F020])=31AC
(DS)=2500                            OF,SF,ZF,CF=0
([2F020])=3456                       AF,PF=1
```

Пример 23.

```
(AL)=73          SUB AL, 0B7H        (AL)=BC
                                     ZF,PF=0
                                     OF,SF,AF,CF=1
```

Команда вычитания с заемом имеет следующее общее представление:

SBB *dst,src* *dst*:= (*dst*) — (*src*) — (CF)

Эта команда из разности операндов вычитает еще значение флажка CF.

Пример 24.

```
(BH)=78          SBB BH,DH           (BH)=6F
(DH)=08                              OF, SF,ZF,CF=0
(CF)=1                               AF, PF=1
```

Пример 25.

```
(BP)=0F6A        SBB [BP+40H], 5555H ([2FFAA])=CAAB
(SS)=2F00                            OF,ZF,PF=0
([2FFAA])=2000                       SF,AF, CF=1
(CF)=0
```

Пример 26.

```
(AX)=6B3A        SBB AX, 4D2CH       (AX)=1E0D
(CF)=1                               OF,SF,ZF,PF,CF=0
                                     AF=1
```

Команда декремента с общим представлением

DEC *dst* *dst*:= (*dst*) — 1

предназначена для уменьшения на единицу содержимого регистра или ячейки памяти. Она не влияет на состояние флажка переноса CF.

Пример 27.

```
(SI)=6000          DEC [SI]      ([30000])=4F
(DS)=2A00                        OF,SF,ZF,PF=0
([30000])=50                     AF,CF=1
(CF)=1
```

В этом примере предполагается байтная операция, хотя из мнемокода команды DEC [SI] не ясно, какую из двух операций декремента (байт или слово) следует выполнять. Способы устранения неоднозначности ассемблерных команд рассмотрены в § 3.2.

Пример 28.

```
(DX)=0000          DEC DX        (DX)=FFFF
(CF)=0                           OF,ZF,CF=0
                                 SF,AF,PF=1
```

Команда NEG изменения знака (или образования дополнительного кода) имеет следующее общее представление:

NEG *dst* *dst*:=0 − (*dst*)

Если операнд равен нулю, его значение не изменяется. Попытка изменить знак байта 80 или слова 8000 не модифицирует операнд, но устанавливает флажок переполнения OF. Команда влияет на все флажки, причем флажок CF всегда устанавливается в единицу, кроме случая, когда операнд равен нулю, — тогда флажок CF=0.

Пример 29.

```
(BX)=0006          NEG BX        (BX)=FFFA
                                 OF,ZF=0
                                 SF,AF,PF,CF=1
```

Команда NEG имеет длину 2—4 байта и выполняется за три (регистр) или (16+EA) (память) тактов синхронизации.

Примеры записи команд вычитания на языке ассемблера приведены ниже.

```
SUB   CX, BX          ; Регистр — регистр
SUB   DX, ALEX, [SI]  ; Регистр — память
SUB   [BP+2], CL      ; Память — регистр
SUB   AL, 10H         ; Регистр — константа
SUB   [SI], 0F000H    ; Память — константа
SBB   AX, BP          ; Регистр — регистр
SBB   [DI], 30H       ; Память — константа
DEC   [BP]            ; Память
DEC   AL              ; Регистр (8 бит)
DEC   BX              ; Регистр (16 бит)
```

Команды сравнения. Команды сравнения, форматы которых приведены на рис. 2.24, очень похожи на соответствующие команды вычитания. Единственное их отличие заключается в том, что результат вычитания операндов нигде не запоминается. Следовательно, данные команды производят так называемое неразрушающее сравнение операндов. Состояния всех флажков определяются получающимся результатом и могут быть (кроме флажка AF) проверены командами условного перехода.

Команда CMP mem/reg_1, mem/reg_2 состоит из 2—4 байт и выполняется за три (регистр—регистр) или (9+EA) так-

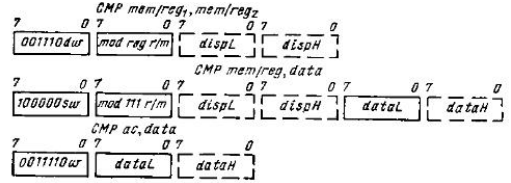


Рис. 2.24. Форматы команд сравнения

тов синхронизации. Длина команды CMP *mem/reg, data* составляет 3—6 байт, а время выполнения равно четырем или (10+EA) тактам синхронизации. Наконец, команда CMP *ac, data* имеет длину 2 или 3 байта и выполняется за два такта синхроинзации.

Пример 30.

```
(DH)=05          CMP DH, [SI+10H]   (DH)=05
(DS)=2000                           ([28010])=28
(SI)=8000                           OF, ZF, PF=0
([28010])=28                        SF, AF,CF=1
```

Пример 31.

```
(AL)=20          CMP AL, 0DH        (AL)=20
                                    OF, ZF, PF, CF=0
                                    SF, AF=1
```

Команды умножения. Форматы команд умножения представлены на рис. 2.25. В МП К1810 имеются две команды умножения: для беззнаковых и знаковых двоичных чисел. Умножение десятичных чисел требует наличия специальных команд коррекции, которые рассматриваются ниже.

Команда умножения беззнаковых целых с обобщенным представлением

MUL *src* ext:*ac*:= (*ac*) • (*src*)

выполняет умножение адресуемого операнда *src* и содержимого аккумулятора *ac*. В операции над байтами функции

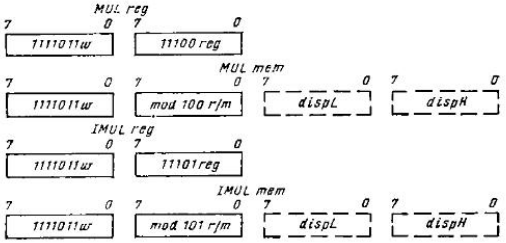


Рис. 2.25. Форматы команд умножения

аккумулятора *ac* выполняет регистр AL, а 16-битное произведение образуется в регистрах AH — AL. Регистр AH называется расширением (*ext*) аккумулятора AL. Если *src* идентифицирует слово, оно умножается на содержимое аккумулятора AX, а произведение длиной 32 бита формируется в регистрах DX — AX. В этой операции расширением аккумулятора AX является регистр DX.

Когда старшая половина произведения отличается от нулевой, флажки OF и CF устанавливаются в единицу, показывая наличие значащих цифр произведения в регистрах AH и DX. В противном случае флажки OF и CF принимают нулевые значения. Состояния остальных флажков после выполнения команды MUL не определены.

Длина команды MUL варьируется от 2 до 4 байт, а время выполнения зависит от режима адресации и значений сомножителей. При умножении 8-битных регистров оно составляет 70—77 тактов, а при умножении 16-битных регистров 118—133 такта синхронизации. Когда сомножителем определена память, соответствующие времена равны ((76...83)+EA) и ((124...139)+EA) тактам синхронизации.

Пример 32.

(AX)=A590	MUL DL	(AX)=1200
(DL)=20		OF, CF=1
		SF, ZF, AF, PF=?

Пример 33.

(AX)=0350	MUL [SI+15H]	(AX)=BA00
(DS)=1000		(DX)=0003
(SI) =F000		OF, CF=1
([1F015])=0120		SF, ZF, AF, PF=?

Команда IMUL *src* осуществляет практически такие же действия, что и команда MUL *src*, но сомножители и произведение интерпретируются как знаковые двоичные числа в дополнительном коде. Если старшая половина произведения, находящаяся в регистрах AH или DX, не является расширением знака младшей половины произведения [не содержит 00 (0000) или FF (FFFF) при умножении байт (слов)], флажки OF и CF устанавливаются в единицу. Это означает, что в старшей половине находятся значащие цифры произведения. В противном случае OF, CF=0. Состояния остальных флажков после выполнения команды IMUL не определены.

Длина команды IMUL составляет 2—4 байта. Время выполнения равно (в тактах синхронизации):

Регистр — регистр (8 бит)	80—98
Регистр — регистр (16 бит)	128—154
Регистр — память (8 бит)	(84—104)+EA
Регистр — память (16 бит)	(134—160)+EA

Пример 34.

(AX)=A590	IMUL DL	(AX)=F200
(DL)=20		OF, CF=1
		SF, ZF, AF, PF=?

Примеры записи команды умножения на языке ассемблера приведены ниже.

```
MUL     BL          ; Регистр, 8 бит, без знака
MUL     [DI—40H]    ; Память, без знака
IMUL    CX          ; Регистр, 16 бит, со знаком
IMUL    ALPHA       ; Память, со знаком
```

Команды деления. В МП К1810 имеются две команды деления, операндами которых являются беззнаковые и знаковые двоичные числа. Форматы команд деления показаны на рис. 2.26.

Команда деления беззнаковых чисел имеет обобщенное представление

DIV *src* $ac := quot\ ((ext:ac)/(src))$

$ext := rem\ ((ext:ac)/(src))$

и производит деление аккумулятора и его расширения (AH—AL, DX—AX для 8- и 16-битного делителя соответственно) на содержимое *src*. Частное *quot* формируется

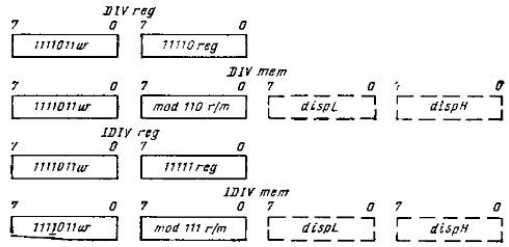


Рис. 2.26. Форматы команд деления

в регистре AL (или AX), а остаток *rem* — в регистре AH (или DX). Дробное частное округляется до целого путем отбрасывания дробной части результата. Состояния всех флажков не определены.

Если частное превышает разрядность аккумулятора (больше FF или FFFF) или делитель является нулем, генерируется прерывание типа 0, а частное и остаток не определены. При возникновении прерывания выполняются следующие действия:

содержимое регистра флажков включается в стек;

флажки IF и TF сбрасываются в 0;

содержимое сегментного регистра CS включается в стек;

в регистр CS загружается слово из памяти по адресу 00002;

содержимое PC включается в стек;

в PC загружается слово из памяти по адресу 00000.

В результате этих действий МП переходит к подпрограмме обработки прерывания типа 0, полный адрес которой сегмент:смещение берется из ячеек 00000 и 00002.

Длина команды DIV может быть 2—4 байта. Время выполнения операции составляет (в тактах синхронизации):

Регистр (8-битный делитель)	80—90
Регистр (16-битный делитель)	144—162
Память (8-битный делитель)	(86—96)+EA
Память (16-битный делитель)	(150—168)+EA

Пример 35.

```
(AX)=002D    DIV   CL    (AX)=0307
(CL)=96                  OF, SF, ZF, AF, PF, CF=?
```

Если генерирование прерывания типа 0 нежелательно (например, при отсутствии соответствующей подпрограммы обработки), необходимо до выполнения операции деления проверить возможность возникновения прерывания. Пусть, например, делитель находится в регистре CH. Тогда проверка возможности возникновения прерывания осуществляется командами

```
CMP   AH, CH          ; Сравнение и переход по пере-
JNB   OVERFLOW        ;   полнению
```

Когда 16-битный делитель находится, например, в регистре CX, аналогичная проверка реализуется командами

```
CMP   DX, CX          ; Сравнение и переход по пере-
JNB   OVERFLOW        ;   полнению
```

Находящаяся за командой сравнения команда условного перехода (перейти, если не ниже) передает управление по метке OVERFLOW, если левый операнд в команде CMP больше или равен правому, а это и есть условие возникновения переполнения.

Команда IDIV осуществляет почти такие же действия, как и команда DIV, но делимое и делитель считаются знаковыми числами. Диапазон представления частного составляет $-127 \div +127$, когда делителем является байт, и $-32\,767 \div +32\,767$, когда делитель — слово. При выходе частного за эти диапазоны и при попытке деления на нуль генерируется прерывание типа 0. Состояния флажков после выполнения команды деления IDIV неизвестны.

Команда IDIV может состоять из 2—4 байт. Время выполнения составляет (в тактах синхронизации):

Регистр (8-битный делитель)	101—112
Регистр (16-битный делитель)	165—184
Память (8-битный делитель)	(107—118)+EA
Память (16-битный делитель)	(171—190)+EA

Пример 36.

(AX)=FF57 (CH)=F3	IDIV CH	(AX)=000D OF, ZF, SF, AF, PF, CF=?

К подгруппе команд деления принадлежат две однобайтные команды преобразования. Команда преобразования байта в слово CBW, имеющая код операции 98, расширяет (копирует) знак содержимого регистра AL в регистр AH. Она не влияет на состояния флажков и выполняется за два такта синхронизации.

Пример 37.

(AL)=83 CBW (AX)=FF83

Команда CWD преобразования слова в двойное слово (код операции 99) передает знак содержимого регистра AX во все биты регистра DX. Команда не модифицирует флажки и выполняется за пять тактов синхронизации.

Пример 38.

(AX)=53AA (DX)=38BF	CWD	(AX)=53AA (DX)=0000

Команды CBW и CWD удобно использовать для превращения делимого одинарной длины в делимое двойной длины, что необходимо для корректного выполнения операции деления. Разумеется, обе команды не изменяют значения делимого.

Примеры записи команд деления на языке ассемблера приведены ниже.

```
DIV     CH          ; Регистр, байт без знака
DIV     [BP+20H]    ; Память, без знака
IDIV    CX          ; Регистр, слово со знаком
IDIV    BETA        ; Память, со знаком
```

Десятичная арифметика. Микропроцессор К1810 допускает два представления десятичных чисел: упакованный формат (BCD-формат) и неупакованный (ASCII-формат). В BCD-формате байт содержит две десятичные цифры (по одной в каждой тетраде), представленные в коде 8421; в ASCII-формате байт содержит одну десятичную цифру (в младшей тетраде), а старшая тетрада содержит либо 0011 (символьный код ASCII), либо нули.

В обоих форматах многоразрядные десятичные числа представляются последовательностями байт. Десятичная арифметика в МП К1810 оперирует только с байтами, причем основным рабочим регистром во всех десятичных операциях является регистр AL (эквивалент аккумулятора МП К580).

Сложение в BCD-формате. Сложение BCD-чисел выполняется в два этапа: сначала байты операндов суммируются как обычные двоичные числа, по правилам двоичной арифметики, а затем осуществляется коррекция результата. Анализ двоичного сложения BCD-чисел показывает, что неправильный BCD-результат появляется в двух ситуациях:

получена недопустимая тетрада, т. е. тетрада, двоичный эквивалент которой больше 9;

получена допустимая тетрада, но при сложении из нее возник двоичный перенос с весом 16, в то время как правильный вес единицы переноса должен быть равен 10.

Отметим, что перенос из младшей тетрады фиксируется флажком AF, а из старшей — флажком CF.

Коррекция двоичной суммы BCD-чисел, полученной с помощью команд ADD или ADC и находящейся в регистре AL, выполняется однобайтной командой DAA десятичной коррекции для сложения, имеющей код операции 27. Алгоритм коррекции состоит из двух шагов:

1) если AF=1 или младшая тетрада регистра AL содержит запрещенную комбинацию (в диапазоне чисел 10—15), к содержимому AL прибавляется 06 и флажок AF устанавливается в единицу;

2) если CF=1 или старшая тетрада регистра AL содержит запрещенную комбинацию, к содержимому AL прибавляется 60 и флажок CF устанавливается в единицу.

Команда DAA в соответствии с полученным в регистре AL результатом воздействует на все флажки, за исключением флажка переполнения OF, состояние которого после выполнения команды DAA не определено.

Пример 39.

(AL)=25 (BL)=68	ADD AL, BL DAA	(AL)=93 SF, AF, PF=1 ZF, CF=0

Вычитание в BCD-формате. Вычитание BCD-чисел, как и сложение, выполняется в два этапа: сначала операнды вычитаются как двоичные числа с помощью команды SUB или SBB, а затем результат, находящийся в регистре AL, корректируется командой DAS десятичной коррекции для вычитания. Однобайтная команда DAS име-

ет код операции 2F. Так как в МП К1810 операция вычитания выполняется путем сложения уменьшаемого и дополнительного кода вычитаемого, действия команды DAS описываются следующим образом:

1) если AF=1 или младшая тетрада регистра AL содержит запрещенную тетраду, из содержимого регистра AL вычитается 06 и флажок AF устанавливается в единицу;

2) если CF=1 или старшая тетрада регистра AL содержит запрещенную тетраду, из содержимого регистра AL вычитается 60 и флажок CF устанавливается в единицу.

Команда DAS модифицирует флажки так же, как команда DAA. Напомним, что флажки CF и AF в операции вычитания интерпретируются как флажки заема и устанавливаются в единицу, когда соответствующие переносы при сложении уменьшаемого и дополнительного кода вычитаемого не возникают.

Пусть, например, регистр AL содержит 10000110, а регистр AH 00000111. При двоичном вычитании с помощью команды SUB AL, AH получаем

```
10000110 <-------- Уменьшаемое
+
11111001 <-------- Дополнительный код вычитаемого
01111111 <-------- Неправильный результат; CF=0, AF=1

        По команде DAS выполняется коррекция:

01111111 <-
-
00000110 <-
--------
01111001 <-------- Правильный BCD-результат; SF=0, ZF=0,
                   AF=1, PF=0, CF=0.
```

Отметим, что при получении отрицательной разности она представлена в десятичном дополнительном коде и CF=1.

Сложение в ASCII-формате. ASCII-числа складываются в два этапа: сначала байты операндов суммируются как двоичные числа командой ADD или ADC с получением промежуточного результата в регистре AL, а затем команда коррекции для сложения AAA преобразует промежуточный результат в ASCII-формат. Действия команды AAA, как и других команд коррекции для ASCII-формата, построены с учетом реализации операций над многоразрядными числами. Коррекция сложения включает в себя следующие шаги:

1) если младшая тетрада регистра AL содержит допустимую тетраду и AF=0, перейти к шагу 3;

2) если младшая тетрада регистра AL содержит запрещенную комбинацию или AF=1, то необходимо прибавить 06 к содержимому регистра AL, прибавить единицу к содержимому регистра AH и установить AF=1;

3) сбросить старшую тетраду регистра AL;

4) установить флажок CF в то же состояние, в каком находится флажок AF.

Однобайтная команда AAA имеет код операции 37 и воздействует только на флажки AF и CF, а состояния остальных флажков после ее выполнения не определены.

Пример 40.

```
(AX)=0535    ADD    AL, BL    (AX)=0604
(BL)=39      AAA              AF, CF=1
```

Вычитание в ASCII-формате. Для коррекции результата двоичного вычитания (командой SUB или SBB) ASCII-чисел, который должен находиться в регистре AL, имеется команда коррекции для вычитания AAS. Эта команда реализует следующий алгоритм:

1) если младшая тетрада регистра AL содержит допустимую комбинацию и AF=0, перейти к шагу 3;

2) если младшая тетрада регистра AL содержит запрещенную комбинацию или AF=1, необходимо вычесть 06 из содержимого регистра AL, вычесть 1 из содержимого регистра AH и установить AF=1;

3) сбросить старшую тетраду регистра AL;

4) установить флажок CF в такое же состояние, в каком находится флажок AF.

Однобайтная команда AAS с кодом операции 3F воздействует только на флажки AF и CF.

Пример 41.

```
(AX)=0438    SUB    AL, 35H    (AX)=0403
             AAS               AF, CF=0
```

Умножение в ASCII-формате. Команды десятичной коррекции для умножения и деления BCD-чисел в МП К1810 отсутствуют, но если использовать неупакованный формат представления десятичных чисел (или специально их распаковывать), то можно выполнять в МП операции умножения и деления по правилам десятичной арифметики.

Умножение неупакованных десятичных чисел, как сложение и вычитание, выполняется в два этапа;

1) умножение одноразрядных сомножителей, представленных байтами, в которых младшие тетрады содержат десятичные цифры, а старшие тетрады нулевые. Умножение выполняется командой MUL, которая формирует в регистре AL двоичное произведение;

2) двухбайтная команда коррекции для умножения AAM преобразует полученный результат в двухбайтное произведение, находящееся в регистрах AH (старший разряд) и AL (младший разряд). Младшие тетрады в обоих байтах содержат цифры произведения, а старшие — нулевые.

Команда AAM, имеющая код операции D40A, осуществляет деление содержимого регистра AL на десять (0A) и загружает частное в регистр AH, а остаток — в регистр AL. Состояния флажков SF, ZF и PF зависят от содержимого регистра AL, а состояния флажков OF, AF и CF не определены.

Пример 42.

```
(AL)=07   MUL   BL    (AX)=0603
(BL)=09   AAM         PF=1
                      SF, ZF=0
```

Деление в ASCII-формате. Деление неупакованных десятичных чисел отличается от предыдущих операций тем, что необходимая коррекция производится до операции собственного деления. В двухбайтной команде коррекции для деления AAD предполагается, что в регистрах AH и AL находится двухразрядное делимое, причем регистр AH содержит цифру десятков, регистр AL — цифру единиц и обе старшие тетрады нулевые.

Двухбайтной командой AAD с кодом операции D50A выполняются следующие действия:

1) содержимое регистра AH умножается на десять (0A);

2) полученный результат прибавляется к содержимому регистра AL;

3) содержимое регистра AH сбрасывается;

4) состояния флажков SF, ZF и PF определяются по результату в регистре AL, а состояния флажков OF, AF и CF не определены.

Собственно деление полученного в регистре AX делимого на одноразрядный делитель осуществляется командой DIV.

Пример 43.

```
(AX)=0604   AAD   (AX)=0040
                  SF, ZF, PF=0
```


В заключение приведем алгоритмы арифметических операций над многоразрядными десятичными числами, представленными в неупакованном формате [6].

Сложение. Пусть $a(n)a(n-1)...a(2)a(1)$ — первое слагаемое, $b(n)b(n-1)...b(2)b(1)$ — второе слагаемое, $c(n)c(n-1)...c(2)c(1)$ — их сумма. Тогда алгоритм сложения включает в себя следующие шаги:

1) сбросить флажок переноса CF;

2) повторить следующий цикл n раз ($i=1, ..., n$ — переменная цикла):

загрузить $a(i)$ в регистр AL;

прибавить к AL цифру $b(i)$, пользуясь командой сложения с переносом ADC;

скорректировать результат в AL командой AAA;

загрузить содержимое регистра AL в $c(i)$.

Вычитание. Пусть $a(n)a(n-1)...a(2)a(1)$ — уменьшаемое, $b(n)b(n-1)...b(2)b(1)$ — вычитаемое, $c(n)c(n-1)...c(2)c(1)$ — их разность. Алгоритм многоразрядного вычитания состоит из следующих шагов:

1) сбросить флажок переноса CF;

2) выполнить цикл n раз ($i=1...n$ — переменная цикла):

загрузить $a(i)$ в регистр AL;

вычесть из AL цифру $b(i)$, пользуясь командой вычитания с заемом SBB;

скорректировать результат в AL командой AAS;

передать содержимое регистра AL в $c(i)$.

Умножение. Пусть $a(n)a(n-1)...a(2)a(1)$ — множимое, b — одноразрядный множитель, $c(n+1)c(n)...c(2)c(1)$ — их произведение.

Умножение реализуется следующим алгоритмом:

1) сбросить старшую тетраду b;

2) сбросить цифру $c(1)$;

3) повторить следующий цикл n раз ($i=1...n$ — переменная цикла):

сбросить старшую тетраду $a(i)$;

передать $a(i)$ в регистр AL;

умножить содержимое регистра AL на b, пользуясь командой MUL;

скорректировать произведение командой AAM;

прибавить к содержимому регистра AL значение $c(i)$;

скорректировать результат сложения командой AAA;

передать содержимое регистра AL в $c(i)$;

передать содержимое регистра AH в $c(i+1)$.

Деление. Пусть $a(n)a(n-1)...a(2)a(1)$ — делимое,

b — одноразрядный делитель, $c(n)c(n-1)...c(2)c(1)$ — частное. Деление осуществляется по следующему алгоритму:

1) сбросить старшую тетраду b;

2) сбросить регистр AH;

3) выполнить следующий цикл n раз ($i=1...n$ — переменная цикла):

сбросить старшую тетраду $a(i)$;

передать $a(i)$ в регистр AL;

скорректировать содержимое регистра AX командой AAD;

разделить содержимое регистра AL на b с помощью команды DIV;

передать содержимое регистра AL в $c(i)$.

В заключение приведем несколько простых программ, иллюстрирующих применение команд арифметических операций [5, 7].

1. Пусть необходимо сложить два многобайтных числа, находящихся в памяти (в текущем сегменте данных), и записать результат на место одного из них. Предположим, что младшие байты слагаемых находятся по меньшим адресам памяти. Примем следующее распределение регистров: регистр SI содержит адрес текущего байта первого слагаемого, регистр DI адресует текущий байт второго слагаемого, которое замещается суммой, регистр CX хранит число байт слагаемых.

Программа 8. Сложение многобайтных двоичных чисел

```
ADDBN:   CLC              ; Сбросить флажок CF
REPEAT:  MOV  AL, [SI]    ; Передать байт первого слагаемого
         ADC  [DI], AL    ; Сложить с учетом переноса
         INC  SI          ; Продвинуть
         INC  DI          ;   указатели
         DEC  CX          ; Проверить окончание
         JNZ  REPEAT      ;   сложения
```

Аналогичная программа обеспечивает сложение чисел, состоящих из нескольких слов. Необходимо только заменить регистр AL регистром AX в первых двух командах и ввести еще по одной команде INC SI и INC DI для правильной коррекции указателей.

Сделаем два замечания относительно приведенной выше программы. Во-первых, с ее помощью можно суммировать как беззнаковые, так и знаковые числа. Необходимо только учитывать, что при сложении беззнаковых чисел о

переполнении сигнализирует флажок CF=1, а при сложении знаковых чисел — флажок OF=1. Во-вторых, эта программа поясняет, почему в командах инкремента и декремента не нужно модифицировать флажок CF. В программе 8 состояние флажка CF, сформированное при выполнении команды ADC, не изменяется до выполнения этой же команды при сложении следующих байт.

2. Почти такой же вид, как и программа 8, имеет программа сложения многобайтных BCD-чисел. При прежнем распределении регистров получается программа, приведенная ниже.

Программа 9. Сложение многобайтных BCD-чисел

```
ADDBCD:  CLC              ; Сбросить флажок CF
REPEAT:  MOV  AL, [SI]    ; Передать байт первого слагаемого
         ADC  AL, [DI]    ; Сложить с учетом переноса
         DAA              ; Скорректировать сумму
         MOV  [DI], AL    ; Запомнить результат
         INC  SI          ; Продвинуть указатели
         INC  DI          ;
         DEC  CX          ; Проверить окончание
         JNZ  REPEAT      ;   сложения
```

Аналогичная программа реализует сложение многобайтных чисел в формате ASCII. При прежнем распределении регистров (регистры SI и DI адресуют текущие байты чисел, а регистр CX содержит количество байт в числах) в программе 9 для перехода к ASCII-формату необходимо только заменить команду DAA командой AAA.

3. Приведенные выше программы можно трансформировать для вычитания многобайтных чисел путем замены команды ADC командой SBB и использования соответствующих команд коррекции. Рекомендуется разработать их самостоятельно.

4. Чтобы показать возможности команды умножения, приведем программу умножения 32-битных беззнаковых чисел с получением 64-битного произведения. Пусть сомножители размещаются в блоке памяти, показанном на рис. 2.27, *а*, и этот блок адресуется регистром BX. Умножение 32-битных чисел осуществляется четырьмя умножениями 16-битных сомножителей и суммированием частичных произведений с учетом их местоположения в полном произведении. Предполагается, что в исходном состоянии байты произведения содержат нули.

b — одноразрядный делитель, $c(n)c(n-1)...c(2)c(1)$ — частное. Деление осуществляется по следующему алгоритму:

1) сбросить старшую тетраду b;

2) сбросить регистр AH;

3) выполнить следующий цикл n раз ($i=1...n$ — переменная цикла):

сбросить старшую тетраду $a(i)$;

передать $a(i)$ в регистр AL;

скорректировать содержимое регистра AX командой AAD;

разделить содержимое регистра AL на b с помощью команды DIV;

передать содержимое регистра AL в $c(i)$.

В заключение приведем несколько простых программ, иллюстрирующих применение команд арифметических операций [5, 7].

1. Пусть необходимо сложить два многобайтных числа, находящихся в памяти (в текущем сегменте данных), и записать результат на место одного из них. Предположим, что младшие байты слагаемых находятся по меньшим адресам памяти. Примем следующее распределение регистров: регистр SI содержит адрес текущего байта первого слагаемого, регистр DI адресует текущий байт второго слагаемого, которое замещается суммой, регистр CX хранит число байт слагаемых.

Программа 8. Сложение многобайтных двоичных чисел

```
ADDBN:  CLC                 ; Сбросить флажок CF
REPEAT: MOV   AL, [SI]      ; Передать байт первого слагаемого
        ADC   [DI], AL      ; Сложить с учетом переноса
        INC   SI            ; Продвинуть
        INC   DI            ;   указатели
        DEC   CX            ; Проверить окончание
        JNZ   REPEAT        ;   сложения
```

Аналогичная программа обеспечивает сложение чисел, состоящих из нескольких слов. Необходимо только заменить регистр AL регистром AX в первых двух командах и ввести еще по одной команде INC SI и INC DI для правильной коррекции указателей.

Сделаем два замечания относительно приведенной выше программы. Во-первых, с ее помощью можно суммировать как беззнаковые, так и знаковые числа. Необходимо только учитывать, что при сложении беззнаковых чисел о

переполнении сигнализирует флажок CF=1, а при сложении знаковых чисел — флажок OF=1. Во-вторых, эта программа поясняет, почему в командах инкремента и декремента не нужно модифицировать флажок CF. В программе 8 состояние флажка CF, сформированное при выполнении команды ADC, не изменяется до выполнения этой же команды при сложении следующих байт.

2. Почти такой же вид, как и программа 8, имеет программа сложения многобайтных BCD-чисел. При прежнем распределении регистров получается программа, приведенная ниже.

Программа 9. Сложение многобайтных BCD-чисел

```
ADDBCD: CLC                 ; Сбросить флажок CF
REPEAT: MOV   AL, [SI]      ; Передать байт первого слагаемого
        ADC   AL, [DI]      ; Сложить с учетом переноса
        DAA                 ; Скорректировать сумму
        MOV   [DI], AL      ; Запомнить результат
        INC   SI            ; Продвинуть указатели
        INC   DI            ; Проверить окончание
        DEC   CX            ;   сложения
        JNZ   REPEAT
```

Аналогичная программа реализует сложение многобайтных чисел в формате ASCII. При прежнем распределении регистров (регистры SI и DI адресуют текущие байты чисел, а регистр CX содержит количество байт в числах) в программе 9 для перехода к ASCII-формату необходимо только заменить команду DAA командой AAA.

3. Приведенные выше программы можно трансформировать для вычитания многобайтных чисел путем замены команды ADC командой SBB и использования соответствующих команд коррекции. Рекомендуется разработать их самостоятельно.

4. Чтобы показать возможности команды умножения, приведем программу умножения 32-битных беззнаковых чисел с получением 64-битного произведения. Пусть сомножители размещаются в блоке памяти, показанном на рис. 2.27, *а*, и этот блок адресуется регистром BX. Умножение 32-битных чисел осуществляется четырьмя умножениями 16-битных сомножителей и суммированием частичных произведений с учетом их местоположения в полном произведении. Предполагается, что в исходном состоянии байты произведения содержат нули.

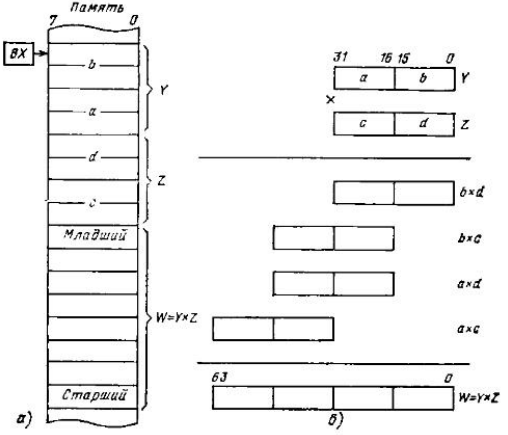


Рис. 2.27. Пояснение программы умножения

Программа 10. Умножение 32-битных сомножителей

```
        MOV     AX, [BX]        ; Умножение
        MUL     [BX+4]          ;   b * d
        MOV     [BX-8],AX       ; Запоминание
        MOV     [BX+10],DX      ;   произведения b * d
        MOV     AX, [BX]        ; Умножение
        MUL     [BX+6]          ;   b * c
        ADD     [BX+10],AX      ; Сложение
        ADC     [BX+12],DX      ;   b * d и b * c
        JNC     NEXT            ; Возник перепос?
        INC     [BX+14]         ; Да, учесть в старшем слове
NEXT:   MOV     AX, [BX+2]      ; Умножение
        MUL     [BX+4]          ;   a * d
        ADD     [BX+10],AX      ; Прибавление к полному
        ADC     [BX+12],DX      ;   произведению
        JNC     HIGH            ; Возник перепос?
        INC     [BX+14]         ; Да, учесть в старшем слове
HIGH:   MOV     AX, [BX+2]      ; Умножение
        MUL     [BX-6]          ;   a * c
        ADD     [BX+12],AX      ; Прибавление к полному
        ADC     [BX+14],DX      ;   произведению
```


5. Последующие примеры иллюстрируют применение команд сравнения. Пусть требуется подсчитать число символов в цепочке при следующем распределении функций регистров:

SI адресует текущий символ (байт) в цепочке;

AH содержит код символа, идентифицирующего конец цепочки (например, символ NULL с нулевым кодом);

CX является счетчиком.

Программа 11. Подсчет символов в цепочке

```
        MOV     CX, —1          ; Начальное значение — 1
MORE:   INC     CX              ; Инкремент счетчика
        INC     SI              ; Продвижение указателя
        CMP     AH, [SI—1]      ; Сравнение с терминатором
        JNZ     MORE            ; Проверка окончания цикла
```

Здесь предполагается, что при инкременте SI невозможен переход от FFFF к 0000 (цепочка не переходит от конца сегмента данных к началу).

6. Приводимая ниже программа предназначена для нахождения максимального 8-битного беззнакового числа в массиве, адресуемом регистром SI. Регистр CX содержит число элементов в массиве. По окончании программы в регистре AL находится максимальное значение, а в регистре DX — его адрес в массиве.

Сначала в качестве максимального принимается первый элемент массива, а затем с ним сравниваются следующие элементы. Если текущий элемент больше ранее найденного максимума, он замещает его в регистре AL.

Программа 12. Поиск максимального 8-битного беззнакового числа

```
NEW:    MOV     AL, [SI]        ; Загрузка максимума
        MOV     DX, SI          ; Индекс максимума
NEXT:   DEC     CX              ; Проверка окончания поиска
        JZ      DONE            ;
        INC     SI              ; Продвижение указателя
        CMP     AL, [SI]        ; Сравнение с максимумом
        JB      NEW             ; Новый максимум
        JMP     NEXT            ; Продолжение поиска
DONE:
```

7. Следующая программа предназначена для нахождения максимального числа в массиве 16-битных знаковых чисел. Регистр SI адресует текущий элемент массива, а регистр CX содержит число слов в массиве. Сначала за максимум принимается число 8000 (—32 768), а затем с ним сравниваются следующие слова массива и при необходимости производится замена максимума.

Программа 13. Поиск максимального 16-битного знакового числа

```
        MOV     BX, 8000H       ; Первый максимум
MORE:   MOV     AX, [SI]        ; Текущий элемент
        CMP     BX, AX          ; Сравнение с максимумом
        JGE     NOEXC           ; Заменять не нужно
        MOV     BX, AX          ; Новый максимум
        MOV     DX, SI          ;   и его указатель
NOEXC:  INC     SI              ; Продвижение указателя
        INC     SI
        DEC     CX              ; Проверка окончания
        JNZ     MORE            ;   поиска
```

2.3.3. Команды логических операций и команды сдвигов

Логические операции, реализуемые в МП К1810, представлены булевыми операторами NOT (инверсия), AND (конъюнкция), OR (дизъюнкция), XOR (исключающее ИЛИ, т. е. сложение по модулю 2) и командой TEST, которая выполняет конъюнкцию операндов, но не изменяет их значений (неразрушающая проверка). Все логические операции являются поразрядными, т. е. выполняются независимо для всех бит операндов.

Обобщенное представление команд логических операций имеет следующий вид:

AND	*dst*, *src*	$dst := (dst) \wedge (src)$
OR	*dst*, *src*	$dst := (dst) \vee (src)$
XOR	*dst*, *src*	$dst := (dst) \oplus (src)$
TEST	*dst*, *src*	$(dst) \wedge (src)$
NOT	*src*	$src := \overline{(src)}$

Бинарные команды AND, OR, XOR и TEST воздействуют на арифметические флажки следующим образом:

флажки OF и CF всегда переводятся в нулевое состояние, так как межразрядные связи при выполнении операций отсутствуют;

состояния флажков SF, ZF и PF зависят от полученного результата и определяются по тем же правилам, что и в командах арифметических операций;

состояние флажка AF не определено.

Унарная команда инверсии NOT не влияет на состояние флажков.

Команда поразрядной конъюнкции AND в основном применяется для перевода в нулевое состояние тех бит операнда, которые определяются другим операндом — маской. Маска должна содержать нули в сбрасываемых битах и единицы в остальных.

Команда поразрядной дизъюнкции OR применяется для установки в 1 определенных бит операнда с помощью маски, а также упаковки байт или слов из полей других элементов данных.

С помощью команды исключающего ИЛИ XOR можно инвертировать определенные биты операнда (на основе тождества $1 \oplus x = \bar{x}$), сравнивать операнды на абсолютное равенство и переводить регистр в нулевое состояние (на основе тождества $x \oplus x = 0$).

Форматы команд логических операций представлены на рис. 2.28.

Как видно из рисунка, в каждом из трех форматов команд AND, OR, XOR и TEST допускаются операции над

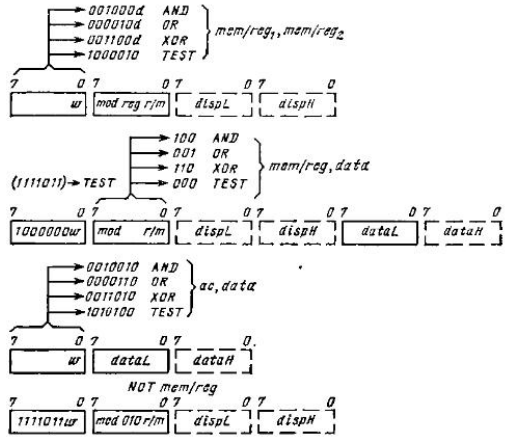


Рис. 2.28. Форматы команд логических операций

байтами и словами (об этом свидетельствует наличие бита *w* в коде операции).

Наиболее гибким является формат с операндами в виде mem/reg_1, mem/reg_2. Он дает возможность определять опе-

рации регистр—регистр/память и память—регистр с широким набором режимов адресации памяти. Длина команд в этом формате может быть 2—4 байта, а время выполнения составляет (в тактах синхронизации):

Р... ...р — регистр	3
Р... ... — регистр	9+EA
Р... ... — память	16+EA

Пример 44.

(AX)=3456 (SI)=F260 (DS)=EA40 ([F2670]=AAAA	AND AX,[SI + 10H)	(AX)=2002 OF, SF, ZF, PF, CF =0

Пример 45.

(DL) = 33 (DS) = 1000 ([TEMP]) = BC	OR TEMP,DL	([TEMP]) = BF OF, ZF, PF, CF =00 SF = 1

Пример 46.

(AX) = 7777 (DX) = BCDE	XOR AX,DX	(AX) = CB49 OF, ZF, CF =0 SF, PF = 1

Пример 47.

(CX) = 1234 (SI) = 6000 (DS) = CC00 ([D2000]) =0010	TEST [SI],CX	OF, SF, ZF, PF, CF =0

Команды логических операций в формате, имеющем опе... *mem/reg, data*, выполняют заданную операцию над ...ным адресуемых регистра или ячейки памяти и не...ственными данными, находящимися в команде. Дли... команд составляет 3—6 байт, а время выполнения равно ...ором тактам, если определен регистр, и (17+EA) так... ..., если определена память. Время выполнения команды ... SI при заданин памяти несколько меньше и равно (10+ ...) тактам, так как результат нигде не запоминается.

Пример 48.

(DI) = 3500 (BP) = 6000 (SS) = A800 ([B1500]) = 39AF	AND [BP][DI], 0F000H	([B1500]) = 3000 OF, SF, ZF, CF = 0 PF = 1

Пример 49.

(DX) = 382F	OR DX, 0F0F0H	(DX) = F8FF OF, ZF, CF =0 SF, PF = 1

Пример 50.

(BX) = FF20 (DS) = 2C00 ([3BF20]) = 375F	XOR [BX], 00FH	([3BF20]) = 3750 OF, SF, ZF, CF = 0 PF = 1

Пример 51.

(DI) = 825\ (DF) = CFC0 ([D7D7A])=780A	TEST [DI + 20H], 8000H	OF, SF, CF = 0 ZF, PF = 1

Наконец, формат с операндами *ac, data* представляет собой укороченный формат предыдущей команды. Команды с этим форматом имеют длину 2 или 3 байта и выполняются за четыре такта синхронизации. В операции над байтами аккумулятором служит регистр AL, а в операции над словами — регистр AX.

Команда NOT *mem/reg* предназначена для инвертирования содержимого регистра или ячейки памяти. Она имеет длину 2—4 байта и выполняется за три и (16+EA) тактов при определении регистра и памяти соответственно.

Команда NOT не изменяет состояний флажков.

Пример 52.

(CX) = 5678 NOT CX (CX) = A987.

Пример 53.

(SI) = 8800 (DS) = CC00 ([D4900]) = 0F0F	NOT [SI + 100H)	([D4900]) = F0F0

Команды сдвигов. Форматы команд сдвигов МП К1810 представлены на рис. 2.29. Коды операций всех команд сдвигов содержат бит *w*, и, следовательно, имеется возможность сдвигать байты и слова. Поле операнда имеет вид *mem/reg, count*. Здесь *mem/reg* стандартным образом адресует общий регистр или ячейку памяти, а *count* (счет или счетчик) определяет число сдвигов. Этот операнд может быть указан как константа 1 (статический сдвиг) или как регистр CL. В первом случае осуществляется сдвиг на 1 бит, а во втором — число сдвигов определяется содержимым регистра CL, которое должно являться беззнаковым целым двоичным числом. Таким образом, число сдвигов можно задать переменной, вычисляемой во время выполнения про-

граммы (так называемый динамический сдвиг). Выбор константы 1 или регистра CL идентифицирует бит v в коде операции: если $v=0$, то $count=1$, а если $v=1$, то $count=(CL)$.

Команды сдвигов подразделяются на команды циклических сдвигов (ротаций) и «обычных». В циклических

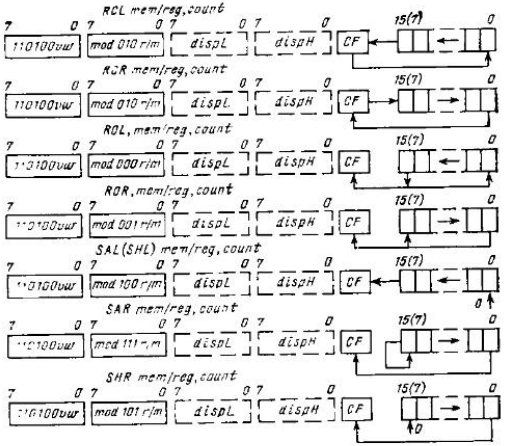


Рис. 2.29. Форматы команд сдвигов

сдвигах выдвигаемый бит помещается на место освобождающегося бита.

При выполнении команд сдвигов флажки модифицируются следующим образом:

состояние флажка AF всегда не определено;

флажок CF всегда содержит значение последнего выдвинутого бита;

в однобитных сдвигах флажок OF=1, если операция изменила значение старшего (знакового) бита операнда; при сдвиге на несколько бит состояние флажка OF не определено;

циклические сдвиги влияют только на флажки OF и CF;

в «обычных» сдвигах флажки SF, ZF и PF модифицируются в соответствии с полученным результатом.

Все команды сдвигов имеют длину 2—4 байта, а время их выполнения в тактах синхронизации равно:

$count=1$	регистр	2
	память	10 + EA
$count=(CL)$	регистр	8+4 * (CL)
	память	20 + EA + 4 * (CL)

Первые четыре команды на рис. 2.29 реализуют циклические сдвиги. Команды RCL и RCR называются командами циклического сдвига влево и вправо через перенос, так как флажок CF расширяет сдвигаемый операнд на 1 бит. В МП К1810 можно выполнить сдвиги содержимого любого общего регистра и байта или слова в памяти на любое число бит вплоть до максимума, равного 255.

Примечание. Из синтаксиса команд сдвига с адресацией памяти, как и других унарных операций, невозможно определить, подлежит сдвигу байт или слово, например:

```
RCL   [SI],CL
ROR   [BP][DI],1
RCR   [DI+10H],CL
```

Вопрос об идентификации формата операнда рассмотрен в § 3.5. В приводимых ниже примерах команд сдвигов предполагается, что операндом в памяти является байт.

Пример 54.

(DL) = 3A (CF) = 1	RCL DL,1	(DL) = 75 OF, CF = 0

Пример 55.

(SI) = 1500 (DS) = EF00 (CL) = 03 (CF) = 1 ([F0500]) = 4C	RCR [SI],CL	([F0500]) = 29 CF = 1 OF = ?

Пример 56.

(BX) = A108 (CL) = 05	ROL BX,CL	(BX) = 2114 CF = 0 OF = ?

Пример 57.

(BP) = 2000 (SS) = 4F00 ([51040]) = 75	ROR [BP+40H],1	([51040]) = BA OF, CF = 1

Команды SHL и SHR реализуют логнческнй сдвиг влево и вправо соответственно. Для логического сдвига характерно, что в освобождающийся бит загружается нуль, а выдвигаемый бит теряется.

Команды SAL и SAR предназначены для арифметического сдвига влево и вправо. Арифметический сдвиг вправо отличается от логического сдвига тем, что знаковый бит не сдвигается, а тиражируется в соседнем правом бите, сохраняя знак числа. Арифметический сдвиг влево в дополнительном коде не отличается от логического сдвига. Поэтому мнемоники SAL и SHL обозначают одну и ту же машинную команду и введены для удобства программирования.

Команды арифметического сдвига по существу реализуют умножение и деление чисел на степень двух. Отметим, однако, что команда SAR не дает такого же результата, как деление командой IDIV в том случае, когда делимое отрицательное, а при сдвиге выдвигаются единичные биты. Например, сдвиг —10 (десять) на 2 бита вправо дает результат —3, а при делении —10 на 4 получается частное —2. Отличие в результатах объясняется тем, что команда IDIV усекает (округляет путем отбрасывания дробной части) все частные к нулю, а команда SAR усекает положительные числа к нулю, а отрицательные — к минус бесконечности.

Пример 58.

(AX) = 9A58	SAL AX,1	(AX) = 34B0 OF, CF = 1 SF, ZF, PF = 0

Пример 59.

(BP) = 3890 (SS) = CF00 (CL) = 05 ([D1910]) = 84	SAR [BP + 80H],CL	([D1910]) = FC SF, PF = 1 ZF, CF = 0 OF = ?

Пример 60.

(DH) = 93	SHR DH,1	(DH) = 49 OF, CF = 1 SF, ZF, PF = 0

Приведем несколько простых программ, иллюстрирующих применение команд логических операций и команд сдвигов.

1. Предположим, что в области памяти, адресуемой регистром SI, находится цепочка 7-битных кодов символов, заканчивающаяся символом NULL. Старшие биты всех кодов символов нулевые. Необходимо записать в старший бит

каждого символа контрольный разряд четности, а на место символа NULL — байт продольной четности всего блока символов. Требуемые действия выполняет следующая программа.

Программа 14. Формирование контрольных бит

```
        XOR   AH,AH     ; Сбросить регистр AH
START:  MOV   AL,[SI]   ; Передать в AL код символа
        AND   AL, AL    ; Установить флажки
        JZ    EXIT      ; Символ NULL — конец цепочки
        JPE   NON       ; Четное число единиц?
        OR    AL,80H    ; Нет, записать 1 в старший бит
NON:    MOV   [SI],AL   ; Вернуть символ в цепочку
        XOR   AH,AL     ; Учесть в контрольном байте
        INC   SI        ; Продвинуть указатель
        JMP   START     ; Повторить цикл
EXIT:   MOV   [SI],AH   ; Записать контрольный байт
```

2. В п. 2.3.2 был приведен алгоритм умножения многобайтных ASCII-чисел на одну ASCII-цифру. Составим программу, реализующую этот алгоритм, при следующем распределении регистров: регистр SI адресует текущий байт множимого, регистр DL содержит множитель, регистр DI адресует текущий байт произведения, а содержимое регистра CX определяет число цифр в множимом.

Программа 15. Умножение многобайтного ASCII-числа

```
        MOV   [DI],0    ; Сбросить первый байт произве-
                        ;   дения
        AND   DL,0FH    ; Скорректировать множитель
MULT:   MOV   AL,[SI]   ; Очередная цифра множимого
        INC   SI        ; Продвинуть указатель
        AND   AL,0FH    ; Сбросить старшую тетраду
        MUL   DL        ; Умножить
        AAM             ; Скорректировать произведение
        ADD   AL,[DI]   ; Прибавить в произведение
        AAA             ; Скорректировать сумму
        MOV   [DI],AL   ; Запомнить результат
        INC   DI        ; Продвинуть указатель
        MOV   [DI],AH   ; Старшая цифра произведения
        DEC   CX        ; Проверить окончание умноже-
        JNZ   MULT      ;   ния
```

3. Приведенная ниже программа реализует алгоритм деления многобайтного ASCII-делимого на одну ASCII-цифру. Предполагается, что регистры SI, DI и CX выполняют те же функции, как и в предыдущей программе. Отличия заключаются в том, что SI сначала должен адресовать старший байт делимого; частное получается, начиная также со старшего байта, и регистр DL содержит цифру делителя.

Программа 16. Деление многобайтного ASCII-числа

```
        AND     DL,0FH      ; Сбросить старшую тетраду
        XOR     AH,AH       ; Сбросить регистр AH
DIVID:  MOV     AL,[SI]     ; Цифра делимого
        INC     SI          ; Продвинуть указатель
        AND     AL,0FH      ; Сбросить старшую тетраду
        AAD                 ; Скорректировать делимое
        DIV     DL          ; Разделить на делитель
        MOV     [DI],AL     ; Запомнить цифру частного
        INC     DI          ; Продвинуть указатель
        DEC     CX
        JNZ     DIVID       ; Проверить окончание
                            ;   деления
```

4. Как отмечалось выше, команды арифметических сдвигов удобно применять для умножения (сдвиг влево) и деления (сдвиг вправо) на степени двух и близкие к ним числа. Прямое использование для этого команд умножения MUL и IMUL иногда нецелесообразно из-за потерь времени, так как минимальное время их выполнения составляет 71 такт синхронизации.

Пусть, например, содержимое регистра AL нужно умножить на 8. С помощью сдвигов такое умножение реализуется командами

```
MOV     CL,3
SAL     AL,CL
```

Эти две команды выполняются за 24 такта. Возможно несколько сократить время умножения, воспользовавшись тремя командами:

```
SAL     AL,1
SAL     AL,1
SAL     AL,1
```

которые выполняются всего за шесть тактов синхронизации.

Нетрудно составить аналогичную программу умножения содержимого регистра AX на 15.

Программа 17. Умножение (AX) на 15

```
MOV     CL,4        ; Константа сдвига
MOV     DX,AX       ; Сохранить (AX)
SAL     AX,CL       ; Умножить (AX) на 16
SUB     AX,DX       ; Скорректировать для 15
```

Данный фрагмент выполняется за 33 такта синхронизации. Для ускорения умножения воспользуемся четырьмя командами SAL:


```
MOV     DX,AX       ; Сохранить (AX)
SAL     AX,1        ; Умножить (AX) на 16
SAL     AX,1
SAL     AX,1
SAL     AX,1
SUB     AX,DX       ; Скорректировать для 15
```

Время выполнения этого фрагмента составляет всего 13 тактов синхронизации.

2.3.4. Команды передачи управления

При выполнении линейных программных фрагментов операционное устройство выбирает байты из очереди команд и интерпретирует их в соответствии с семантикой команд. Как только в очереди появляются два свободных байта, устройство шинного интерфейса инициирует цикл шины, выбирает из программной памяти очередное слово и помещает его в очередь команд. При этом соответственно корректируется программный счетчик PC. Такие действия соответствуют естественному порядку выполнения команд. Физический адрес программной памяти определяется содержимым сегментного регистра кода CS и программного счетчика PC.

На практике обойтись только линейными программами невозможно. В разветвляющихся и циклических программах, а также при организации подпрограмм необходимо выполнять не следующую по порядку команду, а команду, находящуюся в другой ячейке программной памяти и определяемую адресом перехода. Специальные команды, которые модифицируют указатели программной памяти (регистры PC и CS), называются командами передачи управления. В системе команд МП К1810 имеется обширный набор таких команд. Данные команды не изменяют состояний флажков, за исключением команды возврата из прерывания IRET.

Сегментная организация программной памяти определяет две основные разновидности команд передачи управления. Передача управления в пределах текущего сегмента кода (программы) называется внутрисегментной — при этом модифицируется только PC и адрес перехода представляется одним словом. Такая передача управления называется еще близкой (тип NEAR), а ее вариант с сокращенным диапазоном адресов переходов короткой. Передача управления за пределы текущего сегмента кода называется меж-

сегментной — при этом необходимо модифицировать содержимое регистров PC и CS и адрес перехода представляется двумя словами (сегмент : смещение). Данная передача управления называется еще далекой (тип FAR), так как она позволяет перейти к любой ячейке адресного пространства памяти.

Команды передачи управления МП К1810 подразделяются на команды безусловных переходов, условных переходов, вызовов, возвратов, управления циклами и команды прерываний.

Команды безусловных переходов. При выполнении команд безусловных переходов происходит модификация PC или PC и CS, а их прежнее содержимое теряется. Форматы данных команд, упорядоченные по расширению области перехода, представлены на рис. 2.30. Первые три команды имеют тип NEAR, а две последние команды тип FAR.

Двухбайтная команда JMP *dispL* содержит во втором байте смещение, которое интерпретируется как знаковое целое. При выполнении команды значение смещения прибавляется (с расширением знака до 16 бит) к содержимому PC, которое соответствует адресу команды, находящейся после команды JMP. Диапазон значений байта смещения составляет —128÷+127. Если смещение положительное, осуществляется переход вперед, а если отрицательное — переход назад.

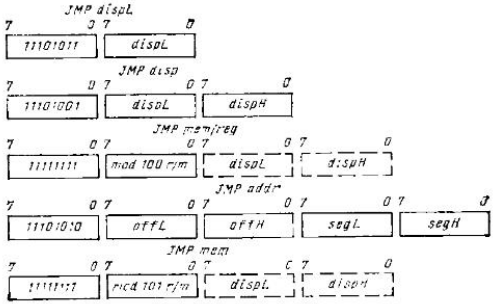


Рис. 2.30. Форматы команд безусловных переходов

Данная команда удобна для организации коротких программных циклов. Относительная адресация обеспечивает позиционную независимость, так как значение смещения не зависит от положения программы в памяти.

Команда JMP *dispL* выполняется за 15 тактов синхронизации.

Примечания: 1. Время выполнения команд передачи управления не учитывает реинициализацию очереди команд. Это действие увеличивает время выполнения на четыре такта синхронизации.

2. В ассемблерных программах операндом команд передачи управления является метка, ассоциируемая с той командой, которой передается управление. В приводимых ниже примерах условно дается численное значение операнда в том формате, в каком оно хранилось бы в памяти.

3. Начальным состоянием PC считается адрес команды, находящейся после команды передачи управления.

Пример 61.

(PC) = 1240 JMP E8 (PC) = 1228

Трехбайтная команда JMP *disp* производит такое же действие, как предыдущая команда, но содержит 16-битное смещение. Оно по-прежнему интерпретируется как знаковое целое, поэтому область перехода увеличивается до —32 768÷+32 767 байт относительно адреса команды, находящейся после команды JMP *disp*. Время выполнения этой команды также равно 15 тактам.

Пример 62.

(PC) = 3E60 JMP 0002 (PC) = 4060

Команда JMP *mem/reg* реализует косвенный безусловный переход в программе. Здесь адресом перехода, загружаемым в PC, служит содержимое 16-битного общего регистра или слова памяти, определяемое постбайтом режима адресации. Отметим, что необязательные байты *displ.*, *dispH* в команде относятся к режиму адресации памяти.

Длина команды может быть 2—4 байта, а время выполнения равно 11 тактам при указании регистра и (18+EA) тактам при указании памяти.

Пример 63.

(BX) = 3000 JMP BX (PC) = 3000

Пример 64.

(BX) = 68A0 JMP [BX] (PC) = 3560
(DS) = AA00
([B08A0]) = 3560

Последние две команды JMP (см. рис. 2.30) реализуют прямой и косвенный межсегментные переходы, т. е. допускают передачу управления любой ячейки в адресном пространстве памяти. Команда JMP *addr* содержит 4 байта прямого адреса перехода, которые определяют новое содержимое регистров PC и CS: значение *off* загружается в PC, а значение *seg* — в регистр CS. Время выполнения этой 5-байтной команды составляет 15 тактов синхронизации.

Пример 65.

(PC) = 18A6	JMP 0020 A040	(PC) = 2000
(CS) = 0200		(CS) = 40A0

В команде косвенного межсегментного перехода JMP *mem* допускается адресация только памяти. Слово из адресуемой ячейки памяти загружается в PC, а следующее слово — в регистр CS. Длина команды может быть 2—4 байта, а время выполнения равно (24+EA) тактам синхронизации.

Пример 66.

(DS) = 6000	JMP[DI + 100H]	(PC) = 3750
(DI) = 2680		(CS) = F000
([62780]) = 3750		
([62782]) = F000		

Команды условных переходов. В системе команд МП К1810 есть 19 двухбайтных команд условных переходов, называемых также разветвлениями. Все они имеют единый формат, представленный на рис. 2.31. При выполнении этих

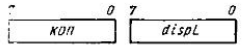


Рис. 2.31. Формат команд условных переходов

команд анализируется некоторое условие, закодированное текущими состояниями флажков (а в команде JCXZ — содержимым регистра CX), и в зависимости от удовлетворения условия переход осуществляется или нет. Данные команды позволяют проверить оба состояния всех флажков арифметических операций (кроме флажка AF), а также ряд комбинаций состояний нескольких флажков. Если условие истинно, управление передается по адресу перехода путем прибавления к содержимому PC однобайтного знакового смещения (с расширением знака до 16 бит), а если условие ложно, выполняется следующая по порядку команда. Таким образом, все условные переходы в МП К1810 являются короткими. Время выполнения команд составляет

четыре такта (переход не осуществляется) или восемь тактов (переход осуществляется) синхронизации.

Следует отметить, что большинство команд условных переходов имеет две (и даже три) мнемоники, подчеркивающие содержательный смысл проверяемого условия и введенные для удобства программирования.

Команды позволяют проверить все отношения между знаковыми и беззнаковыми числами. Фигурирующие в определении команд термины «больше» и «меньше» относятся к знаковым числам, представленным в дополнительном коде, а «выше» и «ниже» — к беззнаковым. Например, число BE «меньше» и «выше» числа 37.

Достаточно подробная информация о всех командах условных переходов содержится в табл. 2.7.

Таблица 2.7. Команды условных переходов

Мнемоника	Условие	Отношение	КОП	Функция
				Перейти, если
JA/JNBE	CF∨ZF=0	>	77	выше/не ниже или равно
JAE/JNB	CF=0	≥	73	выше или равно/не ниже
JB/JNAE	CF=1	<	72	ниже/не выше или равно
JBE/JNA	CF∨ZF=1	≤	76	ниже или равно/не выше
JC	CF=1		72	есть перенос
JE/JZ	ZF=1	=	74	равно/нуль
JG/JNLE	(*)	>	7F	больше/не меньше или равно
JGE/JNL	SF⊕OF=0	≥	7D	больше или равно/не меньше
JL/JNGE	SF⊕OF=1	<	7C	меньше/не больше или равно
JLE/JNG	(**)	≤	7E	меньше или равно/не больше
JNC	CF=0		73	нет переноса
JNE/JNZ	ZF=0	≠	75	не равно/не нуль
JNO	OF=0		71	нет переполнения
JNP/JPO	PF=0		7B	нет паритета/паритет нечетный
JNS	SF=0		79	нет знака
JO	OF=1		70	есть переполнения
JP/JPE	PF=1		7A	есть паритет/паритет четный
JS	SF=1		78	есть знак
JCXZ	(CX)=0		E3	содержимое регистра CX=0

(*) (SF⊕OF)∨ZF=0
(**) (SF⊕OF)∨ZF=1

Пример 67.

```
(PC) = 3560     JNC A0      (PC) = 3590
(CF) = 0
```

Пример 68.

```
(PC) = A000     JCXZ 80     (PC) = A000
(CX) = 0005
```

Команду JCXZ удобно помещать в начале цикла, особенно в том случае, если возможна ситуация, при которой цикл (со счетчиком CX) не выполняется ни разу.

При программировании может возникнуть необходимость передачи управления за пределы действия команд условного перехода. Пусть, например, при ZF=1 необходимо передать управление команде с меткой MORE, которая находится через 0400 байт от данной точки в программе. В этом случае приходится использовать две команды:

```
        JNZ     NEXT    ; Флажок ZF=0
        JMP     MORE    ; Флажок ZF=1
NEXT:
```

Ассемблер сформирует команду JMP с двухбайтным смещением.

Команды вызовов (подпрограмм). Команда вызова подпрограммы CALL передает управление с автоматическим сохранением адреса возврата. В поле операнда этой команды находится метка первой команды вызываемой подпрограммы. При переходе к подпрограмме необходимо временно запомнить адрес команды, находящейся после команды CALL. Этот адрес называется адресом возврата. После того, как подпрограмма закончит свои действия, завершающая ее команда RET возврата передает управление по запомненному адресу возврата. Удобным местом хранения адресов возвратов, который обеспечивает очень простую организацию вложенных подпрограмм, является стек.

Сегментная организация памяти МП К1810 влияет на реализацию команд вызовов. Как и команды безусловного перехода, вызовы могут быть внутрисегментными и межсегментными. В первом случае вызываемая подпрограмма находится в текущем сегменте кода (тип (NEAR), а во втором — в произвольном (тип FAR). В соответствии с этим в стеке приходится запоминать содержимое либо только PC, либо PC и CS.

Форматы команд вызовов показаны на рис. 2.32. Как видно, команды CALL имеют такие же форматы, как и

команды безусловных переходов (см. рис. 2.30); отсутствует только формат CALL *dispL*. По воздействию на регистры PC и CS команды CALL также соответствуют командам JMP, но дополнительно они запоминают в текущем сегменте стека (адресуемом регистром SS) адрес возврата с соот-

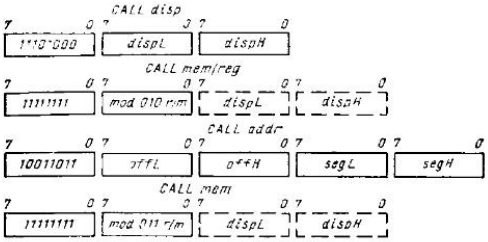


Рис. 2.32. Форматы команд вызовов подпрограмм

ветствующей модификацией указателя стека SP. Напомним, что включение в стек сопровождается декрементом указателя стека и что SP адресует последние включенные в стек данные. Первые две команды CALL на рис. 2.32 реализуют внутрисегментные вызовы, а последние две — межсегментные вызовы подпрограмм.

Трехбайтная команда CALL *disp* производит декремент SP на 2, включает в стек содержимое PC, а затем прибавляет к PC 16-битное смещение, которое интерпретируется как знаковое целое. Все эти действия выполняются за 19 тактов синхронизации.

Пример 69.

```
(PC) = 3A08     CALL  0A08    (PC) = 4212
(SP) = 1804                   (SP) = 1802
(SS) = A000                   ([A1802]) = 3A08
```

Команда CALL *mem/reg* осуществляет внутрисегментный косвенный вызов, причем источником адреса перехода может быть определен 16-битный общий регистр или ячейка памяти. Длина этой команды составляет 2—4 байта, а время выполнения равно 16 тактам, если определен регистр, и (21+EA) тактам, если определена память.

Пример 70.

(PC) = 8AAA	CALL AX	(PC) = A020
(SP) = 0800		(SP) = 07FE
(SS) = BBB0		([BC2FE]) = 8AAA
(AX) = A020		

Команда прямого межсегментного вызова CALL *addr* имеет длину 5 байт и позволяет вызвать подпрограмму, находящуюся в любой области адресного пространства памяти. Этой командой выполняются следующие действия:

содержимое SP уменьшается на 2;

в адресуемую регистрами SP и SS ячейку памяти пересылается содержимое CS;

содержимое SP уменьшается на 2;

в адресуемую регистрами SP и SS ячейку памяти записывается содержимое PC;

в PC загружается смещение *off*;

в CS загружается сегментный адрес *seg*.

Указанные действия выполняются за 28 тактов синхронизации.

Пример 71.

(PC) = 3700	CALL 2AA0 84BC	(SP) = 1734
(CS) = 6000		(PC) = A02A
(SP) = 1738		(CS) = BC84
(SS) = 8000		([81734]) = 3700
		([81736]) = 6000

Команда CALL *mem* осуществляет косвенный межсегментный вызов подпрограмм через память. Текущее содержимое регистров PC и CS запоминается в стеке, после чего слово из адресуемой ячейки памяти загружается в PC, а следующее слово — в регистр CS. Длина команды может быть 2—4 байта (в зависимости от режима адресации памяти), а время выполнения составляет (37÷EA) тактов синхронизации. Если *mod*=11, то операция не определена.

Пример 72.

(PC) = 7240	CALL [BX][SI]	(PC) = 8790
(CS) = 2000		(CS) = A060
(SP) = 028A		(SP) = 0286
(SS) = 3FF0		([40186]) = 7240
(BX) = 0400		([40188]) = 2000
(SI) = 1200		
(DS) = 1800		
([19600]) = 8790		
([19602]) = A060		

Действие этой команды схематически показано на рис. 2.33.

В МП К1810 нет команд условных вызовов подпрограмм, и поэтому при необходимости механизм условных вызовов реализуется двумя командами. Например, если

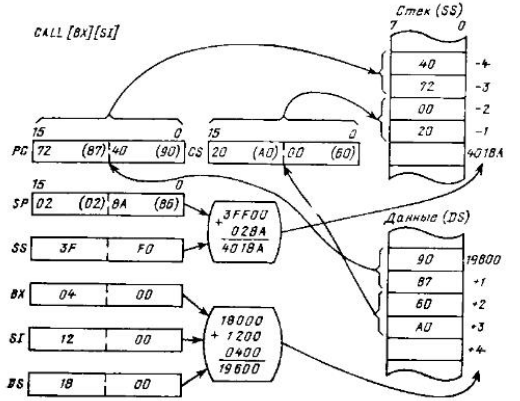


Рис. 2.33. Действие команды косвенного межсегментного вызова через память

требуется вызвать подпрограмму SUBR при ненулевом результате операции (CNZ), то это выполняется следующим образом:

```
        JZ      NEXT
        CALL    SUBR
NEXT:
```

В зависимости от того, определен тип подпрограммы SUBR как NEAR или FAR, ассемблер сформирует команду внутрисегментного или межсегментного вызова.

Команды возвратов (из подпрограмм). Каждая подпрограмма должна содержать минимум одну команду возврата RET, которая возвращает управление вызывающей про-

грамме. Такая передача управления осуществляется путем извлечения из стека адреса возврата, включенного в него командой вызова подпрограммы. Таким образом, команда возврата не содержит никакой адресной информации, неявно адресуя вершину стека.

В соответствии с типом команды вызова (NEAR или FAR) необходимо применять команды возвратов двух типов: внутрисегментные и межсегментные. Форматы команд возврата МП К1810 показаны на рис. 2.34.

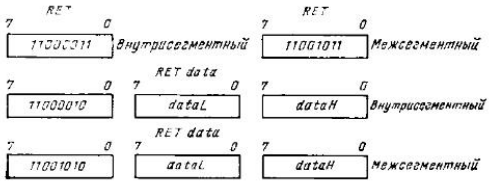


Рис. 2.34. Форматы команд возвратов из подпрограмм

Однобайтная команда RET с кодом операции C3 реализует внутрисегментный возврат. Ее действия заключаются в том, что верхнее слово стека передается в PC, а содержимое указателя стека SP увеличивается на 2. Эти действия выполняются за восемь тактов синхронизации.

Однобайтная команда RET с кодом операции CB осуществляет межсегментный возврат. Ее выполнение сводится к следующим действиям:

слово из вершины стека передается в PC;
производится инкремент SP на 2;
слово из новой вершины стека передается в регистр CS;
производится инкремент SP на 2.

В результате этих действий в регистрах PC и CS оказывается полный адрес возврата, а указатель стека адресует новую вершину. Время выполнения данной команды равно 18 тактам синхронизации.

Пример 73.

(SP) = 1080	RET	(PC) = 3740
(SS) = A900	(C3)	(SP) = 1082
([AA080]) = 3740		

Пример 74.

(SP) = 1080	RET	(PC) = 3740
(SS) = A900	(CB)	(CS) = 57A8
([AA080]) = 3740		(SP) = 1084
([AA082) = 57A8		

Еще две команды возврата имеют длину 3 байта и содержат 2 байта данных, интерпретируемых как беззнаковое целое. Первая из них реализует внутрисегментный возврат, а вторая — межсегментный. Они производят точно такие же действия, как соответствующие однобайтные команды возврата, но дополнительно прибавляют содержащиеся в них данные к указателю стека SP (после извлечения из стека адреса возврата). Эти команды предусмотрены для упрощения возврата из тех подпрограмм, параметры которых передаются в стеке. Прибавление к SP данных из команды RET эквивалентно удалению параметров из стека (подробнее см. ниже). Время выполнения команд RET *data* составляет 12 и 17 тактов синхронизации.

Пример 75.

(SP) = 29A8	RET 0600	(PC) = 1340
(SS) = A600	(CA)	(CS) = 7000
([A89A8]) = 1340		(SP) = 29B2
([A89AA]) = 7000		

Схема выполнения этой команды представлена на рис. 2.35. В микропроцессоре К1810 нет команд условного возврата, но при необходимости реализации механизма условного возврата из подпрограммы это можно сделать следующими командами:

```
        JZ      NEXT
        RET
NEXT:
```

Команды управления циклами. Три команды управления циклами (или итерациями) применяются для организации программных циклов. В них предусматривается использование регистра CX в качестве счетчика цикла. Форматы данных команд представлены на рис. 2.36. Второй байт интерпретируется как знаковое целое и при необходимости передачи управления в начале цикла прибавляется к содержимому PC. Следовательно, диапазон переходов этих команд составляет $-128 \div +127$ байт от следующей команды. В ассемблерных программах поле операнда команд управления циклами содержит метку первой команды

цикла. Метка должна находиться в диапазоне переходов, как во всех командах условной передачи управления.

Когда выполняется команда LOOP (повторить цикл), производится декремент регистра CX и, если (CX)≠0, смещение прибавляется к PC — происходит переход к началу

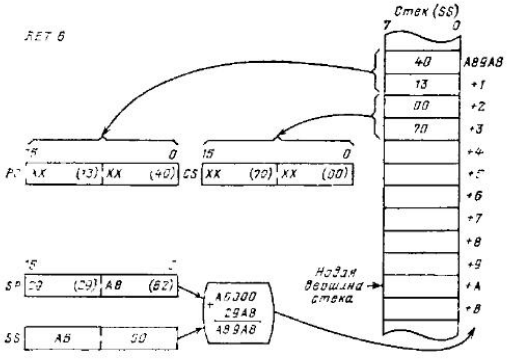


Рис. 2.35. Действие команды межсегментного возврата

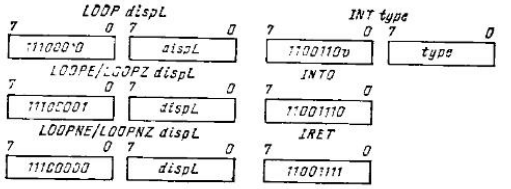


Рис. 2.36. Форматы команд повторений циклов

Рис. 2.37. Форматы команд прерываний

цикла; в противном случае [(CX)=0 и цикл окончен] выполняется следующая по порядку команда. Другими словами, команда LOOP MORE эквивалентна двум командам:


```
DEC     CX
JNZ     MORE
```

Следовательно, данную пару команд в программах 1—3 и др. можно заменить одной командой LOOP. Каждая такая замена экономит байт объектного кода и один такт, поскольку время выполнения команды LOOP составляет 5 (переход не происходит) или 17 (переход происходит) тактов синхронизации. Несущественное на первый взгляд ускорение выполнения операции проверки окончания цикла может оказаться заметным при большом числе повторений цикла.

Пример 76.

```
(PC) = 3826     LOOP 8A     (PC) = 37B0
(CX) = 0001                 (CX) = 0000
```

Мнемоники LOOPE и LOOPZ определяют одну и ту же машинную команду, которая производит декремент регистра CX, а затем передает управление в начало цикла (попрежнему путем прибавления смещения к PC), если (CX)≠0 и ZF=1. В противном случае будет выполняться следующая по порядку команда. Следовательно, по сравнению с командой LOOP данная команда вводит дополнительное условие повторения цикла — единичное состояние флажка ZF. Время выполнения составляет 6 или 18 тактов синхронизации.

Мнемоники LOOPNE и LOOPNZ также определяют одну и ту же машинную команду. Действия и время ее выполнения аналогичны команде LOOP, но дополнительным условием перехода к началу цикла является нулевое состояние флажка ZF.

Пример 77.

```
(PC) = 4004     LOOPNZ B8   (PC) = 3FBC
(CX) = 0020                 (CX) = 001F
(ZF) = 0
```

Команды прерываний. В микропроцессоре К1810 имеются три команды, относящиеся к прерываниям. Их форматы приведены на рис. 2.37. Первые две команды позволяют вызвать подпрограмму обработки прерывания примерно так же, как это делают аппаратные (внешние) запросы прерываний по входу INT, когда они разрешены, или запросы немаскируемых прерываний по входу NMI. Естественно, реагируя на программное прерывание, МП не выполняет цикла шины подтверждения прерывания.

Команда INT *type* вызывает подпрограмму обработки, определяемую типом прерывания. Тип прерывания зависит от значения бита *v* в коде операции, который, в свою очередь, зависит от того, имеется или отсутствует операнд в ассемблерной записи команды INT. Если *v*=0, второй байт команды отсутствует и тип прерывания принимается равным трем — это прерывание контрольной точки, или контрольного останова. Если *v*=1, тип прерывания задается вторым байтом команды и может принимать значение от 0 до 255.

Выполнение команды INT инициирует следующую последовательность действий:

декремент указателя стека на 2;

включение в стек содержимого регистра флажков (как в команде PUSHF);

сброс флажков IF и TF (запрещение восприятия прерываний и покомандной работы);

декремент указателя стека на 2;

включение в стек содержимого регистра CS;

определение значения ADDRESS путем умножения кода типа прерывания на 4;

загрузка в регистр CS слова памяти по адресу ADDRESS+2;

декремент указателя стека на 2;

включение в стек содержимого PC;

загрузка в PC слова памяти по адресу ADDRESS.

В результате этих действий осуществляется межсегментный косвенный вызов подпрограммы обработки прерывания через память, причем адрес памяти однозначно определяется типом прерывания.

Команду прерывания можно использовать для вызова супервизора, т. е. как запрос на обслуживание операционной системой. Для каждого вида обслуживания, которое требуется от операционной системы, в прикладной программе определяется свой собственный тип прерывания. Кроме того, программное прерывание применяется для отладки подпрограмм обслуживания прерываний от периферийных устройств.

Однобайтная команда INT (тип прерывания равен 3) используется в процессе отладки прикладных программ. Вызываемая ею подпрограмма по адресу 0000C обычно является частью пакета отладочных программ.

Команда программного прерывания короче команды CALL межсегментного вызова, и, кроме того, она запоминает в стеке содержимое регистра флажков, что часто требуется для подпрограмм обслуживания прерывания. При этом вызванная подпрограмма обязательно должна заканчиваться командой возврата из прерывания IRET (см. ниже).

Команда INT выполняется за 52 такта синхронизации.

Пример 78.

```
(PC) = 1234          INT 7     ([E89BA]) = 8746
(CS) = 5678                    ([E89B8] = 5678
Флажки = XXXX                  ([E89B6]) = 1234
(SP) = 9ABC                    (PC) = 09A2
(SS) = DEF0                    (CS) = 0484
([0001C]) = 09A2               Флажки = YYYY
([0001E]) = 0484               (SP) = 9AB6
```

Однобайтная команда прерывания при переполнении INTO генерирует программное прерывание в том случае, если в результате операции установлен флажок переполнения (OF=1). Команда INTO выполняется так же, как команда INT, но имеет фиксированный тип прерывания, равный 4. Следовательно, она загружает в PC слова из ячейки с адресом 00010, а в регистр CS — из ячейки с адресом 00012.

Эта команда обычно применяется после команд арифметических операций или после команд сдвига, которые могут вызвать переполнение. Так как переполнение ведет к ошибочным результатам, то его необходимо обрабатывать специальными подпрограммами.

Время выполнения команды INTO составляет 4 (OF=0) либо 52 (OF=1) такта синхронизации.

Однобайтная команда возврата из прерывания IRET предназначена для выхода из подпрограмм обработки прерываний, инициированных аппаратно или программно. По существу действия команды IRET противоположны действиям команды INT:

слово из вершины стека передается в PC;

производится инкремент SP на 2;

слово из вершины стека извлекается в CS;

производится инкремент SP на 2;

слово из вершины стека передается в регистр флажков;

производится инкремент SP на 2.

Время выполнения команды IRET составляет 24 такта синхронизации.

Приводимые ниже программы иллюстрируют некоторые применения команд передачи управления.

1. Двухбайтные команды условных переходов обеспечивают диапазон переходов —128÷+127 байт. Обычно этого диапазона достаточно, но иногда требуется условно перейти к метке, находящейся вне указанного диапазона. Эту проблему можно решить двумя способами. В обоих способах используется расширенный диапазон переходов команды JMP, равный —32 768÷+32 767 байт.

В первом способе, называемом векторным переходом, условный переход осуществляется к метке промежуточной команды JMP, которая и передает управление в нужную точку. Естественно, при этом способе предполагается наличие места для команды JMP в диапазоне условного перехода. Удобно размещать эту промежуточную команду сразу после другой команды JMP или после команды RET. В приводимом фрагменте программы векторный переход используется для перехода по нулю к метке AGAIN, которая находится вне диапазона перехода команды JZ.

Программа 18. Векторный переход

```
AGAIN:      MOV     AX,0
            ADD     BL,DL

            JMP     WHERE
NEXT:       JMP     AGAIN       ; Промежуточная команда
            .
            .
            .
            JZ      NEXT
```

Во втором способе, называемом «переходом через переход», команда JMP находится сразу после команды условного перехода с противоположным требуемому условием.

Программа 19. Переход через переход

```
AGAIN:      MOV     AX,0
            ADD     BL,DL

            JNZ     CONT        ; Противоположное условие
            JMP     AGAIN       ; Длинный переход
CONT:                           ; Условие не удовлетворяется
```

Если условие не удовлетворяется, управление обходит команду JMP, а если оно удовлетворяется, то выполняется команда JMP, которая и передает управление метке AGAIN.

Выбор способа реализации такого длинного условного перехода зависит от конкретной ситуации. Если, например, места для команды JMP в векторном переходе нет, следует применить метод перехода через переход.

2. Во многих языках высокого уровня передача параметров подпрограмм осуществляется через стек. У этого способа имеется несколько достоинств, одно из которых заключается в динамическом использовании области стека для всех подпрограмм. Передачу параметров через стек можно организовать и при программировании на языке ассемблера.

Предположим, что перед вызовом подпрограммы SUBR в стек включены следующие параметры: слово WPAR, байт BPAR и полный адрес APAR (в формате база:смещение) массива данных. Состояние стека после выполнения команды межсегментного вызова показано на рис. 2.38, *а*. Поскольку все стековые операции манипулируют словами, каждая ячейка на рис. 2.38 соответствует

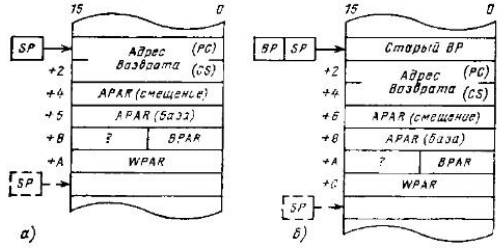


Рис. 2.38. Доступ к параметрам в стеке

слову. Отметим нахождение в вершине стека двух слов, представляющих собой адрес возврата.

Доступ к параметрам в стеке осуществляется одним из двух способов. Первый способ заключается в том, чтобы параметры извлечь из стека в регистры микропроцессора или в ячейки локальных переменных в памяти. Для этого сначала необходимо извлечь из стека и временно запомнить адрес возврата, а после извлечения параметров снова включить его в стек. При этом способе возврата из подпрограм-

мы применяется команда возврата RET без операнда. Описанные выше действия иллюстрирует следующий фрагмент подпрограммы SUBR.

Программа 20. Первый способ доступа к параметрам в стеке

```
SUBR:   POP     SI              ; Запоминание адреса возврата в
        POP     DI              ;       регистрах SI и DI
        POP     TEMP            ; Извлечение параметра в память
        POP     TEMP + 2
        POP     AX              ; Параметр BPAR в регистре AL
        POP     CX              ; Параметр WPAR в регистре CX
        PUSH    DI              ; Восстановление адреса возврата
        PUSH    SI              ;       в стеке
                .
                .
                .
                Команды под-
                программы
                .
                .
                .
        RET                     ; Возврат
```

Второй способ доступа к параметрам заключается в адресации их относительно содержимого регистра BP, при указании которого в формировании адреса по умолчанию участвует регистр SS. Для этого регистр BP должен адресовать вершину стека, а обращения к параметрам выполняются с помощью смещения от BP. Необходимо временно запомнить старое содержимое регистра BP в стеке. Естественно, перед возвратом из подпрограммы необходимо восстановить это старое содержимое регистра BP. После запоминания содержимого BP в стеке в него передается содержимое SP, а для адресации параметров используются смещения от BP. При этом следует помнить, что последний включенный в стек параметр имеет наименьшее смещение и что все смещения являются положительными.

После использования параметров подпрограммой они остаются в стеке. Следовательно, до возврата в вызывающую программу необходимо удалить параметры из стека. Для этого используется специальная команда возврата с непосредственным операндом RET *data*. При возврате значение *data* прибавляется к содержимому указателя стека, что приводит к удалению параметров из стека. Операнд *data* представляет собой число байт, в котором не должны учитываться байты, предназначенные для запоминания содержимого регистра BP и адреса возврата. Содержимое регистра BP извлекается специальной командой POP, а адрес возврата извлекается из стека при выполнении команды RET.

Описанные выше действия иллюстрируются на рис. 2.38, *б* и реализуются фрагментом программы, приведенным ниже.

Программа 21. Второй способ доступа к параметрам в стеке

```
SUBR:   PUSH    BP              ; Запоминание содержимого BP
        MOV     BP, SP          ; BP адресует вершину стека
                .
                .
        MOV     AL, [BP + 10]   ; Параметр BPAR в регистре AL
        LES     DI, [BP + 6]    ; Адрес в регистрах ES : DI
        MOV     CX, [BP + 12]   ; Параметр WPAR в регистре CX
                .
                .
                .
        Команды подпрограммы
                .
                .
                .
        POP     BP              ; Восстановление (BP)
        RET     8               ; Возврат и удаление параметров
```

3. Приведем два простых примера, иллюстрирующих работу команд управления циклами.

Предположим, что необходимо просуммировать элементы массива данных, которые представлены 16-битными беззнаковыми целыми числами. Пусть длина массива определяется переменной LENGTH, а начальный адрес массива находится в регистре DI. Сумма формируется в регистрах AX (младшее слово) и BX (старшее слово).

Программа 22. Суммирование элементов массива

```
        XOR     AX, AX          ; Сбросить регистры
        XOR     BX, BX
        MOV     CX, LENGTH      ; Длина массива в CX
MORE:   ADD     AX, [DI]        ; Прибавить текущий элемент
        JNC     NOC             ; Имеется перенос?
        INC     BX              ; Да, учесть в регистре BX
NOC:    INC     DI              ; Продвинуть указатель
        INC     DI
        LOOP    MORE            ; Повторить цикл сложения
```

В этом фрагменте программы команда LOOP эффективно заменяет последовательность двух команд: DEC CX и JNZ MORE.

Во втором примере предположим наличие двух групп входных портов со смежными адресами. Начальный адрес

первой группы FIRST, а второй — SECOND. Необходимо вводить и сравнивать данные из соответствующих портов обеих групп. Как только считываются неодинаковые данные, ввод прекращается. Число портов в группах определяется переменной PORTS. Ввод осуществляется с косвенной адресацией через регистр DX.

Программа 23. Ввод и сравнение данных

```
        MOV     CX,PORTS        ; Инициализировать счетчик и указа-
        MOV     DX,FIRST        ;   тели
        MOV     BX,SECOND
REPT:   IN      AX,DX           ; Ввести из первой группы
        INC     DX              ; Продвинуть указатель
        XCHG    BX,DX           ; В DX адрес второй группы
        XCHG    AX,BP           ; Временно сохранить в BP
        IN      AX,DX           ; Ввести из второй группы
        INC     DX              ; Продвинуть указатель
        XCHG    BX,DX           ; В DX адрес первой группы
        CMP     AX,BP           ; Сравнить данные
        LOOPE   REPT            ; Зациклить
        JMP     NOMATCH         ; Данные не равны
```

В данном фрагменте команда JMP выполняется после возникновения одного из двух событий:

сравнение привело к образованию нулевого состояния флажка ZF, т. е. введенные из соответствующих портов данные не равны;

цикл повторился заданное число раз, и при вводе из всех соответствующих портов фиксировались одинаковые данные.

2.3.5. Цепочечные команды

Под цепочкой понимается последовательность любых контекстно-связанных байт или слов, находящихся в смежных ячейках памяти. В системе команд МП К1810 имеется пять однобайтных команд, предназначенных для обработки одного элемента цепочек. Цепочечной команде может предшествовать специальный однобайтный префикс повторения REP, который вызывает повторение действия команды над следующим элементом. Благодаря такому префиксу повторения цепочки данных обрабатываются значительно быстрее, чем при организации программного цикла. Повторение рассчитано на максимальную длину цепочки 64К байт и может заканчиваться по нескольким условиям. Кроме того, повторяющуюся операцию можно прерывать и возобновлять. Операции над цепочками выполняются в соответствии с рис. 2.39.

Команды могут иметь операнд-источник, операнд-получатель или то и другое одновременно. Подразумевается, что цепочка-источник по умолчанию находится в текущем сегменте данных, но допускается префикс замены сегмента. Цепочка-получатель должна находиться только в текущем дополнительном сегменте. Ассемблер не использует операнды команд для адресации цепочек. Вместо этого содержимое регистра SI всегда считается смещением текущего элемента цепочки-источника, а содержимое регистра DI — смещением текущего элемента цепочки-получателя. Эти регистры необходимо соответственно инициализировать до выполнения цепочечной команды с помощью команд загрузки адреса LEA, LDS и LES.

При выполнении цепочечной команды содержимое регистров SI и DI автоматически модифицируется, чтобы адре-

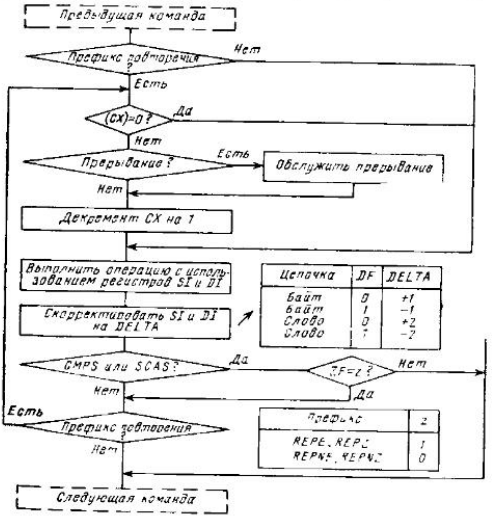


Рис. 2.39. Выполнение цепочечной команды

совать следующие элементы цепочек (см. переменную DELTA на рис. 2.39). Флажок DF направления определяет автоинкремент (DF=0) или автодекремент (DF=1) индексных регистров.

Если команде предшествует префикс повторения, то после каждого ее выполнения производится декремент регистра-счетчика CX, поэтому его необходимо предварительно инициализировать на требуемое число повторений. Когда содержимое CX достигает нуля, управление передается следующей команде.

Префикс (комаида) повторения. Как отмечалось выше, собственно цепочечная комаида оперирует одним элементом цепочки. Повторяющееся (циклическое) выполнение команды обеспечивает однобайтный префикс повторения REP. Наличие этого префикса позволяет, например, передать одной командой цепочку произвольной длины (не превышающей максимальной) из одной области памяти в другую при соответствующей инициализации регистров DS: :SI, ES: DI и CX.

Префикс повторения имеет пять мнемокодов: REP, REPE, REPZ, REPNE и REPNZ. Они определяют только два объектных кода префикса и введены для лучшей передачи содержательного смысла команды. Префикс повторения не влияет на состояния флажков. Кодирование префикса повторения и форматы цепочечных команд приведены на рис. 2.40. Отметим наличие во всех кодах операций бита w (байт или слово).

Префикс REP используется с командами MOVS и STOS и инициирует действие «повторять, пока не достигнут конец цепочки», т. е. до тех пор, пока содержимое регистра CX не достигнет нуля. Префиксы REPE и REPZ действуют аналогично и представляют собой такой же байт, как и

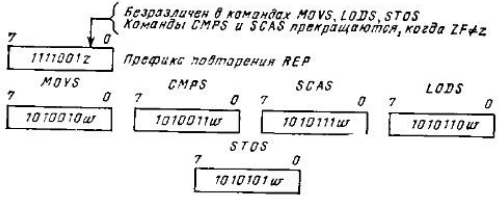


Рис. 2.40. Форматы команд обработки цепочек данных

префикс REP. Они используются с командами CMPS и SCAS и оперируют с флажком ZF=1, состояние которого определяется результатом исполнения этих команд. Префиксы REPNE и REPNZ действуют аналогично предыдущим префиксам, но флажок ZF должен быть равен нулю. В противном случае повторение заканчивается.

При выполнении цепочечных операций МП реагирует на прерывание до обработки следующего элемента цепочки. После возврата из прерывания операция возобновляется с точки прерывания. Однако возобновление операции будет неправильным, если, кроме любого префикса повторения, определены еще один или два префикса (например, замены сегмента или блокировки). Во время обработки прерывания МП помнит действие только одного непосредственно предшествующего команде префикса. Когда осуществлен возврат из прерывания, любые дополнительные префиксы не действуют. Поэтому, если с цепочечной командой должны использоваться несколько префиксов, необходимо запрещать прерывания на время ее выполнения (напомним, что немаскируемые прерывания запретить нельзя). При этом следует учитывать, что временной интервал, в течение которого прерывания запрещены, при обработке длинных цепочек может оказаться неприемлемым.

Команда MOVS. Команда передачи цепочки имеет следующее обобщенное представление:

MOVS *dst,src* *dst*:=(*src*)

Данная комада передает байт или слово из цепочки *src*, адресуемой регистром SI, в цепочку *dst*, адресуемую регистром DI, и соответственно модифицирует указатели SI и DI для адресации следующих элементов цепочек. Состояния флажков не модифицируются. При использовании с префиксом построения REP команда MOVS осуществляет блоковую передачу память — память. Тип цепочки ассемблер определяет по атрибутам операндов. Обычно ассемблер (в зависимости от версии) допускает использование для передачи цепочки мнемокоды MOVSB и MOVSW, которые определяют тип элементов цепочки (B — байт и W — слово). При этом операнды в команде могут отсутствовать.

Например, команда MOVS BUFFER, DATA передает один элемент из области DATA в область BUFFER, а команда REP MOVSB осуществляет блоковую передачу цепочки, элементами которой являются байты.

Время одного выполнения команды без префикса REP равно 18 тактам, а при наличии префикса REP — (9+17 • • (CX)) тактам синхронизации.

Пример 79.

```
(DS) = 1500          MOVSW     (SI) = 000A
(ES) = 27A0                    (DI) = 037C
(SI) = 0008                    ([27D7A]) = ABCD
(DI) = 037A
(DF) = 0
([15008]) = ABCD
```

Команда CMPS. Команда сравнения цепочек имеет следующее обобщенное представление:

CMPS *dst,src* (*src*) — (*dst*)

Эта команда производит вычитание байта или слова цепочки *dst*, адресуемой регистром DI, из байта или слова цепочки *src*, адресуемой регистром SI. В зависимости от результата вычитания устанавливаются флажки, но сами операнды не изменяются. Регистры-указатели продвигаются на следующие элементы цепочек. Если, например, после команды CMPS находится команда JL (перейти, если меньше), то переход осуществляется в том случае, если элемент *src* меньше элемента *dst*.

Когда перед командой CMPS находится префикс REPE (или REPZ), операция интерпретируется как «сравнивать, пока не достигнут конец цепочек или пока элементы цепочек будут не равны». При наличии префикса REPNE (или REPNZ) операция приобретает смысл «сравнивать, пока не достигнут конец цепочек (или пока элементы цепочек будут равны)». Таким образом, команду CMPS удобно применять для нахождения одинаковых или различающихся элементов цепочек.

Для явного указания типа элементов цепочек обычно допускаются мнемокоды CMPSB и CMPSW; при этом операнды в команде отсутствуют.

Время выполнения команды CMPS такое же, как и команды MOVS.

Пример 80.

```
(DS) = 1600          CMPSB     (SI) = 00FF
(ES) = 3780                    (DI) = 0453
(SI) = 0100                    ([16103]) = 87
(DI) = 0454                    ([37C54]) = 87
(DF) = 1                       OF, SF, CF = 0
([16100]) = 87                 ZF, AF, PF = 1
([37C54]) = 87
```


Команда SCAS. Команда сканирования (или просмотра) цепочки имеет обобщенное представление:

SCAS *dst* (*ac*) — (*dst*)

Эта команда вычитает элементы цепочки *dst* (байт или слово), адресуемое регистром DI, из содержимого аккумулятора AL (байт) или AX (слово). В соответствии с полученной разностью устанавливаются флажки, но значения операндов не изменяются. С префиксом REPE (или REPZ) данную команду можно использовать для поиска элемента цепочки со значением, отличающимся от заданного значения. Если перед командой SCAS находится префикс REPNE (или REPNZ), то операция интерпретируется как «просматривать до тех пор, пока не будет достигнут конец цепочки или значение элемента цепочки не будет равно отыскиваемому значению».

Время выполнения команды SCAS без префикса повторения составляет 15 тактов, а с префиксом повторения — (9+15 • (число повторений)) тактов синхронизации.

Для явного указания типа элементов цепочки обычно используются мнемокоды SCASB и SCASW; при этом операнд в команде отсутствует.

Пример 81.

```
(AX) = 3745          SCASW     (DI) = 3ABE
(ES) = A000                    (AX) = 3745
(DI) = 3ABC                    ([A3ABC]) = 743D
(DF) = 0                       OF, ZF, PF, AF = 0
([A3ABC]) = 743D               SF, CF = 1
```

Команда LODS. Команда загрузки цепочки в аккумулятор имеет обобщенное представление:

LODS *src* *ac*:= (*src*)

Когда выполняется эта команда, элемент цепочки (байт или слово), адресуемый регистром SI, загружается в аккумулятор AL или AX, а указатель SI продвигается на следующий элемент цепочки. Состояния флажков не изменяются. Обычно эта команда с префиксом повторения не применяется, но ее удобно использовать в программных циклах вместо команд MOV *ac*, *src* и INC SI (или DEC SI — в зависимости от направления продвижения по цепочке). Время выполнения команды без префикса повторения равно 12 тактам, а при наличии префикса — (9+13• (число повторений)). Допускается использование мнемокодов LODSB и LODSW, указывающих тип элемента цепочки.

Пример 82.

```
(SI) = 8AB0        LODSB   (AL) = AF
(DS) = 3FD4                (SI) = 8AAF
(DF) = 1                   ([487F0]) = AF
([487F0]) = AF
```

Команда STOS. Команда запоминания содержимого аккумулятора в цепочке с обобщенным представлением

STOS *dst* *dst* := (*ac*)

передает байт (слово) из аккумулятора AL (AX) в элемент цепочки, адресуемый регистром DI, и продвигает DI на следующий элемент. Сегментный адрес для этой команды всегда находится в регистре ES, и префикс замены сегмента не используется, а при его наличии — игнорируется. Состояния флажков не изменяются. С префиксом повторения эта команда представляет собой удобное средство инициализации цепочки на фиксированное значение, например нуль или пробел. Время выполнения команды STOS без префикса повторения составляет 11 тактов, а с префиксом повторения (9+10 * (число повторений)) тактов синхронизации. Тип элементов цепочки можно указывать с помощью мнемокодов STOSB и STOSW.

Пример 83.

```
(DI) = 1700        STOSW   (AX) = 3784
(ES) = AD00                (DI) = 16FE
(DF) = 1                   ([AE700]) = 3784
(AX) = 3784
```

Приведем несколько примеров, иллюстрирующих возможности цепочечных команд.

1. Цепочечные команды представляют собой мощное программное средство. В этом нетрудно убедиться, если всю программу I, состоящую из шести команд, заменяют две однобайтные команды:

```
REP     MOVSB
```

Аналогично программа 2 из восьми команд заменяется цепочечной командой с префиксом повторения:

```
REP     MOVSW
```

Такие команды оказываются экономичнее одной команды CALL, если передачу данных из одной области памяти в другую оформить в виде подпрограммы.

2. Пусть необходимо сравнить два массива байт, находящихся в памяти, и при равенстве элементов выйти из цикла. Предположим, что регистр SI адресует текущий элемент первого массива, регистр DI — текущий элемент второго массива, а содержимое регистра CX представляет собой длину массивов. Тогда стандартная программа сравнения принимает следующий вид.

Программа 24. Сравнение массивов байт

```
COMPR:  MOV     AL, [SI]        ; Элемент первого массива
        CMP     [DI], AL        ; Сравнить
        JZ      EQUAL           ; Элементы равны
        INC     SI              ; Продвинуть указатели
        INC     DI
        DEC     CX              ; Проверить окончание
        JNZ     COMPR
```

При распределении регистров задача сравнения массивов решается гораздо более коротким фрагментом:

```
        REPNZ   CMPSB
                JZ      EQUAL
```

3. Как уже отмечалось, команда STOS очень удобна для инициализации областей данных на конкретное значение. Следующая программа сбрасывает массив слов с начальным адресом ARRAY и длиной, определяемой переменной LENGTH.

Программа 25. Сброс массива

```
        LES     DI, ARRAY       ; Адрес в ES : DI
        MOV     CX, LENGTH      ; Длина в CX
        MOV     AX, 0           ; Начальное значение
REP     STOSW                   ; Инициализация
```

2.3.6. Команды управления микропроцессором

Команды данной группы обеспечивают программное управление различными функциями МП. Они делятся на две подгруппы, приведенные в табл. 2.8.

Команды первой подгруппы предназначены для управления состояниями отдельных флажков, а второй — для синхронизации МП с внешними событиями. Последние не влияют на состояния флажков.

Команды CLC, CMC и STC выполняют соответственно сброс, инвертирование и установку в 1 флажка CF. Их удобно применять с командами сдвига через перенос RCR и RCL.

Команды CLD и STD осуществляют сброс и установку флажка направления DF. Состояние этого флажка определяет автодекремент или автоинкремент индексных регистров SI и DI в цепочечных командах.

Таблица 2.8. Команды управления микропроцессором

Мнемоника	КОП	Длина, байт	Время выполнения, тактов	Функция
CLC	F8	1	2	Сбросить флажок CF в 0
CMC	F5	1	2	Инвертировать флажок
STC	F9	1	2	Установить флажок CF в 1
CLD	FC	1	2	Сбросить флажок DF в 0
STD	FD	1	2	Установить флажок DF в 1
CLI	FA	1	2	Сбросить флажок IF в 0
STI	FB	1	2	Установить флажок IF в 1
HLT	F4	1	2 и более	Остановить микропроцессор
WAIT	9B	1	3 и более	Ожидать внешнего события
LOCK	F0	1	2	Выдать сигнал LOCK
ESC	11011X	2—4	8+EA	Переключиться на сопроцессор
NOP	90	1	3	Команда пустой операции

Команды CLI и STI управляют состоянием флажка прерываний IF. После выполнения команды CLI флажок IF сброшен и МП не распознает аппаратные прерывания на входе INT (маскируемые прерывания запрещены). Однако немаскируемые прерывания на входе NMI и программные прерывания МП распознает и соответственно реагирует на них. При подтверждении прерывания флажок IF автоматически сбрасывается.

Команда STI переводит флажок IF в состояние 1, разрешая восприятие прерываний на входе INT. Ожидающее прерывание не распознается до завершения команды, находящейся после команды STI.

Команда HLT останова заставляет МП перейти в состояние останова. Из этого состояния МП может быть выведен или сигналом сброса CLR, или аппаратным прерыванием на входе NMI, или запросом прерывания на входе INT, если эти прерывания разрешены. Данную команду можно использовать вместо бесконечного программного цикла, когда программа должна ожидать внешнего прерывания.

Команда WAIT ожидания переводит МП в состояние ожидания, в котором он периодически через пять тактов синхронизации проверяет сигнал на входной линии $\overline{TEST}$. При появлении на ней активного уровня МП переходит к выполнению следующей за WAIT команды.

Команда LOCK, называемая также префиксом блокировки, заставляет МП (работающий в максимальном режиме) выдать сигнал $\overline{LOCK}$ на время выполнения следующей за префиксом команды. Как показано в § 1.2, этот сигнал блокирует запросы доступа к шине других подсистем. Префикс LOCK может находиться перед любой командой.

Команда ESC представляет собой средство, с помощью которого внешний процессор (сопроцессор) может получать предназначенные для него команды и операнды в процессе работы МП К1810. Сам МП К1810 по этой команде не делает ничего, кроме обращения к памяти за операндом и выдачи его на шину. Сопроцессор следит за шиной и перехватывает команду ESC в момент ее выборки. Затем, контролируя линии состояния очереди QS, он фиксирует начало «выполнения» этой команды. Она содержит постбайт режима адресации, в котором поле *mod*≠11, а поле *reg* произвольно (другими словами, в команде ESC адресуется только память). Микропроцессор К1810 обращается к памяти и помещает считанный операнд на шину, не вводя его в свои регистры, а сопроцессор принимает этот операнд. Отметим, что МП К1810 игнорирует младшие 3 бита кода операции команды ESC. Поэтому для него все команды с кодами операции D8 — DF являются командой ESC.

Пример 84.

(BX) = 375A (DS) = AA50 ((ADC5A)) = F0F0	ESC [BX]	При считывании из памяти на шине будет F0F0

Когда в постбайте режима адресации поле *mod* все-таки содержит 11 (т. е. адресован регистр), команда ESC не производит никаких действий и «выполняется» за два такта синхронизации.

Команда NOP (нет операции) не производит никаких действий. Она может применяться для удаления из программы ненужных байт, а также в программных циклах задержки.

2.4. Программная совместимость микропроцессоров К580 и К1810

В тех редких случаях, когда имеется необходимость переместить разработанное для МП К580 программное обеспечение в МП К1810, возникает проблема программной совместимости этих микропроцессоров.

Как уже отмечалось, прямой программной совместимости вверх между МП К580 и К1810 нет. Однако большинство команд МП К580 возможно однозначно заменить соответствующими командами МП К1810. Но у МП К580 имеются команды, которые для выполнения эквивалентных действий требуют двух, а иногда и более команд МП К1810 (см., например, команды условных переходов, вызовов и возвратов). Некоторые проблемы возникают при попытках эмуляции команд DAD, INX, DCX, XTHL и др. Чтобы показать возможную замену этих команд, примем соответствие регистров обоих микропроцессоров, приведенное в табл. 2.9.

Таблица 2.9. Соответствие регистров МП К580 и К1810

Регистр МП К580	Регистр МП К1810
A	AL
H	BH
L	BL
B	CH
C	CL
D	DH
E	DL
SP	SP
PC	PC (или IP)
PSW	Младший байт регистра флажков и AL

При таком соответствии регистров для любой команды МП К580 нетрудно построить эквивалентную команду или последовательность команд МП К1810.

Команда двойного сложения DAD *rp* прибавляет к регистрам HL содержимое указанной регистровой пары. Тонкость этой команды заключается в том, что она влияет только на флажок переноса C, не изменяя состояний остальных флажков. Поэтому прямая замена ее командой ADD BX, *reg*, которая модифицирует все арифметические флажки, в общем случае невозможна. Для полностью эквивалентной замены приходится применять следующие команды:

```
LAHF            ;Эти пять команд
ADD     BX, reg ;  полностью  эквивалентны
RCR     SI      ;  команде DAD rp
SAHF
RCL     SI
```


Команда LAHF сохраняет в регистре AH предыдущие состояния флажков. Следующая команда ADD осуществляет функцию суммирования. Новое состояние флажка CF команда RCR передает в старший бит регистра SI, который свободен, так как не соответствует никакому регистру МП К580. Затем в младшем байте регистра флажков восстанавливается прежнее состояние флажков (это делает команда SAHF), и заключительная команда RCL передает в него новое состояние флажка CF.

Более просто заменяются команды INX *rp* и DCX *rp*, при выполнении которых флажки не модифицируются:

```
LAHF            ;Эти три команды
INC     reg     ;  эквивалентны команде INX rp
SAHF
LAHF            ;Эти три команды
DEC     reg     ;  эквивалентны команде DCX rp
SAHF
```

Отметим, что если состояния флажков после выполнения команд INX *rp* и DCX *rp* не влияют на вычислительный процесс, то эти команды заменяются командами INC *reg* и DEC *reg* соответственно.

Функцию команды обмена содержимого регистров HL и вершины стека (XTHL) реализуют следующие три команды МП К1810:

```
POP     SI      ;Эти три команды
XCHG    BX, SI  ;  эквивалентны команде XTHL
PUSH    SI
```

Наконец, приведем компактные по числу байт объектного кода эквиваленты еще четырех команд:

```
MOV     DI, CX  ;Эквивалент команды
STOSB           ;  STAX B
MOB     DI, CX  ;Эквивалент команды
LODSB           ;  LDAX B
MOV     DI, DX  ;Эквивалент команды
STOSB           ;  STAX D
MOV     DI, DX  ;Эквивалент команды
LODSB           ;  LDAX D
```

Таким образом, принципиально возможно осуществить покомандное преобразование ассемблерных программ для

МП К580 в ассемблерные программы для МП К1810. Однако при таком преобразовании не используются более широкие возможности системы команд МП К1810, а также команды, свойственные только этому МП, например умножения, деления и др. Для иллюстрации этого приведем пример программы сортировки [2]. Необходимо упорядочить по возрастанию элементы массива, которые интерпретируются как 8-битные беззнаковые целые. Метод пузырьковой сортировки заключается в просмотре массива и сравнении значений соседних элементов. Если значение элемента в ячейке X больше значения элемента в ячейке X+1, производится обмен этих элементов, а в противном случае обмена не происходит. После этого аналогично сравниваются значения элементов в ячейках X+1 и X+2 и т. д. до достижения конца массива, когда просмотр заканчивается.

Факт выполнения хотя бы одного обмена при просмотре массива отмечается записью ненулевого кода в некотором регистре, называемом «индикатором обмена». После окончания просмотра анализируется значение индикатора обмена, и, если он зафиксировал обмен, весь массив просматривается еще раз. Сортировка заканчивается, когда при просмотре массива не было зафиксировано ни одного обмена.

Программа сортировки имеет два параметра: начальный адрес массива в регистре HL, а конечный — в регистре DE. В начале программы в регистре DE образуется длина массива, для чего начальный адрес вычитается из конечного. После этого начальный адрес и длина массива запоминаются в стеке для использования на последующих просмотрах. Затем сравниваются значения двух соседних элементов и при необходимости производится их обмен. Индикатором обмена служит регистр C — в начале просмотра он сбрасывается, а при выполнении обмена в него загружается код FF. После сравнения значений пары элементов выполняется декремент DE и происходит зацикливание до окончания просмотра. Наконец, проверяется индикатор обмена: если регистр C содержит не нуль, то инициируется новый просмотр массива.

При покомандной эмуляции получается показанная ниже программа сортировки для МП К1810, в которой принято соответствие регистров, приведенное в табл. 2.9. Так как программа невелика, в ней использованы короткие условные переходы. Предполагается, что все сегментные регистры находятся в нулевом состоянии.

Программа 26. Сортировка (МП К580)

```
BSORT:  MOV   A,E       ; Образовать в DE
        SUB   L         ;      длину массива
        MOV   E,A
        MOV   A,D
        SBB   H
        MOV   D,A
PASS:   PUSH  H         ; Запомнить начальный адрес и
        PUSH  D         ;      длину
        MVI   C,0       ; Сбросить индикатор
COMP:   MOV   A,M       ; Считать значение XI
        INX   H         ; Указатель на XI+1
        CMP   M         ; Сравнить XI и XI+1
        JC    NEXT      ; Обмен не нужен:
        JZ    NEXT      ;      XI+1≥XI
        MOV   B,M       ; Обменять XI+1 и XI
        MOV   M,A
        DCX   H
        MOV   M,B
        INX   H         ; Указатель на XI+1
        MVI   C,0FFH    ; Установить индикатор
NEXT:   DCX   D         ; Просмотр закончен?
        MOV   A,D
        ORA   E
        JNZ   COMP      ; Нет, продолжить просмотр
        MOV   A,C       ; Да, проверить индикатор
        ORA   A
        POP   D         ; Восстановить начальный ад-
        POP   H         ;      рес и длину
        JNZ   PASS      ; Проверка окончания
```

Программа 27. Сортировка (эмуляция МП К580)

```
BSORT:  MOV   AL,DL     ; Образовать в DX
        SUB   AL,BL     ;      длину массива
        MOV   DL,AL
        MOV   AL,DH
        SBB   AL,BH
        MOV   DH,AL
PASS:   PUSH  BX        ; Запомнить начальный ад-
        PUSH  DX        ;      рес и длину
        MOV   CL,0      ; Сбросить индикатор
COMP:   MOV   AL,[BX]   ; Считать значение XI
        INC   BX        ; Указатель на XI+1
        CMP   AL,[BX]   ; Сравнить XI и XI+1
        JC    NEXT      ; Обмен не нужен
        JZ    NEXT
        MOV   CH,[BX]   ; Обменять XI и XI+1
        MOV   [BX],AL
        DEC   BX
        MOV   [BX],CH
        INC   BX        ; Указатель на XI+1
        MOV   CL,0FFH   ; Установить индикатор
NEXT:   DEC   DX        ; Просмотр закончен?
```

Продолжение

```
        MOV     AL,DH
        OR      AL,DL
        JNZ     COMP            ; Нет, продолжить просмотр
        MOV     AL,CL           ; Да, проверить индикатор
        OR      AL,AL
        POP     DX              ; Восстановить начальный
        POP     BX              ;   адрес и длину
        JNZ     PASS            ; Проверка окончания
```

Если запрограммировать данный алгоритм при тех же начальных условиях, используя все возможности команд МП К1810, программа сортировки получается компактнее.

Программа 28. Сортировка (МП К1810)

```
BSORT:  SUB     DX,BX           ; Образовать в DX длину
PASS:   PUSH    BX              ; Запомнить начальный ад-
        PUSH    DX              ;   рес и длину
        XOR     CL,CL           ; Сбросить индикатор
COMP:   MOV     AL,[BX]         ; Считать значение XI
        CMP     AL,[BX+1]       ; Сравнить XI и XI+1
        JLE     NEXT            ; Обмен не нужен
        XCHG    AL,[BX+1]       ; Произвести обмен значений
        MOV     [BX],AL         ;   XI и XI+1
        INC     BX              ; Указатель на XI+1
        MOV     CL,0FFH         ; Установить индикатор
NEXT:   DEC     DX              ; Просмотр закончен
        JNZ     COMP            ; Нет, продолжить просмотр
        OR      CL,CL           ; Да, проверить индикатор
        POP     DX              ; Восстановить начальный
        POP     BX              ;   адрес и длину
        JNZ     PASS            ; Проверка окончания
```

Таким образом, покомандная эмуляции ассемблерных программ МП К580 в ассемблерные программы для МП К1810 практически всегда оказывается малоэффективной.

ГЛАВА 3

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА

3.1. Пример законченной программы

Приведенный ниже пркмер простой ассемблерной программы, представляющей собой законченный исходный модуль, имеет целью показать общие особенности оформления программ для МП К1810. После программы даются краткие пояснения, а детальное обсуждение конструкций

языка ассемблера составляет содержание данной главы, которую следует рассматривать как продолжение § 2.1 [3, 4].

Программа 29. Законченный исходный модуль

```
NAME    -       EXAMPLE
ASSUME          CS:CODE, DS:DATA,SS:STACK
DATA            SEGMENT
        VAR_1           DW      0       ; Определять и
                                        ;   инициализировать
        VAR_2           DW      0       ;   две переменные
DATA            ENDS
STACK           SEGMENT
                DW      10 DUP (?)      ; Зарезервировать 10
                                        ;   слов
        STK_TOP LABEL WORD              ; Вершина стека
STACK           ENDS
CODE            SEGMENT
        START:          MOV     AX,DATA ; Инициализировать регистр DS
                        MOV     DS,AX
                        MOV     AX,STACK; Инициализировать регистры SS
                        MOV     SS,AX   ;   SP
                        MOV     SP,OFFSET STK_TOP
        PROG:           PUSH    AX      ; Запомнить (AX)
                        MOV     AX,VAR_1; Увеличенное на 5
                        ADD     AX,5    ;   значение VAR_1
                        MOV     VAR_2,AX;   присвоить VAR_2
                        POP     AX      ; Восстановить AX
CODE            ENDS
END     START                           ; Конец исходного модуля
```

В первой строке программы находится директива NAME (наименовать), которая присваивает внутреннее имя объектному модулю, генерируемому ассемблером. Имя

модуля EXAMPLE нельзя путать с именем файла — оно хранится внутри объектного файла.

Отметим один из принципиальных моментов — программа состоит из трех различных логических сегментов DATA, STACK и CODE. Каждый сегмент начинается с директивы SEGMENT и заканчивается директивой ENDS, причем обе директивы для одного и того же сегмента имеют одинаковые имена. Логические сегменты, естественно, соответствуют физическим сегментам в памяти, но привязки логических сегментов к физическим адресам памяти в этом исходном модуле нет.

Директива ASSUME (предположить, считать) во второй строке программы сообщает ассемблеру, что регистр CS будет содержать базовый адрес сегмента CODE, а регистр DS — базовый адрес сегмента DATA. При адресации переменных в рассматриваемой программе регистр ES не используется, поэтому указания о нем в директиве ASSUME отсутствует, а по умолчанию принимается NOTHING (ничего). Смысл информации, которую сообщают ассемблеру директивы SEGMENT/ENDS и ASSUME, наглядно представлен на рис. 3.1.

Сегмент данных DATA содержит всего две переменные (VAR_1 и VAR_2), которые определены и инициализирова-

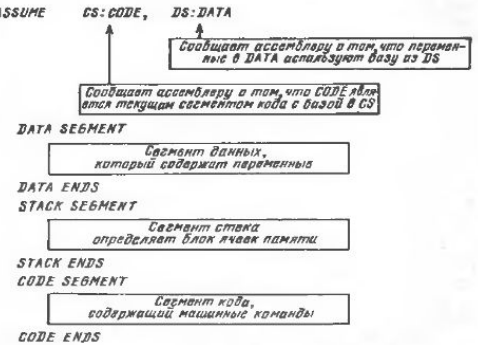


Рис. 3.1. Информация ассемблеру, содержащаяся в директивах SEGMENT/ENDS и ASSUME

ны (установлены в нуль) с помощью директив DW. Обе переменные имеют тип WORD, поэтому в основной области данных рассматриваемой программы имеются два слова.

Первая строка в сегменте STACK (стек) содержит директиву DW с необычным операндом 10 DUP(?). Эта директива резервирует 10 слов памяти, но не инициализирует их. Конструкция 10 DUP (?) означает — «выдать 10 копий значения, находящегося в круглых скобках». Вопросительный знак указывает, что начальные значения резервируемых слов произвольны и никакие предположения о них недопустимы. Таким образом, в программе для стека резервируется 10 слов, или, как говорят, стек имеет глубину 10 слов.

Следующая строка в сегменте STACK содержит директиву LABEL (отметить):

```
STK_TOP   LABEL   WORD
```

Данная директива определяет имя STK_TOP, которое относится к слову, находящемуся после 10 слов, т. е. имя STK_TOP идентифицирует вершину пока пустого стека. Значение смещения STK_TOP от начала сегмента STACK должно находиться в указателе стека SP, когда стек пустой. Содержимое SP будет уменьшаться на 2 при выполнении каждой команды, включающей слово в стек. Когда стек полностью заполнен (в нем находится 10 слов), содержимое SP равно нулю. Включение еще одного слова в заполненный стек приведет к неуправляемому поведению системы.

Сегмент кода начинается с пяти команд пересылки данных MOV. Они осуществляют обязательную инициализацию сегментных регистров DS, SS и указателя стека SP. Как отмечалось в § 1.5, сегментные регистры и SP необходимо инициализировать до выполнения любой команды, осуществляющей обращение к памяти. Имена DATA и STACK идентифицируют базовые адреса сегментов данных и стека. Выражение OFFSET STK_TOP представляет собой значение смещения метки STK_TOP от начала содержащего ее сегмента STACK.

Собственно программа начинается с метки PROG. Программа преследует только иллюстративные цели и выполняет очень простые действия. Сначала содержимое аккумулятора AX включается в стек. Затем в AX загружается значение переменной VAR_1. Оно увеличивается на 5, и ре-

зультат запоминается как значение переменной VAR_2. После этих действий из стека в аккумулятор извлекается его старое содержимое. Нетрудно заметить, что действия данной программы описываются простым оператором присваивания:

$$VAR\_2 := VAR\_1 + 5$$

Последняя строка программы с директивой END, имеющей операнд START, сообщает ассемблеру о достижении конца исходного модуля и необходимости начать выполнение программы с команды, отмеченной меткой START.

Заметим, что любой исходный модуль состоит из логических сегментов, которые представляют собой блоки машинных команд или инициализаций данных, ограниченные директивами SEGMENT и ENDS. Директива ASSUME сообщает ассемблеру, что он должен знать о содержимом сегментных регистров для того, чтобы генерировать команды с обращением к памяти. Сама эта директива не формирует машинных команд, поэтому ответственность за правильную загрузку сегментных регистров несет программист.

3.2. Переменные

При программировании нензбежны ситуации, когда к определенным местам в программе и данным приходится обращаться несколько раз. Вместо того, чтобы оперировать громоздкими численными значениями адресов, удобно определять и применять символические имена, соответствующие адресам указанных элементов. В § 2.1 было показано, что важную роль при программировании играют метки, предназначенные для передачи управления отмеченным командам, и переменные, с помощью которых осуществляются обращения к данным.

Метки и переменные, заменяя абсолютные адреса памяти, различаются несколькими характеристиками. Например, метки определяются записью двоеточия после имени, и любая метка должна иметь атрибут расстояния NEAP и FAR. Кроме того, за исключением директивы LABEL (см. § 3.6), метки соотносятся с сегментным регистром CS. Переменные имеют атрибут типа, но никогда не имеют атрибута расстояния. Их нельзя определять с помощью двоеточия. Переменные всегда обозначают данные безотносительно к сегментным регистрам, но наиболее часто соотносятся с сегментным регистром DS.

В данном параграфе рассматриваются вопросы, относящиеся к определению и использованию переменных как наименованных единиц данных. Поскольку любая программа предназначена для преобразования входных данных в необходимые пользователю выходные результаты, то вопросы определения данных, их типизации и размещения в памяти, а также режимы адресации как средства обращения к нужным данным очень важны для успешного программирования.

Директивы DB, DW и DD. Микропроцессор К1810 оперирует с тремя основными типами данных: байт (BYTE — 8 бит), слово (WORD — 16 бит или 2 байта) и двойное слово (DWORD — 32 бита или 2 слова, или 4 байта). Для определения и, возможно, инициализации байт, слов и двойных слов предназначены операторы распределения данных, которые далее для краткости называются директивами.

Директивы DB (определить байт), DW (определить слово) и DD (определить двойное слово) имеют следующий общий формат, или синтаксис:

$$\langle \text{имя переменной} \rangle \quad \left\{ \begin{matrix} \text{DB} \\ \text{DW} \\ \text{DD} \end{matrix} \right\} \langle \text{нач. значение} \rangle, [\langle \text{нач. значение} \rangle] \ldots$$

Конструкция [ ]..., которая используется и в дальнейшем, обозначает, что находящиеся в квадратных скобках аргументы или поля могут не появляться ни одного раза или появиться произвольное число раз. Следовательно, запись <нач. значение>, [<нач. значение>]... показывает, что в поле операнда всех директив требуется минимум одно начальное значение, но при необходимости допустим и список начальных значений, элементы которого разделяются запятыми. Слова, набранные курсивом, обозначают резервирование места для конкретных имен и (или) чисел, которые использует программист. Отметим, что мнемокоды директив по существу определяют тип переменных.

Поле операнда рассматриваемых директив идентифицирует, сколько байт, слов или двойных слов распределяют директивы и какими должны быть их начальные значения. Атрибутом сегмента переменной является тот сегмент, в котором она определена, а атрибут смещения переменной равен числу байт от начала сегмента до ячейки с переменной. Простой пример определения переменных и пояснения их атрибутов был приведен в § 2.1.

В качестве начального значения может фигурировать выражение, содержащее значение инициализации для од-

ной единицы памяти, т. е. байта, слова или двойного слова. Выражения как достаточно сложные конструкции языка ассемблера подробно рассматриваются в § 3.5. Пока достаточно сказать, что имеются два вида выражений: числовые и адресные. Например, 10 или 3* 20 — это числовое выражение, а переменная или метка — адресное. Адресные выражения, используемые для инициализации памяти, допустимы в поле операнда только для директив DW и DD. Приведем более сложный пример определения переменных:

```
MYSEG   SEGMENT      AT   65H   ; Номер параграфа 0065
  ZERO  DB           0FFH       ; Один байт, равный FF
  ONE   DW           1234 H     ; Одно слово, равное 1234
  TWO   DD           TWO        ; Младшее слово 0003,
                                ;   старшее слово 0065
  FOUR  DW           FOUR +5    ; Одно слово, равное 000C
  SON   DW           ZERO — TWO ; Одно слово, равное
                                ;   FFFD(=—3)
  ATE   DB           5 * 6      ; Один байт, равный 1E
  WIFE  DW           ?          ; Слово без инициализа-
                                ;   ции
MYSEG   ENDS
```

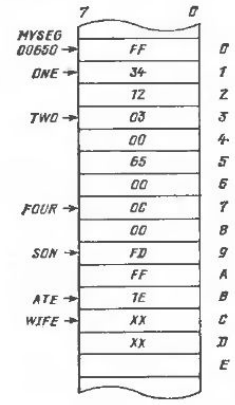


Рис. 3.2. Инициализация памяти

Содержимое памяти, инициализируемой данным фрагментом, показано на рис. 3.2. Первые две директивы особенного интереса не представляют. Директива DD с именем TWO распределяет два слова: первое слово (младшее), равное 0003, — это смещение переменной TWO относительно начала сегмента, а второе слово равно номеру параграфа 0065, с которого начинается сегмент MYSEG. Таким образом, два слова в директиве DD представляют собой адрес переменной TWO в формате сегмент:смещение. Значением переменной FOUR становится сумма ее смещения от начала сегмента (оно равно 7) и числа 5. Переменная SON инициализируется как число —3, представ-

ляемое в дополнительном коде, а значением переменной ATE является 30 (или 1E).

При анализе рис. 3.2 следует помнить соглашение о хранении в памяти слов и двойных слов, принятое в МП К1810: младший байт любого слова хранится по меньшему адресу и младшее слово двойного слова также хранится по меньшему адресу. Поэтому значение переменной TWO, равное в позиционной записи 0065 0003, размещается в четырех байтах как 03 00 65 00 в порядке возрастания адресов. Такая перестановка байт обычно важна в расшифровке дампов памяти при отладке программ. В подавляющем большинстве случаев МП автоматически учитывает соглашение о размещении в памяти слов и двойных слов. Например, в командах загрузки длинного адреса LDS и LES первое слово интерпретируется как смещение, а второе — как номер сегмента. Однако в ситуациях, когда приходится оперировать с байтами, принадлежащими слову, необходимо помнить о том, как слово хранится в памяти. Рассмотрим в качестве примера фрагмент, приведенный ниже.

```
FORM_B  LABEL   BYTE
FORM    DW      1234H
        .
        .
        .
        MOV     CL,FORM_B
        MOV     AL,FORM_B+1
```

Директива LABEL присваивает имя FORM_B следующей за ней ячейке памяти с указанием типа BYTE. Поэтому команда MOV CL,FORM_B загружает в регистр CL значение 34, а команда MOL AL,FORM_B+1 передает в регистр AL значение 12.

На первый взгляд кажется, что операнд директивы DD может быть 8-разрядным 16-ричным числом. Однако ассемблер допускает только 16-битные числа (слова), что согласуется с архитектурой микропроцессора. Поэтому в директиве

```
        DD      1234H
```

предполагается, что старшее слово будет равно 0000. Следовательно, с учетом соглашения о размещении в памяти слов и двойных слов эта директива эквивалентна следующей паре директив DW:

```
        DW      1234H
        DW      0000H
```

В некоторых версиях ассемблера с помощью директивы DD допускается определять длинный указатель в формате база:смещение. Например, директива

```
SEG_1    DD    1234H:5678H
```

поместит в смежные 4 байта памяти коды 78 56 34 12 (в порядке возрастания адресов). Такой указатель можно загрузить в регистре ES : BX с помощью команды LES:

```
LES  BX,SEG_1
```

Вопросительный знак. Одиночный вопросительный знак в языке ассемблера является ключевым словом. Его использование допускается только при распределении памяти и означает, что фактическое содержимое ячеек памяти непредсказуемо. Это отмечено на рис. 3.2 символами XX, представляющими собой неопределенное значение инициализации переменной WIFE. Такой же цели в языках ассемблера других МП служит специальная директива резервирования памяти DS.

Конструкция DUP. Особой разновидностью начального значения, находящегося в поле операнда директив DB, DW и DD, является конструкция DUP, которая применяется для распределения и инициализации нескольких единиц памяти. Общий формат конструкции DUP имеет следующий вид:

n DUP (<нач. значение>,[<нач. значение>] ...)

Здесь *n*, интерпретируемое как беззнаковое целое, показывает, сколько раз повторяются элементы, находящиеся в круглых скобках. В качестве элементов могут фигурировать численные и адресные выражения, вопросительный знак, список или повторная конструкция DUP. Возможности конструкции DUP иллюстрируют следующие примеры:

```
        DB  100 DUP (0)          ; Сто нулевых байт
        DW  10  DUP (?)          ; Десять слов без значений
ADDR    DD  20  DUP (ADDR)       ; 20 полных адресов
        DB  10  DUP (10 DUP(0))  ; Сто нулевых байт
        DW  5   DUP(ADDR, 0, 1)  ; Пять повторений трех слов
```

Таким образом, с помощью небольшой конструкции можно инициализировать большие блоки памяти. Еще один пример с глубиной вложения, равной трем:

```
GAMMA DW 4 DUP (2 DUP (1,2 DUP (3, 4), 5), 6)
```

Эта директива инициализирует 52 слова памяти, создавая четыре копии следующих значений:

0001, 0003, 0004, 0003, 0004, 0005, 0001, 0003, 0004, 0003, 0004, 0005, 0006.

Максимальная глубина вложения конструкций DUP обычно не превышает восьми.

Символьные цепочки. В языке ассемблера допускается использовать символьные цепочки, которые образуются при заключении последовательности символов в апострофы. Каждый символ цепочки занимает в памяти 1 байт, и адресом цепочки считается адрес первого символа. Цепочки, содержащие больше двух символов и предназначенные для инициализации памяти, разрешено применять только в директивах DB. Примеры задания символьных цепочек приведены ниже.

```
MODE_1   DB   'HELLO'
MODE_2   DB   'CONGRATULATE YOU'
MODE_3   DB   'PHASING ERROR'
BLOCK    DB   128 DUP (' '); 128 пробелов
SCREEN   DB   24 DUP (80 DUP (' '), 0DH, 0AH)
```

Последняя директива DB определяет 24 строки, каждая из которых содержит 80 пробелов и заканчивается символами возврата каретки и перевода строки.

В директивах DW и DD допускаются символьные цепочки, состоящие из одного или двух символов. Цепочка из двух символов интерпретируется как двухбайтное число, поэтому необходимо помнить о соглашении, касающемся хранения в памяти слов и двойных слов. Примеры цепочек как операндов директив DW и DD приведены ниже.

```
EXAM_1   DW   'KP'   ; Запоминает 4B50
EXAM_2   DD   'G'    ; Запоминает 0000 0047
```

Операторы LENGTH, SIZE и TYPE. Эти встроенные операторы предназначены для использования в выражениях, где запрашивается тип информации. Они особенно удобны для разработки общих процедур, реализующих аналогичные процессы независимо от того, какие параметры им передаются. Кроме того, использование имен и выражений вместо численных значений преследует и такие практические цели, как лучшая понимаемость листингов и более простая модификация программ. Все рассматриваемые операторы требуют указания после себя имени переменной.

Оператор TYPE (тип) сообщает, сколько байт отведено для переменной. Тип переменной, определенной директивой DB, равен единице, директивой DW — двум и директивой DD — четырем.

Оператор LENGTH (длина) заставляет ассемблер возвратить число основных единиц памяти (байт, слов, двойных слов), распределенных в строке с определенной переменной.

Наконец, оператор SIZE (размер) сообщает, сколько байт памяти распределено при определении переменной.

Исходя из функций, реализуемых каждым из рассмотренных операторов, нетрудно вывести следующее соотношение:

SIZE <*имя*> = LENGTH <*имя*> * TYPE <*имя*>

Применение операторов TYPE, LENGTH и SIZE иллюстрирует следующий фрагмент, формирующий в аккумуляторе AX сумму значений элементов массива ARRAY:

```
ARRAY   DW      50 DUP (?)
        SUB     AX,AX           ; Сбросить аккумулятор AX
        MOV     CX,LENGTH ARRAY ; Число элементов в CX
        MOV     SI,SIZE ARRAY   ; Индекс конца массива
SUMMA:  SUB     SI,TYPE ARRAY   ; Возврат на элемент
        ADD     AX,ARRAY [SI]   ; Прибавить элемент
        LOOP    SUMMA           ; Выполнять до (CX)=0
```

В соответствии с директивой DW элементы массива имеют тип WORD, длина массива равна 50 словам, а размер 100 байтам. Нетрудно убедиться в том, что при изменении типа элемеитов массива, например при определеиии его директивой DB, или при изменеиии числа элементов массива ии одну из машинных команд модифицировать не иужно.

Директива RECORD. В языке ассемблера предусмотрено средство символического определения отдельных бит и битовых цепочек внутри байта или слова. Такое определение называется записью (*record*), а каждая наименованная битовая цепочка в записи (которая может состоять и из одного бита) называется полем. Применение записей обеспечивает эффективное использование памяти, одновременно улучшая читабельность программ и уменьшая вероятность ошибок.

Директива RECORD, определяющая запись, имеет следующий формат:

<*имя*> RECORD <*имя поля:длина*> , [<*имя поля:длина*>] . . .

Каждое поле определяется путем указания его имени, после которого обязательно находятся двоеточие и длина поля. Максимальное число бит, указываемое в директиве RECORD, равное 16 (слово), а минимальное равно 1. Оператор WIDTH (ширина) применительно к записи сообщает ее ширину в битах, т. е. сумму длин всех составляющих ее полей. Размер записи равен числу байт, необходимых для ее размещения, и может быть равен 1 (если ширина записи находится в пределах 1—8 бит) или 2 (если ширина записи составляет 9—16 бит).

Важно подчеркнуть, что определение записи не распределяет памяти. Оно представляет собой шаблон, который сообщает ассемблеру имена и размещение всех битовых полей внутри байта или слова. Когда позднее имя поля фигурирует в команде, ассемблер использует определение записи для формирования непосредственного операнда (маски — MASK) в таких командах, как TEST, AND, XOR и др., или значения счетчика в командах сдвигов.

Приведем пример определения записи:

EMPLOYEE RECORD DATE:3, AGE:6, PAY:7

В соответствии с этим определением ассемблер генерирует для имени EMPLOYEE следующие значения:

DATE = 13, маска DATE = D000;

AGE = 7, маска AGE = 1F80;

PAY = 0, маска PAY = 007F.

Значения слева представляют собой числа бит, которые требуются для правого выравнивания соответствующих полей, а значения справа — это маски, необходимые для выделения или проверки полей в логических командах. Ширина записи EMPLOYEE равна 16, а размер составляет 2 байта.

Приведем более подробный пример использования записи.

```
WORKER  DB      ?               ; Байт без инициализации
        .
        .
        .
BITREC  RECORD                  ; Определение записи
&       YEARS:6,                ; Возраст
&       EDUC:1,                 ; Образование (1=высшее)
&       STATUS:1                ; Состояние (1=работает)

; Необходимо выбрать людей
; с высшим образованием, находящихся на пенсии
```

```
MOV     AL,WORKER          ; Исходный байт
TEST    AL,MASK EDUC       ; Образование?
JZ      FAULT              ; Не высшее
TEST    AL,MACK STATUS     ; Работает?
JNZ     FAULT              ; Да
; Дальнейшая обработка
FAULT:                     ; Не выбран
```

Директива STRUCTURE. Директива STRUCTURE (или короче — STRUC) обеспечивает в языке ассемблера удобное средство определения и доступа к переменным со сложным типом данных. Структура, ограниченная директивами STRUC и ENDS с одинаковыми именами, представляет собой шаблон (называемый также картой), который присваивает имена и атрибуты (тип, длина и размер) набору полей. В этом отношении директива STRUC напоминает директиву RECORD, но поля структуры в отличие от полей записи образованы из основных единиц данных — байт, слов и двойных слов. Поэтому каждое поле в структуре определяется с помощью директив DB, DW или DD. Затем поля такого шаблона используются для определения переменных с новым типом данных, который и порождает директива STRUC. Обращение к различным полям в таких переменных осуществляется с использованием специального оператора точки (.).

Как и при записи, память структуре не распределяется. Вместо этого структура ассоциируется (связывается) с конкретной областью памяти, когда имя поля фигурирует в команде вместе с базовым адресом. Базовым адресом, который локализует структуру, может быть имя переменной или один из базовых регистров BX или BP. Определяя различные базовые адреса, можно ассоциировать структуру с различными областями памяти.

Покажем определение структуры и переменной с новым типом данных на следующем простом примере.

```
; Определение шаблона структуры
RADIANS     STRUS
    PHI     DW?                ; Определяет новый
    CSI     DW?                ;    тип данных,
    TETA    DW?                ;    состоящий из
RADIANS     ENDS               ;    трех слов
; Определение переменной с использованием шаблона
AIRPL       RADIANS  <8, 8, 12>  ; Начальные значе-
                                 ;    ния
; Доступ к полю внутри структуры
            MOV AX, AIRPL.CSI    ; Эта команда
; загружает в AX поле CSI переменной AIRPL
```


Приведем еще один пример определения и использования структуры, состоящей из 15 байт, которые разделены на четыре поля.

```
WORKER      STRUC                          ; Определение
    YEARS   DB      6 DUP (?)              ;    структуры
    PAY     DB      4 DUP (?)
    LAB     DB      2 DUP (?)
    ING     DB      3 DUR (?)
WORKER      ENDS
; Определение переменных
    JOHN    DB      15 DUP (?)  ; Определение пе-
    POUL    DB      15 DUP (?)  ;    ременных
; Изменить значение LAB в переменной JOHN
; на значение этого поля из POUL
            MOV     AX,POUL. LAB
            MOV     JOHN.LAB, AX
```

Замена атрибутов переменных. Как указывалось выше, любая переменная в программе имеет три атрибута: тип, сегмент и смещение в сегменте. Язык ассемблера позволяет программисту временно изменить тип или сегмент, ассоциированный с переменными, с помощью специальных операторов замены атрибутов.

Предположим, что в разрабатываемой программе необходимо иметь несколько областей данных, адресуемых с различными значениями базы. Пусть для определенности имеются три области данных, соответствующих логическим сегментам с именами REGION_0, REGION_1, REGION_2. Если известно, что, например, сегмент REGION_0 содержит наиболее часто используемые переменные, то в качестве базового регистра для него целесообразно назначить сегментный регистр DS. В процессе инициализации в регистр DS будет загружено значение REGION_0, а при выполнении программы он будет участвовать в формировании физического адреса по умолчанию. Для этого потребуется специальная директива ASSUME, которая рассматривается в § 3.3.

Обращения к данным из других областей осуществляются сравнительно редко, поэтому для их адресации можно воспользоваться сегментным регистром ES. Если в нем находится база сегмента REGION_1, то при необходимости обращения через него в область REGION_2 придется загружать в регистр ES новое значение базы.

Как ассемблеру сообщить о том, что при обращении к данным из сегмента REGION_1 нужно брать базу из сег-

ментного регистра ES? Напомним, что МП при любом обращении к данным по умолчанию предполагает использование регистра DS. Имеются два способа передать ассемблеру необходимую информацию. Один из них основывается на возможности директивы ASSUME и эффективен при интенсивных обращениях к данным, которые адресуются через сегментный регистр ES (см. ниже).

Второй способ более удобен, когда через регистр ES требуются лишь эпизодические обращения к данным. Способ основывается на применении оператора замены сегмента. На языке ассемблера этот оператор представляет собой мнемокод сегментного регистра с последующим двоеточием, которые находятся перед именами переменной. Оператор замены сегмента явно сообщает ассемблеру, какой сегментный регистр используется при обращении к переменной только в данной команде. Если, например, переменная BVAR типа BYTE находится в сегменте REGION_1, а база этого сегмента хранится в сегментном регистре ES, то команда

```
XCHG  AL,ES:BVAR
```

произведет обмен значения переменной BVAR и содержимого регистра AL.

Оператор замены сегмента допустим и в так называемых анонимных обращениях, т. е. в адресах памяти, не содержащих имен переменных. Например, команда MOV CX, [SI] передает в регистр CX слово из памяти, адрес которого формируется через регистры DS и SI. Если необходимо передать в CX слово памяти, база которого находится в регистре ES, а смещение в регистре SI, необходимо воспользоваться оператором замены сегмента MOV CX,ES : [SI].

Разумеется, оператор замены сегмента тесно связан с префиксом замены сегмента, который ассемблер помещает (при необходимости) перед командой. Однако наличие в команде оператора не гарантирует образования префикса замены сегмента. Оператор замены сегмента просто сообщает ассемблеру, какой сегментный регистр должен использоваться при обращении к памяти в данной комаиде. Но ассемблер может определить, что префикс замены сегмента в конкретной команде не требуется. Пусть, например, в программе имеется команда ADD CX, SS : [BP]. Оператор замены сегмента сообщает, что при образовании адреса памяти в этой команде должен участвовать сегментный регистр SS. Однако МП любые обращения к памяти через регистр BP по умолчанию считает относящимися к сегменту стека. Следовательно, префикс замены сегмента для данной команды не формируется. Генерируемая ассемблером машинная команда будет иметь префикс замены сегмента, если он необходим для отмены того сегментного регистра, который используется в МП по умолчанию.

Рассмотрим вопрос о замене типа переменной. Напомним, что тип переменной — это число байт, которое предполагается при обращении к переменной по имени. Знание типа необходимо для того, чтобы ассемблер мог сформировать правильную машинную команду с обращением к переменной. Если, например, переменная WVAL имеет тип WORD, то команда MOV WVAL,7 требует загрузки в ячейку WVAL 16-битной константы со значением 7. Тип переменной позволяет также обнаружить некоторые ошибки при обращении к переменным. Например, команда ADD AL,WVAL вызовет сообщение об ошибке, так как регистр AL имеет тип BYTE, а переменная WVAL — тип WORD.

Когда требуется, чтобы команда MOV WVAL,7 или ADD AL,WVAL оперировала байтом, необходимо как-то указать, что в них производится обращение к первому байту слова. Для этого применяется оператор замены типа, имеющий следующий общий формат:

$$\left\{\begin{array}{l}\text{BYTE}\\ \text{WORD}\\ \text{DWORD}\end{array}\right\}\ \text{PTR}\ \langle\textit{имя переменной}\rangle$$

Конструкции $\langle$*тип*$\rangle$ PTR $\langle$*имя переменной*$\rangle$ означает, что в команде следует использовать сегмент и смещение переменной с именем *имя переменной*, но с явно заданным типом. Например, рассмотренные выше две команды следует записать в виде

```
MOV     BYTE PTR WVAL,7     ; Инициализировать байт
ADD     AL,BYTE PTR WVAL    ; Прибавить байт
```

Оператор замены типа также требуется для устранения неоднозначности, которая иногда возникает в анонимных обращениях. Например, из команд

```
SUB     [BX],5      ; Вычитается байт или слово?
INC     [BP]        ; Инкремент байта или слова?
```

явно не видно, какой тип имеет операнд в памяти. Следовательно, ассемблер не может определить, какую команду необходимо сформировать — оперирующую байтом или сло-

вом. В приведенном виде эти команды вызовут сообщение об ошибке. Если первая команда должна оперировать словом, а вторая байтом, следует явно определить тип операнда:

```
SUB     WORD PTR [BX],5     ; Вычитание слов
INC     BYTE PTR [BP]       ; Инкремент байта
```

Отметим, что не все анонимные обращения приводят к неоднозначности. В двухоперандных командах информацию о типе может содержать любой из операндов. Например, команда MOV [BX], AX оперирует словом, так как регистр AX имеет тип WORD. В этом случае тот факт, что BX не содержит информации о типе, не играет роли.

Примечание. Современные микропроцессоры оперируют данными нескольких типов. Сейчас *de facto* стали стандартными типы байт, слово, двойное (длинное) слово и иногда счетверенное слово, содержащее 64 бита. В то же время ради упрощения системы команд для пользователя число мнемокодов стараются уменьшить, т. е. сделать их универсальными, по крайней мере, не зависящими от режимов адресации и типов операндов. Поэтому ситуация, в которой по общему мнемокоду невозможно определить тип операнда, становится для микропроцессоров довольно распространенной. Один из способов устранения неоднозначности наряду с принятым в языке ассемблера МП К1810 заключается в том, чтобы добавить к мнемокоду команды символ, идентифицирующий тип операнда, например:

```
MOVE.B    или MOVEB    — передать байт;
MOVE.W    или MOVEW    — передать слово;
MOVE.L    или MOVEL    — передать длинное (long)
                         слово.
```

3.3. Директивы управления сегментами

В языке ассемблера МП К1810 вводится понятие *логического сегмента*, которое просто означает «часть программы». Логические сегменты отражают разработку программы в виде отличающихся друг от друга областей кода (собственно программы), данных и стека. Простые программы могут состоять всего из трех логических сегментов: один для машинного кода, один для переменных (данных) и один для стека.

Логические сегменты как средство языка ассемблера взаимосвязаны с физическими сегментами, представляющими собой особенности архитектуры микропроцессора. Каждый логический сегмент в ассемблериой программе определяет именованную область памяти, которая адресуется с неизменным содержимым одного из сегментных регистров. Следовательно, логический сегмент — это спецификация программы, определяющая содержимое сегментного регистра.

Директивы SEGMENT/ENDS. Исходный ассемблерный модуль обычно состоит из нескольких логических сегментов. Каждый из них должен начинаться с директивы SEGMENT (сегмент) и заканчиваться директивой ENDS (конец сегмента):

```
<имя>   SEGMENT     [<список атрибутов>]
          .
          .
          .
<имя>   ENDS
```

Каждому логическому сегменту присваивается *имя*, определяемое программистом. После директивы SEGMENT все команды или данные, имеющиеся до директивы ENDS, считаются находящимися в одном сегменте. Операнд директивы SEGMENT содержит список атрибутов, который является необязательным, но необходимым в тех случаях, когда программа состоит из нескольких исходных модулей. Модульное программирование и смысл атрибутов сегментов рассматриваются в § 3.7.

Приведем пример определения простого логического сегмента, содержащего четыре переменные:

```
DATA    SEGMENT
            FUNC_1    DB    ?
            FUNC_2    DW    ?
            ADDR      DD    ?
            POLY      DW    ?
DATA    ENDS
```

До использования сегментного регистра в формировании физических адресов памяти он должен быть инициализирован. База, соответствующая логическому сегменту, представлена именем сегмента. Следовательно, при инициализации сегментного регистра DS, определяющего основную область данных, используется имя основного сегмента данных. Если, например, показанный выше сегмент DATA является в программе основной областью данных, то до любых обращений к переменным необходимо инициализировать сегментный регистр DS:

```
MOV     AX,DATA     ; Инициализация регистра DS
MOV     DS,AX       ;   через регистр AX
```

Регистр SS содержит базу стека, а указатель стека SP адресует вершину стека. До использования стека необходимо инициализировать оба эти регистра. В регистр SS загружается база сегмента стека, а указатель стека SP должен инициализироваться так, чтобы первая команда включения в стек (PUSH) устанавливала SP на слово с максимальным смещением в области стека. Напомним, что в МП К1810, как и в большинстве других процессоров, стек растет вверх, в область меньших адресов. Следовательно, при инициализации в SP следует загрузить смещение первого слова ниже области стека. Инициализация SP обычно осуществляется с привлечением директивы LABEL и оператора OFFSET. Пример инициализации стека и некоторые пояснения были приведены в § 3.1. Обычно в любой программе команды инициализации сегментных регистров DS, SS и SP являются первыми выполняемыми командами, так как обращения к данным и стековые операции предполагают правильную установку этих регистров.

Язык ассемблера МП К1810 допускает так называемые вложенные сегменты. Поясним это понятие следующим примером. Предположим, что парой директив определен некоторый сегмент с именем SEG_0. Если имя SEG_0 появляется позже в новой директиве SEGMENT, то содержащиеся в новом сегменте данные и (или) команды будут автоматически размещаться за операторами, которые находятся в старом сегменте с именем SEG_0. Приведем пример такого построения:

```
SEG_0           SEGMENT
                .                ; Это находится в сегмен-
                .                ;    те SEG_0
                .
        CONT:
SEG_0           ENDS

        SEG_1   SEGMENT
                .                ; Это  содержится в сег-
                .                ;    менте SEG_1
                .
        SEG_1   ENDS

SEG_0           SEGMENT          ; Это находится в SEG_0
                .                ;    после строки с мет-
                .                ;    кой CONT
SEG_0           ENDS
```


Еще один допустимый способ вложения сегментов с аналогичным результатом имеет следующий вид:

```
SEG_0           SEGMENT
                .                ; Это находится в сегмен-
                .                ;    те SEG_0
                .
CONT:
SEG_1           SEGMENT
                .                ; Это  содержится в сег-
                .                    менте SEG_ I
                .
SEG_1           ENDS
                .                ; Это находится в SEG_0
                .                ;    после  строки с мет-
                .                ;    кой CONT
SEG_0           ENDS
```

Вложенный сегмент должен заканчиваться раньше внешнего сегмента. Поэтому ассемблер зафиксирует ошибку, если директива SEG_1 ENDS находится после директивы SEG_0 ENDS.

Директива ASSUME. Чтобы правильно генерировать машинные команды с обращениями к памяти, ассемблеру нужна информация о базовых значениях, загруженных в сегментные регистры. Зная эту информацию, ассемблер определяет, какой из сегментных регистров можно использовать для адресации конкретной ячейки памяти и требуется ли префикс замены сегмента. Если для формирования адреса операнда нельзя использовать ни один из сегментных регистров, ассемблер формирует сообщение об ошибке.

Необходимая ассемблеру информация о содержимом сегментных регистров сообщается в директиве ASSUME (предположить), имеющей следующий формат:

ASSUME *<SR: базовое значение>*, [*<SR: базовое значение>*]...

В этом определении сокращение SR обозначает сегментный регистр.

Поле SR содержит имя одного из сегментных регистров (CS, DS, SS, ES), а *базовое значение* указывает начало области памяти, адресуемой через сегментный регистр. Одним из наиболее часто используемых типов *базового значения* является имя сегмента, например ASSUME DS : DATA. Такая директива сообщает ассемблеру, что к определенным в сегменте DATA переменным можно обратиться через сегментный регистр DS. Кроме того, из этой директивы следует, что регистр DS нельзя использовать для обращения к какому-либо другому сегменту.

Директива ASSUME необходима перед использованием сегментных регистров и перед каждой точкой в программе, в которой при выполнении программы может модифицироваться содержимое сегментных регистров. Во всех обращениях к переменным в памяти будет использоваться один из сегментных регистров DS, ES или SS. Следовательно, указание о содержимом этих регистров должно появиться в директиве ASSUME до того места в программе, где будет происходить фактическое обращение к переменной в памяти. Аналогично метки команд неявно связаны с сегментным регистром CS, поэтому указание о его содержимом должно появиться в директиве ASSUME до того места, где осуществляется передача управления ячейкам. Если какой-либо сегментный регистр не будет использоваться в модуле, то директива ASSUME будет иметь вид ASSUME NOTHING.

Особенно важна директива ASSUME для регистра CS, так как он определяет ячейки памяти, в которых находятся команды. Смещения команд находятся в программном счетчике PC, и он определяет следующую по порядку выполнения команду. В большинстве случаев это допустимо, если регистр CS содержит базу сегмента, предполагаемую для данной команды во время ассемблирования. Если регистр CS содержит другое значение, то результат окажется непредсказуемым и трудно исправимым. По этой причине ассемблер сохраняет соответствующую информацию о предполагаемых значениях регистра CS для каждой метки и команды. Он запрещает внутрисегментные (NEAR) передачи управления меткам со значением регистра CS, которое отличается от предполагаемого содержимого регистра CS для команды передачи управления. Однако такой запрет не имеет силы при межсегментных (FAR) передачах управления, в которых модифицируется содержимое регистра CS. В дальнейшем будет использоваться новое значение из регистра CS, а у метки требуется новая директива ASSUME.

Сообщая информацию о содержимом сегментных регистров во время выполнения программы, директива ASSUME позволяет значительно сократить число команд, требующих операторов замены сегмента. Если директива ASSUME отсутствует, то во всех обращениях к переменной в памяти необходимо явно указывать сегментный регистр, содержащий базу для каждого конкретного обращения к памяти. Это требуется ассемблеру, чтобы при необходимости

сформировать префиксы замены сегментов. Поясним это на примере.

Пусть переменные SRC и DST определяются в сегменте BIG, а переменная CLUE — в сегменте SMALL. Чтобы реализовать оператор CLUE :=SRC—DST+15 потребуется следующий фрагмент:

```
BIG     SEGMENT
        SRC DW ?                ; Переменные SRC и DST — в сег-
        DST DW ?                ;     менте BIG
BIG     ENDS
SMALL   SEGMENT
        CLUE DW ?               ; Переменная CLUE — в SMALL
SMALL   ENDS
CODE    SEGMENT
        ASSUME CS:CODE
        MOV AX, SMALL           ; DS адресует сегмент SMALL
        MOV DS, AX
        MOV AX, BIG             ; ES адресует сегмент BIG
        MOV ES, AX
; Предполагается, что SRC и DST имеют какие-либо значения
        MOV AX, ES:SRC
        SUB AX, ES:DST
        ADD AX, 15
        MOV DS:CLUE, AX
CODE    ENDS
```

Как видно из этого фрагмента, регистр ES адресует сегмент BIG, а регистр DS — сегмент SMALL. Указания о сегменте с помощью оператора замены сегмента требуются при каждом обращении к переменной в данном сегменте. Таких же операторов замены сегмента требуют обращения к данным, находящимся в сегментах кода и стека.

Директива ASSUME дает возможность сообщить ассемблеру один раз (а не в каждой команде!) о том, какие сегментные регистры должны использоваться для обращений к переменным. Поэтому приводимый ниже фрагмент эквивалентен предыдущему (определения сегментов не показаны, так как они не изменяются):

```
ASSUME  CS:CODE, DS:SMALL, ES:BIG
MOV     AX,SMALL                ; Инициализация сегментных
MOV     DS,AX                   ;     регистров
MOV     AX,BIG
MOV     ES,AX
MOV     AX,SRC
SUB     AX,DST
ADD     AX, 15
MOV     CLUE,AX
```

В этом варианте операторы замены сегментов вообще не потребовались.

Директива ASSUME необходима в начале каждого блока команд, в котором происходит модификация содержимого сегментного регистра. Не требуется кодировать операторы замены сегмента для тех переменных, базовый адрес которых предполагается (согласно директиве ASSUME) в каком-либо сегментном регистре. Эту операцию при необходимости реализует ассемблер. При правильном употреблении директив ASSUME программист почти не заботится об операторах замены сегментов. Если базовый адрес, требующийся для обращения к некоторой переменной, не «предполагался» (не фигурировал в директиве ASSUME) в каком-либо сегментном регистре и нет оператора замены сегмента, ассемблер зафиксирует ошибку.

Каждое обращение к данным (переменной) или передача управления по метке (команде) проверяется на соответствие первоначально указанного сегмента сегментам, «предполагаемым» в сегментных регистрах во время выполнения. Если сегмент с объектом обращения «предполагается» и каком-либо сегментном регистре, ассемблер генерирует правильную машинную команду, в которой при необходимости имеется и префикс замены сегмента. Выбор ассемблером сегментных регистров зависит от адресного выражения.

Директива ORG. Основной внутренней переменной ассемблера является счетчик ячеек (адресов), который при ассемблировании выполняет функцию, аналогичную функции программного счетчика при выполнении программы. Счетчик ячеек сообщает ассемблеру адрес следующей ячейки памяти (имеется в виду смещение в сегменте), которая предназначена для размещения следующего байта команды или данных.

Первое появление директивы <*имя*> SEGMENT определяет начало сегмента с заданным именем. При этом организуется новый счетчик ячеек, в который загружается нулевое значение, так как первый байт в сегменте имеет нулевое смещение. При распределении каждого следующего байта производится инкремент счетчика ячеек. Директива ENDS с тем же именем освобождает данный счетчик ячеек до тех пор, пока этот же сегмент не будет открыт еще одной директивой SEGMENT (см. выше вложенные сегменты). В этом случае счетчик ячеек продолжает счет распределяемых байт со старого значения.

Текущее значение счетчика ячеек может быть принудительно изменено программистом с помощью директивы ORG (начало), имеющей следующий формат:

ORG <*выражение*>

При выполнении директивы ORG вычисляется значение *выражения* и полученный численный результат загружается в счетчик ячеек. Ассемблирование следующих команд или элемента данных производится по полученному адресу (смещению в текущем сегменте). В *выражении* могут фигурировать абсолютные и переместимые числа, а также ранее определенные имена. Вычисление значения *выражения* производится по модулю 64К, т. е. оно, будучи смещением в сегменте, считается беззнаковым целым в диапазоне 0—65 535.

Примечание. Во многих исходных модулях директива ORG не используется. Если после первого определения сегмента директива ORG отсутствует, ассемблирование начинается с нулевого содержимого счетчика ячеек.

3.4. Директивы определения имен

В процессе ассемблирования ассемблер автоматически присваивает значения символическим именам, которыми являются метки команд или имена переменных в операторах распределения данных. Присваиваемым значением является текущее содержимое счетчика ячеек. Другими словами, и метки, и имена переменных в директивах DB, DW и DD ассоциируются со смещениями в текущем сегменте.

Директива EQU. В языке ассемблера МП К1810 предусмотрено мощное средство приравнивания (*equate*), которое позволяет программисту определять символические имена для часто используемых выражений. Такие имена создаются с помощью директивы EQU, имеющей следующий формат:

<*имя*> EQU <*выражение*>

Поле *выражения* может определять константы, адреса, регистры и даже мнемокоды макрокоманд. В поле *имени* находится идентификатор, т. е. имя, которое программист использует для представления (в известном смысле для замены) *выражения*. Естественно, именам, как и всегда, целесообразно придавать содержательный смысл.

Покажем удобство применения директивы EQU на про-

стом примере. Пусть программа оперирует таблицей, содержащей 200 элементов, и приходится часто использовать значение, равное максимальному числу элементов. Конечно, при всякой необходимости указания этого числа можно использовать как непосредственный операнд число 200, например в команде MOV CX,200. Однако в этом очевидном способе имеются два недостатка. Во-первых, число 200 может обозначать в программе что-то еще, и необходимы подробные комментарии о конкретном «смысле» числа 200 в каждом случае. Во-вторых, при расширении таблицы, например до 500 элементов, приходится отыскивать во всей программе число 200 и заменять его числом 500. Предположим теперь, что программист определил имя для представления максимального числа элемеитов в таблице следующей директивой

```
MAX_SIZE  EQU  200
```

Такое определение позволяет использовать имя MAX_SIZE в любой команде, где требуется максимальное число элементов в таблице, например в команде MOV CX, MAX_SIZE. Очевидно, этот прием значительно улучшает понимаемость программы, а при расширении таблицы достаточно только заменить в директиве EQU число 200 числом 500 и произвести повторное ассемблирование программы. Ассемблер автоматически заменит каждое появление MAX_SIZE числом 500.

Определенные с помощью директив EQU имена не разрешается переопределять, т. е. любое имя может появиться только в одной директиве EQU. Следовательно, эти имена сохраняют одно и то же значение во всей программе, если не будут отменены директивой PURGE.

Директива EQU определяет имя как синоним другого имени в таблице имен или некоторой константы. Примеры использования директивы EQU приведены ниже.

```
CR      EQU   0DH              ; Численная константа
LF      EQU   0AH
BET     EQU   ALPHA [SI] +3    ; Адресное выражение
COUNT   EQU   CX               ; Регистр
```

Имена, присвоенные директивами EQU, можно использовать в любых операторах исходного модуля, допускающих выражения из правой части директив EQU, например:

```
MOV     CL,LF      ; Загрузить в CL значе-
                   ;   ние 0A
INC     COUNT      ; Инкремент CX
```


Директиву EQU можно использовать для создания альтернативных имен параметров, передаваемых процедуре в стеке. Пусть процедура PROC-A типа NEAR обращается к двум параметрам в стеке через регистр BP. Предположим, что параметрами являются байт, определяющий некоторый диапазон (*range*), и слово, представляющее собой текущее значение скорости (*rate*). После обычных первых команд процедуры PUSH BP и MOV SP,BP в стеке обеспечивается доступ к параметрам с помощью альтернативных имен, вводимых директивами EQU:

```
RANGE   EQU   BYTE PTR [BP+6]
RATE    EQU   WORD PTR [BP+4]
```

Введение альтернативных имен позволяет улучшить понимаемость программы. Например, команда MOV AL, RANGE более понятна, чем стандартная команда MOV AL, [BP+ +6].

Директива PURGE. Директива PURGE (удалить, освободить) предназначена для удаления имен из таблицы имен, которую формирует ассемблер. Эта директива имеет следующий формат:

PURGE *<имя>*, [*<имя>*] . . .

Сама директива PURGE не должна отмечаться.

Если какое-либо *имя* удалено с помощью директивы PURGE, его можно переопределить. Во всех последующих случаях появления *имени* после его удаления и переопределения будет использоваться самое последнее значение *имени*. Если *имя* было удалено и не определено снова, его появление в программе будет отмечено как ошибка («неопределенное имя»).

3.5. Выражения

Адресные и числовые выражения. Наиболее сложными конструкциями полей операндов ассемблерных строк оказываются выражения, т. е. допустимые элементы этих полей, связанные разрешенными в языке ассемблера операторами. Здесь и далее под оператором понимается некоторая функция с четко определенным результатом.

Важнейшими элементами выражений являются символические имена. Напомним, что в языке ассемблера существуют имена двух классов. Переменная — это имя (адрес) ячейки памяти, содержимое которой считается данными.

Переменные определяются с помощью директив DB, DW и DD распределения данных. Каждая переменная имеет три атрибута: сегмент, смещение и тип (см. § 3.2). Вместе с тем метка представляет собой имя (адрес) ячейки памяти, содержимое которой интерпретируется как машинная команда. Обычно метка определяется указанием в поле метки имени с последующим двоеточием. Но метки могут определяться директивами LABEL и в определениях процедур и без завершающего двоеточия. Метки имеют четыре атрибута: сегмент, смещение, предположение о регистре CS и расстояние (тип NEAR или FAR). В совокупности переменные и метки можно отнести к адресным именам.

Символическими именами могут обозначаться также любые числовые значения. Для этого в языке ассемблера предусмотрена директива EQU. В этом случае имя заменяет число и может быть названо «числовым именем».

В соответствии с классами символических имен в языке ассемблера появляются и выражения двух классов. Адресным выражением называется выражение, которое вычисляется для получения адреса памяти. Следовательно, одиночные переменные и метки представляют собой простейшие адресные выражения. Но адресные выражения могут быть и более сложными, например, в них могут фигурировать имена регистров BX, BP, SI и DI, заключенные в квадратные скобки. Любое адресное выражение имеет три обязательные составляющие компоненты: сегмент, смещение и тип.

Ко второму классу относятся числовые выражения, в результате вычисления которых получаются числа. Из приведенных определений следует, что каждый компонент адресного выражения представляет собой число, но само оно числом не является.

Чтобы подчеркнуть различия между адресными и числовыми выражениями, рассмотрим встроенный оператор OFFSET (смещение). Этот оператор, находящийся перед адресным выражением, возвращает смещение адресного выражения как число. Рассмотрим следующий пример простой программы:

```
DATA    SEGMENT  AT 100H
        DB       0,0,0,0
  NUM   DB       77H
DATA    ENDS
CODE    SEGMENT
```



```
        START  :  MOV    AX,DATA     ; Базовый адрес сегмента
                  MOV    DS,AX       ;   (0100) в регистре DS
        M1:       MOV    CL,4        ; Число 4 в регистре CL
        M2:       MOV    CL, NUM     ; Адресное выражение
                                     ;   NUM; в регистре CL
                                     ;   будет 77
        M3:       MOV    CL, OFFSET NUM ; В регистре CL число 4
CODE            ENDS
                END    START
```

Символическое имя NUM представляет собой адресное выражение, поэтому команда с меткой M2 загружает в регистр CL значение 77 из ячейки памяти с адресом NUM. Однако выражение OFFSET NUM дает число, и команда с меткой M3 загрузит в регистр CL смещение адресного выражения NUM, равное числу 4, так как предыдущая директива DB резервирует 4 байта от начала сегмента. Число 4 превратится в часть команды как непосредственный операнд. Такое же действие выполняет команда с меткой M1.

Таким образом, использование адресного выражения в поле операнда команд преобразований данных означает, что операндом будет содержимое адресуемой ячейки памяти. Адресное выражение в поле операнда команд передачи управления (условные и безусловные переходы, вызовы и возвраты) определяет встраиваемый в команду адрес перехода. Если в команде фигурирует числовое выражение, то его значение (число) превращается в непосредственный операнд.

Максимальный диапазон чисел составляет от —FFFF до FFFF. Выход за границы этого диапазона приводит к выдаче сообщения об ошибке (переполнение). Все арифметические операции выполняются в дополнительном коде. Вычисляя значения числовых выражений, ассемблер добавляет слева дополнительный бит, считающийся знаковым, а в окончательном результате отбрасывает его. Например, в директивах DB разрешаются значения от —256 до 255. При этом любое значение, находящееся между —256 и —129, представляется положительным числом от 0 до 127 соответственно. Аналогичная ситуация справедлива для директивы DW, но с расширением диапазона. Так как с адресным выражением ассоциируются три компонента, оно будет недопустимым при выходе за границы диапазона любого компонента.

Арифметические операторы. В выражениях допускаются бинарные арифметические операторы сложения (+), вы-

читания (—), умножения (*) и деления (/), а также унарные операторы плюс и минус. Кроме того, имеются «буквенные» операторы:

MOD — вычисляет остаток от деления левого операнда на правый;

SHL — осуществляет сдвиг влево левого операнда на число бит, определяемое значением правого операнда;

SHR — осуществляет сдвиг вправо правого операнда на число бит, определяемое значением левого операнда.

При выполнении сдвига в освобождающиеся биты помещаются нули, а выдвигаемые биты теряются. Кроме того, если правый операнд оказывается отрицательным числом, направление сдвига меняется на противоположное.

Арифметические операторы должны отделяться от своих операндов хотя бы одним пробелом. Это условие позволяет избежать интерпретации их как части символических имен, вводимых программистом.

Употребление арифметических операторов сложения и вычитания подчиняется следующим правилам:

абсолютное число всегда можно складывать или вычитать из переменной, метки, абсолютного или переместимого числа; когда число прибавляется к переменной (метке), результатом будет переменная (метка), смещение которой равно сумме числа и первоначального смещения переменной (метки);

вычитание переменных и меток допускается только в том случае, если они находятся в одном сегменте;

сложение переменных и меток не допускается;

допустимо сложение одного базового и одного индексного регистров, например [BX+SI]; абсолютные или переместимые числа могут прибавляться или вычитаться из таких регистров или выражений, но регистры не могут вычитаться из чисел.

Операторы умножения, деления, сдвига и вычисления остатка могут выполнять действия только с явными числами.

Логические операторы. Бинарные логические операторы AND,OR и XOR выполняют операции конъюнкции, дизъюнкции и исключающего ИЛИ (сложения по модулю 2) над отдельными битами операндов. Унарный оператор NOT отрицания осуществляет инверсию каждого бита операнда.

Когда в выражении присутствуют несколько логических операторов, первым выполняется оператор NOT, следующим — AND и последними OR и XOR, имеющие одинаковое старшинство.

Все логические операторы, как и арифметические, могут выполнять операции только с явными числами.

Операторы отношений. Ассемблер разрешает применять стандартные операторы отношений GT (больше), GE (больше или равно), EQ (равно), NE (не равно), LE (меньше или равно), LT (меньше). Результатом этих операторов является булева переменная со значениями «истина» или «ложь». Значение «истина» представляется 16-битным числом, состоящим из единиц, а значение «ложь» — числом, содержащим 16 нулей. Чтобы можно было сравнивать переменные или метки, они должны быть определены в одном и том же сегменте. Операторы отношений сравнивают смещение таких операндов. Сравнивать с переменными или метками числа не разрешается.

Один из вариантов дальнейшего использования в выражении результата операторов отношений предполагает употребление оператора AND. Например, запись

```
MOV BH, 50 AND A EQ B
```

приведет к генерированию команды MOV BH,50 или MOV BH,0 в зависимости от того были равны или нет во время ассемблирования значения A и B.

Старшинство операторов. Вычисление выражения производится слева направо. Операторы с более высоким приоритетом выполняются прежде других операторов, которые непосредственно предшествуют им или следуют за ними. Когда два оператора имеют одинаковый приоритет, первым выполняется оператор, находящийся слева. Как обычно, для изменения порядка выполнения операторов применяются круглые скобки (подвыражение, заключенное в круглые скобки, вычисляется первым). При наличии вложенных круглых скобок первым вычисляется подвыражение, находящееся во внутренних скобках.

Операторы в порядке возрастания старшинства упорядочиваются следующим образом:

оператор усечения (SHORT);
логические операторы OR и XOR;
логический оператор AND;
логический оператор NOT;
операторы отношений;
операторы сложения и вычитания;

операторы умножения, деления, вычисления остатка и сдвига;
операторы HIGH (старший) и LOW (младший);
операторы манипуляций переменными;
выражения, заключенные в круглые скобки, операторы LENGTH, SIZE, TYPE, WIDTH и выражения, находящиеся в квадратных скобках.

Оператор SHORT сообщает ассемблеру о том, что для представления окончательного значения выражения будет достаточно 1 байта. Поэтому машинный код генерируется только в том случае, если при вычислении выражения будет получено значение длиной в 1 байт. Если значение результата превышает 1 байт, формируется сообщение об ошибке. В тех ситуациях, когда выражение с оператором SHORT участвует в дальнейших арифметических операциях, действие оператора теряется. Обычно оператор SHORT применяется в командах условного перехода, осуществляющих передачу управления в прямом направлении (вперед). С его помощью программист сообщает о том, что смещение в команде должно быть длиной 1 байт.

Замена сегмента. Оператор замены сегмента, который помещается перед адресным выражением, имеет вид *<сегментный регистр:>*. В качестве *сегментного регистра* могут фигурировать имена сегментных регистров CS, DS, SS и ES. Этот оператор предназначен для изменения такого атрибута переменной, как сегмент.

Напомним, что каждая команда с обращением к памяти формирует физический адрес с обязательным привлечением одного из сегментных регистров, поэтому ассемблер должен определить используемый сегментный регистр. Такое определение для каждой команды зависит только от вида находящегося в команде адресного выражения и текущих значений, предполагаемых (директивой ASSUME) в сегментных регистрах.

Алгоритм выбора сегментного регистра встроен в программу-ассемблер. При наличии нескольких дополнительных вариантов (например, когда одна и та же база содержится в нескольких регистрах) ассемблер по возможности формирует команду без префикса замены сегмента. Когда возникает необходимость, программист может отменить выбор ассемблера — при явном указании сегментного регистра ассемблер образует и автоматически помещает перед командой префикс замены сегмента.

Префикс замены необходим, если требуемая ячейка

памяти доступна только с помощью сегментного регистра, отличающегося от используемого по аппаратному умолчанию. Если к ней нельзя обратиться с помощью предполагаемого директивой ASSUME любого сегментного регистра, команда считается ошибочной.

Приняты следующие правила анализа адресных выражений и аппаратные варианты по умолчанию, которые нельзя заменять:

любое адресное выражение типа NEAR всегда ассоциируется с регистром CS;

любое адресное выражение типа FAR всегда ассоциируется с регистрами CS и PC;

в командах PUSH, POP, CALL, RET и IRET с неявным использованием указателя стека сегментным регистром всегда служит SS;

если в цепочечной команде операнд-получатель адресуется через регистр DI, то с этим операндом всегда используется сегментный регистр ES.

Когда в адресном выражении фигурирует регистр BP (его имя находится в квадратных скобках), сегментным регистром по умолчанию является регистр SS, но его замена допускается.

Выражение

<сегментный регистр:> <адресное выражение>

порождает новое адресное выражение, сегментным компонентом которого является указанный *сегментный регистр*. Если он совпадает с сегментным регистром, который принят по умолчанию для адресного выражения, префикс замены сегмента не формируется. В противном случае ассемблер образует перед командой байт префикса замены сегмента.

Для пояснения изложенных правил приведем следующий пример:

```
                ASSUME  CS:CODE,DS:DATA
DATA            SEGMENT
    MYBYTE      DB      ?
DATA            ENDS
CODE            SEGMENT
                MOV     DL,[BP]         ;По умолчанию
                                        ;принимается
                                        ;SS, префикса
                                        ;замены нет
                MOV     DL,DS:[BP]      ;Формируется
                                        ;префикс замены
```

Продолжение

```
        MOV     DL,MYBYTE       ; Префикса замены
                                ;   нет, регистр
                                ;   DS указан в
                                ;   ASSUME
        ASSUME  DS:NOTHING,ES:DATA
        MOV     CL,MYBYTE       ; Формируется пре-
                                ;   фикс для ES
        MOV     CL,DS:MYBYTE    ; Явно указан DS,
                                ;   префикс за-
                                ;   мены не нужен
CODE    ENDS
```

Префикс замены обязательно указывается перед адресным выражением в следующих ситуациях:

если регистр BP используется для индексирования в сегменте, отличающемся от сегмента стека;

если в цепочечной команде адресация операнда осуществляется через регистр SI, а данные находятся в дополнительном сегменте, например ES:[SI];

если программист использует данные в дополнительном сегменте; при этом любое адресное выражение с регистрами (имя регистра в квадратных скобках) требует замены сегмента, например ADD AX, ES:[BX].

Замена сегмента всегда относится только к одной команде с единственным исключением, когда замена указана в директиве EQU. Например, если записать

```
ITEM  EQU  ES:ARRAY[SI]
```

то каждое последующее появление имени ITEM в любых командах будет означать использование ES в качестве сегментного регистра.

Оператор PTR (указатель) создает переменную или метку. Новая переменная имеет те же смещение и сегмент, что и операнд справа от ключевого слова PTR, и атрибут, указанный программистом слева от PTR. Оператор PTR изменяет атрибут только в одной команде. Применение оператора PTR для изменения типа переменных рассмотрено в § 3.2.

С помощью данного оператора можно определять также тип метки. Например, в команде

```
JMP  FAR  PTR  TARGET
```

тип метки TARGET определен как FAR (межсегментный переход). Поэтому в этой команде будет находиться два

слова адреса, а не одно, как обычно. В команде JMP BELOW тип метки BELOW по умолчанию принимается NEAR.

Оператор PTR, используемый с явным числом, создает переменную или метку со следующими атрибутами: сегмент равен нулю, смещение равно числу, а тип указан программистом слева от оператора PTR. Такое выражение допустимо, если ему предшествуют префикс замены сегмента или операторы TYPE и OFFSET. Выражение DS:BYTE PTR 85 временно определяет переменную с типом BYTE и смещением 85 в сегменте, адресуемом регистром DS. Без префикса замены данное выражение не имеет атрибута сегмента и было бы недопустимым. Команда MOV CL, DS:BYTE PTR 85 загрузит в регистр CL содержимое байта, находящегося на расстоянии 85 байт от начала текущего сегмента данных. Однако такой прием менее целесообразен и удобен, чем определение имени для нужной ячейки.

С помощью оператора PTR устраняется неоднозначность типа переменной или метки в некоторых командах. Примерами могут служить следующие команды:

```
NOT   [BX]    ; Инвертировать байт или сло-
              ;   во?
INC   [SI]    ; Инкремент байта или слова?
JMP   [BX]    ; Переход типа NEAR или FAR?
```

Правильная запись таких команд имеет вид:

```
NOT  WORD  PTR  [BX]    ; Инвертировать слово
INC  BYTE  PTR  [SI]    ; Инкремент байта
JMP  DWORD PTR  [BX]    ; Межсегментный переход
```

Оператор THIS. Назначение данного оператора — создать переменную или метку, тип которой должен быть обязательно указан программистом, а смещение и сегмент являются текущими значениями счетчика адреса (ячеек) программы-ассемблера.

Обычно оператор THIS применяется для определения альтернативного имени с другим типом в том месте, где производится определение основного имени. Такая необходимость возникает, например, в ситуациях, когда к одним и тем же ячейкам памяти приходится обращаться по командам, оперирующим с переменными различных

типов. Ниже показано определение альтернативных имен:

```
DATA    SEGMENT   PARA     'DATA'
        W_ONE     EQU      THIS WORD
        B_ONE     DB       100 DUP(0)
        B_TWO     EQU      THIS BYTE
        W_TWO     DW       300 DUP(?)
        W_THREE   EQU      THIS WORD
        B_THREE   EQU      THIS BYTE
        THREE              100 DUP(?)
        CONT
        .
        .
        .
DATA    ENDS
```

В этом фрагменте определения данных имя W_ONE позволяет обращаться к парам байт в массиве B_ONE как к словам. Атрибуты смещения и сегмента для переменных W_ONE и B_ONE одинаковы. Аналогично имя B_TWO позволяет обращаться к байтам массива W_TWO, элементы которого имеют тип WORD, задаваемый директивой DW. Массив THREE определен как массив двойных слов. Альтернативные имена B_THREE и W_THREE позволяют обращаться к байтам и словам этого массива соответственно.

Операторы SEG, OFFSET и TYPE. Данные операторы предназначены для образования чисел путем выделения соответствующих атрибутов (сегмент, смещение и тип) переменных или меток. Для приведенного выше фрагмента определения данных имеем следующие значения операторов SEG, OFFSET и TYPE:

```
SEG B_ONE = SEG W_TWO = SEG THREE = DATA,
  OFFSET B_ONE = 0,
  OFFSET W_TWO = 100,
  OFFSET THREE = 700,
  TYPE B_ONE = TYPE B_TWO = TYPE B_THREE = 1,
  TYPE W_ONE = TYPE W_TWO = TYPE W_THREE = 2,
  TYPE THREE = 4.
```

Использование оператора TYPE, а также операторов LENGTH и SIZE было показано в § 3.2.

Процедура (или замкнутая *подпрограмма*) является одним из средств разработки модульных программ. Она представляет собой законченную командную последовательность, которая приводится в действие командой вызова CALL. Широкое применение процедур имеет большое значение при проектировании прикладных программ благодаря следующим положительным качествам:

сложную программу можно разделить на небольшие, сравнительно простые и управляемые модули, разработка которых может производиться несколькими программистами с соответствующим сокращением сроков разработки всей программы;

каждая процедура допускает автономную отладку, что ускоряет отладку всей прикладной программы;

процедуры сокращают длину прикладной программы, обеспечивая экономию программной памяти;

отлаженные процедуры организуются в библиотеки.

Команда вызова CALL содержит метку одной из команд процедуры; обычно ею является первая команда процедуры, но это необязательно. Метка в процедуре, которой передает управление команда CALL, называется точкой входа процедуры. При выполнении команды вызова адрес возврата, т. е. адрес команды, находящейся после команды CALL, включается в стек, а затем управление передается в точку входа. Процедура выполняется до тех пор, пока не встретится команда возврата RET, которая извлекает из стека запомненный адрес возврата, и управление передается в точку вызова. Разумеется, в процедуре в зависимости от ее функций может быть несколько команд возврата.

Процедура обычно разрабатывается как функциональный блок, который формирует набор выходных данных путем преобразования четко определенного набора входных данных, называемых параметрами. Поэтому с каждой процедурой ассоциируется способ передачи ей необходимых параметров и получения из нее результатов.

Директивы PROC и ENDP. Для организации процедур в языке ассемблера предназначены директивы PROC (procedure) и ENDP. Директива PROC отмечает точку входа процедуры, а директива ENDP — окончание про-

цедуры. Формат этих директив имеет следующий вид:

```
<имя>   PROC    <тип>
        .
        .
        .
        Тело процедуры
        .
        .
        .
<имя>   ENDP
```

В обеих директивах, как и в директиве SEGMENT/ /ENDS, должно находиться одно и то же *имя* процедуры. Оно представляет собой метку, указывающую точку входа процедуры. Справа от ключевого слова PROC указывается *тип* процедуры NEAR или FAR (по умолчанию принимается тип NEAR). Необходимость типизации процедур объясняется сегментной организацией памяти в МП К1810. Ассемблер использует *тип* для определения того, какую команду CALL генерировать при вызове процедуры. Если указан *тип* NEAR, процедура находится в том же сегменте кода, в котором находятся все команды CALL, вызывающие данную процедуру. В этом случае ассемблер генерирует команду внутрисегментного вызова, т. е. машинная команда CALL содержит только смещение точки входа. При передаче управления в стеке запоминается, а затем восстанавливается только содержимое программного счетчика PC. Если указан тип FAR, процедура находится в сегменте, отличающемся от того сегмента кода, в котором находятся команды вызова CALL. Следовательно, ассемблер должен генерировать длинную команду межсегментного вызова, которая содержит базу и смещение точки входа. При передачах управления в стеке приходится запоминать, а затем восстанавливать содержимое регистров CS и PC.

В соответствии с двумя формами команд вызова необходимы и две разновидности команды возврата RET. Команда возврата в процедуре типа NEAR должна восстанавливать из стека только содержимое PC, а команда RET в процедуре типа FAR — содержимое регистров CS и PC. Обе разновидности команд возврата имеют один и тот же мнемокод RET, (но, разумеется, различные машинные коды операций). Ассемблер определяет генерируемую разновидность команды RET на осиове типа, указанного в директиве PROC. Этим объясняется и необходимость директивы ENDP — пара директив PROC/ENDP ограничивает

область, в которой ассемблер генерирует только одну разновидность команды возврата.

Приведем элементарный пример определения процедуры, которая помещает в регистр CX среднее значение содержимого регистров AX и DX.

```
AVRG    PROC    NEAR
        MOV     CX, AX  ; Передать (AX) в CX
        ADD     CX, DX  ; Прибавить (DX) к CX
        RCR     CX, 1   ; Сдвинуть (CX) вправо
        RET
AVRG    ENDP
```

Процедура начинается с команды MOV CX, AX, и точка входа отмечена именем AVRG. Так как процедура имеет тип NEAR, то команда CALL AVRG в любом месте программы приведет к формированию короткой команды CALL, которая воздействует только на программный счетчик PC. Команда RET находится внутри пары директив PROC/ENDP, определяющей процедуру типа NEAR, поэтому ассемблер сформирует машинную команду возврата, которая будет восстанавливать из стека только значение PC.

Таким образом, действие директив PROC/ENDP заключается в том, чтобы именовать процедуру и установить ее тип. В соответствии с типом ассемблер генерирует требуемые команды CALL и RET. В отличие от процедур языков высокого уровня ассемблерные процедуры не имеют ограниченной области действия. Это означает, что к переменным или меткам внутри пары PROC/ENDP можно обращаться из любого места программы. Кроме того, командная последовательность, находящаяся сразу перед директивой PROC, в случае появления ошибки войдет в тело процедуры вместо того, чтобы обойти его. Для предотвращения случайного выполнения процедуры без ее вызова целесообразно размещать процедуры выше (по меньшим адресам) тех программ, которые их вызывают, а также избегать вложенных определений PROC/ENDP. Вложение процедур аналогично вложению сегментов и в данном контексте означает, что одна процедура полностью находится в другой.

Значения, которые процедура использует как входные данные, называются *параметрами*. Параметрами могут быть численные значения, адреса и т. д. Имеется несколько способов передачи параметров процедурам. Один из про-

стых способов заключается в том, чтобы значения параметров разместить в регистрах МП. В приведенной выше процедуре AVRG два входных параметра передаются в регистрах AX и DX. До ее вызова в эти регистры необходимо загрузить усредняемые значения. Еще один способ передачи параметров связан с использованием общей области данных, доступной вызывающей программе и процедуре. Однако в большинстве языков высокого уровня передача параметров процедурам организуется через стек. Наличие в МП К1810 регистра BP, для которого базой служит сегментный регистр SS, обеспечивает простой доступ к параметрам, передаваемым в стеке (см. § 2.3). Целесообразно придерживаться этого способа и в ассемблерных программах, но, в общем, выбор способа передачи параметров зависит от конкретных функций, выполняемых процедурой.

Процедура, которая возвращает одно значение, называется процедурой-функцией. Примером служит та же процедура AVRG, вычисляющая арифметическое среднее содержимого регистров AX и DX и возвращающая результат в регистр CX. Чтобы упростить объединение исходных модулей, составленных на языке ассемблера и языках высокого уровня, предусматривается соглашение о том, где возвращает значение любая процедура-функция. В языке ассемблера МП К1810 следует придерживаться следующего соглашения: значение переменной типа WORD возвращается в аккумуляторе AX, значение переменной типа BYTE — в аккумуляторе AL, короткий указатель (смещение) возвращается в регистре BX и, наконец, длинный указатель (база : смещение) — в регистрах ES : BX.

Директива LABEL (отметить) выполняет несколько функций, одной из которых является обеспечение нескольких точек входа в процедурах.

Напомним, что в языке ассемблера применяются символические имена, обозначающие адреса ячеек памяти, которые содержат команды или переменные (данные). Имена, содержащие адресную информацию о командах, называются метками.

Директива LABEL может создавать имена для любых ячеек памяти независимо от их содержимого и предполагаемого использования. Она содержит информацию о типе определяемого имени, а тип однозначно указывает допустимое употребление имени. Директива LABEL имеет следующий формат:

<имя> LABEL *<тип>*

В качестве *типа* фигурирует одно из пяти ключевых слов: BYTE, WORD, DWORD, NEAR и FAR. Если в директиве LABEL указан тип BYTE, WORD или DWORD, то *имя* представляет собой переменную. Имена переменных допустимы во всех командах, оперирующих данными, но их нельзя употреблять в командах передачи управления. Имя с типом NEAR или FAR представляет собой метку, которая может быть операндом в командах передачи управления, но недопустима в командах манипуляции данными.

Приведем некоторые примеры использования директивы LABEL. Две строки в программе

```
BUFFER   LABEL   WORD
         DW      0FFFFH
```

эквивалентны одной наименованной директиве DW:

```
BUFFER   DW      0FFFFH
```

В обоих случаях ассемблер присваивает текущему адресу имя BUFFER и полагает, что по этому адресу хранится переменная типа WORD с начальным значением FFFF. После этого имя BUFFER может фигурировать в командах манипуляций данными, например:

```
MOV   AX, BUFFER    ; Передать значение BUFFER в AX
INC   BUFFER        ; Инкремент переменной BUFFER
```

Однако команды передачи управления, содержащие это имя, например JMP BUFFER, не имеют смысла.

Директива

```
TARGET   LABEL   NEAR
```

определяет имя TARGET как метку. Определение меток разрешено только тогда, когда ассемблируемый в текущий момент сегмент предполагается находящимся в пределах достижимости регистра CS.

Директива LABEL обычно применяется для присваивания второго имени некоторой ячейке памяти, чтобы к ней можно было обращаться командами, оперирующими различными типами данных, без указания оператора замены типа. Если, например, одну и ту же область памяти в одном случае необходимо считать массивом байт, а в другом — массивом слов, то удобно определить ее следующим образом:

```
ARRAYB   LABEL   BYTE
ARRAY    DW      200 DUP (?)
```

Команда MOV AL, ARRAYB передает в аккумулятор AL первый байт массива, а команда MOV AX, ARRAY пере-

дает в аккумулятор AX первое слово массива. Если бы директивы LABEL не было, для передачи в AL первого байта массива потребовалась бы более громоздкая команда

```
MOV     AL, BYTE PTR ARRAY
```

Как уже отмечалось, директива LABEL позволяет также указать несколько точек входа в процедуре. Такое указание становится обязательным в процедурах типа FAR. Для создания альтернативной точки входа перед ней необходимо записать директиву LABEL.

Наконец, директива LABEL удобна для определения начальной вершины стека, например:

```
STACK_1     SEGMENT
            DW 20 DUP(?)
TOS         LABEL WORD
STACK_1     ENDS
```

Директива DW резервирует 20 слов памяти без их инициализации. Директива LABEL именует TOS слово, находящееся после 20 слов от начала сегмента STACK_1. Смещение TOS от начала сегмента STACK_1 представляет собой то значение, которое будет содержать указатель стека SP, когда стек пустой.

3.7. Директивы связи модулей и сегментов

Основной принцип структурного программирования заключается в разделении сложной прикладной программы на несколько сравнительно автономных блоков или модулей. При этом модульность программы понимается двояко: 1) модульная программа состоит из нескольких исходных модулей, допускающих ассемблирование (или компилирование) независимо друг от друга; 2) модульная программа содержит несколько процедур, выполняющих необходимые преобразования данных. Вопросы организации процедур были рассмотрены в § 3.6, поэтому в данном параграфе обсуждаются вопросы программирования с несколькими исходными модулями и средства языка ассемблера, поддерживающие такое программирование.

Разработка сложных прикладных программ как набора исходных модулей имеет несколько преимуществ. Одним из ее очевидных положительных качеств является то, что сравнительно небольшими модулями проще управлять. Например, при просмотре отдельных модулей приходится оперировать текстом меньшего объема, меньшим числом символических имен и т. п. Кроме того, модули обычно организуются как законченные функциональные блоки, которые допускают индивидуальные проверку и отладку. Наконец, отдельные модули можно программировать в зависимости от их функций на языках высокого уровня или на языке ассемблера.

Однако при модульном программировании возникают специфические проблемы *объединения* (связи, связывания, редактирования связей, компоновки) отдельных модулей в единую прикладную программу. Для этого необходимы специальные директивы ассемблера, а также специальная программа, называемая *редактором* связей или *компоновщиком*. Одна из стандартных проблем объединения модулей заключается в том, что из данного модуля возможны обращения (ссылки) к именованным элементам (переменным и меткам в первую очередь) в других модулях, а другие могут обращаться к таким же элементам данного модуля. Именно с помощью межмодульных обращений разобщенные модули объединяются в единую программу.

При ассемблировании (или компилировании) каждого исходного модуля порождается объектный модуль (чаще называемый объектным файлом), содержащий служебную информацию о межмодульных обращениях. Редактор связей использует эту информацию для объединения отдельных объектных файлов в единый объектный файл, в котором учтены все межмодульные обращения. Следовательно, получение многомодульной программы в такой форме, в какой ее можно загружать в память для выполнения, состоит из двух этапов. Сначала отдельные исходные модули ассемблируются, а затем образованные объектные файлы связываются в единый объектный файл. Ниже рассматриваются конструкции языка ассемблера, предназначенные для разработки многомодульных программ.

Описанная выше ситуация с межмодульными обращениями возникает при разработке программ для всех ЭВМ. Но при использовании МП К1810 возникает еще одна, характерная только для этого МП, проблема объединения логических сегментов, из которых состоят исходные модули. Она решается с помощью введения атрибутов сегментов, указываемых в директиве SEGMENT.

Директивы PUBLIC и EXTRN. Предположим, что программа состоит всего из двух модулей: MOD_1 и MOD_2

и что в модуле MOD_1 определена переменная BVAL_1 типа BYTE, к которой необходимо обращаться, и из модуля MOD_2. Чтобы обеспечить такое обращение в модуле MOD_1, необходима директива, которая объявит переменную BVAL_1 глобальной (*public*), т. е. доступной для обращения из других модулей. Для этого предназначена директива PUBLIC, и в модуле MOD_1 появляются следующие строки:

```
BVAL_1   DB 0              ; Определение переменной
         PUBLIC BVAL_1     ; Объявление ее глобальной
```

В модуле MOD_2 также необходима специальная директива, которая сообщала бы, что переменная BVAL_1 типа BYTE определена в другом модуле, т. е. является для MOD_2 внешним именем. Такую информацию сообщает директива EXTRN (*external* — внешний):

```
EXTRN    BVAL_1:BYTE    ; Внешняя переменная
```

Отметим, что директива EXTRN требует указания типа, а директива PUBLIC этого не требует. Объяснение этому довольно простое. Директива PUBLIC находится в том модуле, в котором определяется переменная BVAL_1, и ассемблер знает тип переменной. Но этого определения не видно из модуля MOD_2, поэтому директива EXTRN должна содержать информацию о типе.

Рассматриваемые директивы имеют следующий формат:

```
PUBLIC   <имя> ,[<имя>] . . .
EXTRN    <имя:тип> ,[<имя:тип>]. . .
```

Имя в обоих директивах является именем переменной, метки или константы, определенной директивой EQU. *Типом* может быть BYTE, WORD или DWORD для переменной, NEAR или FAR для метки и ABS для константы, причем ABS обозначает абсолютное число, а не переменную и не метку.

В поле операнда директивы PUBLIC находится список глобальных имен. Каждое имя должно быть определено в данном модуле и может быть указано в директиве PUBLIC только один раз. Разумеется, ни одно из имен, находящихся в списке, не должно быть внешним. Директивы PUBLIC размещаются в исходном модуле произвольно, но целесообразно сгруппировать их в начале модуля. Если после ассемблирования оказывается, что какое-либо имя из списка имен в директиве PUBLIC не определено, т. е. не содержится в таблице имен, ассемблер выдает сообщение об ошибке.

Каждый элемент в списке имен директивы EXTRN идентифицирует имя, к которому имеются обращения в данном модуле, но которое определено в другом модуле. Тип, фигурирующий в директиве EXTRN, должен совпадать с типом в определении имени. Если некоторое имя из списка имен директивы EXTRN определено в данном модуле, то это считается однократным определением имени, но ассемблер сообщает об этом как об ошибке. Если программист использовал какое-либо имя без определения в данном модуле и без указания в директивах EXTRN, ассемблер считает такое имя не определенным и формирует сообщение об ошибке.

Приведем примеры директив PUBLIC и EXTRN:

```
PUBLIC   SIN, COS, TAN, COTAN
EXTRN    ENTRY:BYTE, ADDR:FAR, START:WORD
```

Заметим, что отмечать директивы PUBLIC и EXTRN не разрешается.

Директива EXTRN внутри пары SEGMENT/ENDS сообщает ассемблеру о том, что обращения к перечисленным в ней именам осуществляются из этого логического сегмента. Важно, чтобы программист указал внешние имена, находящиеся в сегментах текущего модуля, внутри соответствующих пар SEGMENT/ENDS. Директивы EXTRN рекомендуется размещать в начале модуля или в начале сегмента, чтобы избежать некоторых проблем, связанных с обращением вперед.

Если ассемблер встречает обращение к внешнему имени, он не знает адреса или значения имени, поэтому он не может сформировать машинную команду стандартным образом. Вместо этого в команде резервируется необходимое место (с учетом типа имени!), а в объектном файле делается отметка об этом. При объединении нескольких объектных файлов редактор связей просматривает такие отметки и, используя доступную ему информацию о глобальных именах, подставляет необходимые адреса или значения.

Директивы NAME и END. Директива NAME (наименовать) с форматом

```
NAME     <имя модуля>
```

присваивает внутреннее *имя* объектному модулю, генерируемому ассемблером. Имена модулей необходимы для того, чтобы программист мог обращаться к модулям и определять требуемый ему порядок их следования. Директива

NAME может появиться в исходном модуле только 1 раз и сама не отмечается.

Директива END (окончить, конец) имеет следующий формат:

END [*<пусковой адрес>*]

В каждом исходном модуле может быть только одна директива END, причем она должна находиться в последней строке.

Пусковой адрес представляет собой метку той команды, с которой должно начаться выполнение программы. Если поле операнда в директиве END пустое, считается, что данный модуль не является главным (основным). При объединении нескольких модулей только в одном из них должен быть указан пусковой адрес — этот модуль становится главным.

Объединение логических сегментов. Предположим, что разрабатываемая программа состоит из двух модулей (MOD_1 и MOD_2) и что каждый модуль содержит логические сегменты, соответствующие областям кода, данных и стека. Назовем эти сегменты для определенности CODE_1, DATA_1 и STACK_1 для модуля MOD_1 и CODE_2, DATA_2 и STACK_2 для модуля MOD_2. Если

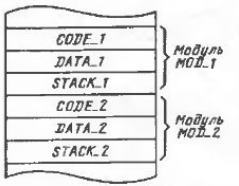


Рис. 3.3. Необъединенные логические сегменты двух модулей

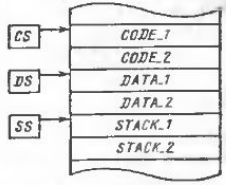


Рис. 3.4. Целесообразное объединение логических сегментов

два объектных модуля объединяются, получающаяся программа размещается в памяти так, как показано на рис. 3.3.

Проблема при таком распределении памяти заключается в том, что соответствующие логические сегменты обоих модулей не адресуются при одном и том же содержимом сегментного регистра. Следовательно, при каждой передаче управления между модулями необходимо модифицировать содержимое регистра CS. Доступ к двум областям данных организуется либо путем изменения содержимого регистра DS, либо путем адресации одной области через DS, а другой — через регистр ES (но здесь потребуются операторы замены сегмента). Наконец, самая неприятная ситуация связана с тем, что программе нужен один стек, и следовательно, одна из областей стека оказывается ненужной, а размер другой должен соответствовать требованиям к стеку обоих модулей.

Если объединенный размер сегментов кода меньше размера физического сегмента 64K байт, то сегменты CODE_1 и CODE_2 лучше объединить так, чтобы к содержащимся в них командам можно было обращаться при одном и том же содержимом регистра CS. При таком объединении межмодульные передачи управления потребуют модификации только содержимого программного счетчика PC, а это позволит избежать межсегментных команд JMP и CALL. Аналогичные соображения свидетельствуют о пользе такого объединения областей данных, чтобы к ним можно было обращаться через сегментный регистр DS. Целесообразно также объединить и сегменты стеков STACK_1 и STACK_2. Желательное объединение логических сегментов двух модулей показано на рис. 3.4.

Необходимая информация о способах объединения логических сегментов модулей MOD_1 и MOD_2 содержится в директивах SEGMENT. Общий формат пары директив SEGMENT/ENDS, который был приведен в § 3.3, имеет следующий вид:

```
<имя>   SEGMENT  [ <список атрибутов> ]
          .
          .
          .
<имя>   ENDS
```

До сих пор список атрибутов в директиве SEGMENT не рассматривался. Причина этого заключалась в том, что в приводимых простых примерах было достаточно атрибутов сегментов, принимаемых по умолчанию. Однако при объединении модулей атрибуты сегментов начинают играть важную роль. Поэтому приведем полный формат директив SEGMENT/ENDS:

```
<имя>   SEGMENT [ <тип выравнивания> [ <тип объединения>]
                [ 'имя класса' ]
          .
          .
          .
<имя>   ENDS
```

Прежде всего отметим, что объединяемые сегменты должны иметь одинаковые *имена*. Поэтому в рассматриваемом примере области кода в модулях MOD_1 и MOD_2 назовем общим именем P_CODE, области данных — именем P_DATA и области стека — именем P_STACK.

Как видно из формата директивы SEGMENT, в поле операнда ее содержатся три поля: *тип выравнивания, тип объединения и имя класса*. Эти поля, каждое из которых является необязательным, должны следовать, если они есть, в указанном порядке. Рассмотрим смысл всех атрибутов логических сегментов.

Тип объединения. Атрибут типа объединения показывает, каким образом данный логический сегмент должен объединяться с другим логическим сегментом, имеющим то же самое *имя*. Если для какого-либо сегмента этот атрибут не определен, то сегмент считается необъединяемым. Другими словами, если сегменту назначен атрибут *типа объединения* по умолчанию, то этот сегмент не будет объединяться ни с каким другим логическим сегментом, даже имеющим то же самое имя.

Тип объединения при необходимости следует указывать в первом определении сегмента. Его можно опускать в последующих директивах SEGMENT для данного сегмента, но при указании нельзя противоречить первой спецификации. Когда этот атрибут отсутствует в первой директиве SEGMENT некоторого сегмента, то принимается его значение по умолчанию, и любые другие спецификации в последующих директивах SEGMENT этого же сегмента вызовут сообщение об ошибке.

Атрибут PUBLIC применяется для логических сегментов, которые объединяются путем конкатенации, т. е. находятся в смежных (соседних) областях памяти. Когда некоторый сегмент объединяется с другими сегментами и все они имеют атрибут PUBLIC, смещения внутри сегмента корректируются с учетом суммарного размера уже объединенных сегментов. Если, например, объединяются два сегмента с атрибутом PUBLIC, имеющие размер 48 и 64, то смещения в первом сегменте не корректируются, а во втором — увеличиваются на 48, т. е. размер предыдущего логического сегмента. Такая коррекция смещений позволяет адресовать объединенные сегменты при одном и том же содержимом сегментного регистра.

Атрибут PUBLIC подходит для сегментов P_CODE и P_DATA рассматриваемого примера. Пусть в сегменте

P_DATA модуля MOD_1 находится 10-байтный массив DIGITS, а в сегменте P_DATA модуля MOD_2 определены четыре переменные типа BYTE. Такие логические сегменты с типом объединения PUBLIC следует закодировать следующим образом:

```
                (в модуле MOD_1)
P_DATA SEGMENT PUBLIC
        DIGITS      DB      0, 1, 2, 3, 4, 5, 6, 7, 8, 9
P_DATA ENDS

                (в модуле MOD_2)
P_DATA SEGMENT PUBLIC
        BVAL_0      DB      0
        BVAL_1      DB      ?
        BVAL_2      DB      20
        BVAL_3      DB      ?
P_DATA ENDS
```

Пусть при объединении первым следует модуль MOD_1. Тогда редактор связей скорректирует смещения переменных BVAL_0, BVAL_1, BVAL_2 и BVAL_3 так, чтобы учесть предыдущую 10-байтную область. Поэтому смещение переменной BVAL_0 будет равно 10, хотя в модуле MOD_2 оно было равно нулю. Аналогично корректируются и смещения других переменных в MOD_2. Объединение модулей MOD_1 и MOD_2 показано на рис. 3.5. Как видно

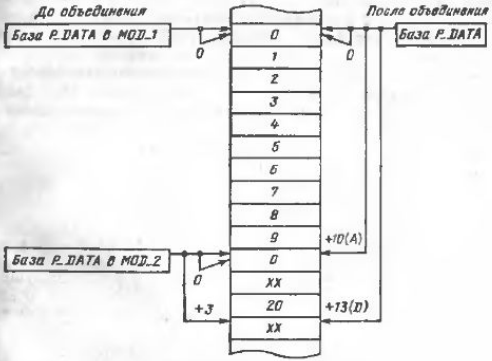


Рис. 3.5. Объединение сегментов P_DATA

из этого рисунка, после объединения сегментов переменные из обоих модулей адресуются от одного базового значения. Для обоих модулей имя P_DATA представляет собой базу объединенных логических сегментов. Если значение этого имени загрузить в сегментный регистр DS, его можно использовать для адресации DIGITS, BVAL_0, BVAL_1, BVAL_2 и BVAL_3.

Тип объединения STACK применяется для логических сегментов, которые во время выполнения программы будут частью стека. Сегменты с атрибутом STACK необходимо объединить так, чтобы размер полученной области памяти был равен сумме размеров объединяемых сегментов. Смещения внутри каждого из объединяемых сегментов с атрибутом STACK корректируются таким образом, чтобы последний байт (байт с наибольшим адресом) в каждом сегменте совпадал с последним байтом объединенной области. Другими словами, вместо того, чтобы один сегмент начинался с того места, где кончается предыдущий, все сегменты начинаются с одного и того же базового адреса. Стековые сегменты перекрываются в области памяти с большими адресами. Объединение осуществляется так, чтобы вершина пустого стека (начальное содержимое указателя стека SP, определяемое директивой LABEL) была первой ячейкой после области стека. Напомним, что первой стековой операцией обязательно должна быть операция включения в стек (PUSH), а она производит декремент SP.

Пусть в модуле MOD_1 определен сегмент с атрибутом STACK размером в 48 слов и вершина стека обозначена именем STACK_TOP. Кодирование такого сегмента имеет следующий вид:

```
                    (в модуле MOD_1)
P_STACK     SEGMENT STACK
            DW      48 DUP (?)
STACK_TOP   LABEL   WORD
P_STACK     ENDS
```

Далее предположим, что в модуле MOD_2 для стека требуются 24 слова. Если в этом случае вершину стека наименовать TOP STACK, то получается такое кодирование:

```
                    (в модуле MOD_2)
P_STACK     SEGMENT STACK
            DW      24 DUP (?)
TOP_ STACK  LABEL   WORD
P_STACK ENDS
```


При объединении сегментов (рис. 3.6) образуется область из 72 слов, причем смещения STACK_TOP и TOP_STACK будут скорректированы так, чтобы они относились к первой ячейке всей области стека.

До сих пор в примерах не было привязки к физическим адресам памяти. Но иногда требуется точно определить, в

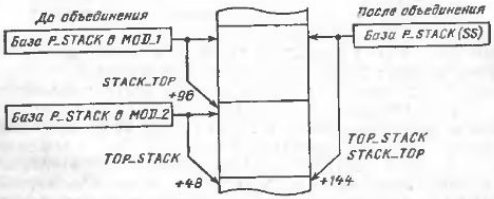


Рис. 3.6. Объединение сегментов P_STACK

каком месте физической памяти должен быть размещен некоторый сегмент. Для этой цели предусмотрен *тип объединения* AT. Конструкция AT <*выражение*> позволяет программисту задать начальный адрес логического сегмента, т.е. фиксировать сегмент в нужной области памяти. Значение *выражения* есть номер параграфа, который равен базовому значению, необходимому для доступа к сегменту. Чтобы задать конкретное смещение переменной или метки от начала сегмента, применяется директива ORG. В примере

```
ZERO    SEGMENT          AT 0
        ORG     CH
VECT    LABEL   DWORD
ZERO    ENDS
```

определена переменная VECT типа DWORD, значение которой находится в ячейке 000C физической памяти (это вектор команды INT 3). Отметим, что для определения VECT использована директива LABEL, а не директива DD. Следовательно, это фрагмент программы не распределяет память.

Тип объединения COMMON (общий) означает, что данный сегмент разделяет одинаковые ячейки памяти со всеми другими сегментами, имеющими такие же имена, из

других модулей. Это приводит к тому, что различные метки и имена переменных из всех модулей, имеющих одинаково наименованные сегменты с атрибутом COMMON, будут относиться к одним и тем же адресам. Такая ситуация напоминает действие атрибута STACK, но без суммирования отдельных сегментов.

Наконец, атрибут MEMORY действует аналогично атрибуту COMMON, однако в памяти сегменты с данным типом объединения размещаются после всех других сегментов. В объединяемых модулях должен быть один сегмент с атрибутом MEMORY, так как редактор связей учитывает атрибут MEMORY только у первого сегмента. Если, например, SEG_M есть имя первого сегмента с атрибутом MEMORY, который встречает редактор связей, то этот сегмент и другие сегменты с именем SEG_M из остальных модулей будут размещены после всех сегментов (по старшим адресам памяти). Другие сегменты с атрибутом MEMORY будут считаться имеющими тип объединения COMMON, и редактор связей выдает предупреждающее сообщение.

В заключение отметим, что ключевые слова ассемблера STACK и MEMORY в директивах SEGMENT допускается использовать в качестве имен, определяемых программистом.

Тип выравнивания. Данный атрибут определяет границу, на которой должен быть размещен логический сегмент. Ассемблер использует его для формирования в объектном файле служебной информации, которая потребуется редактору связей для выравнивания сегментов. Когда программа размещается в памяти и относительные адреса заменяются на абсолютные, сегменты различной длины будут заканчиваться на разных ячейках. Если следующий сегмент размещается с границы параграфа (без всякого сдвига), то все относительные адреса, которые были определены в исходном модуле для этого сегмента, оказываются правильными абсолютными смещениями для данного параграфа. Но может оказаться, что между концом предыдущего сегмента и началом данного сегмента находятся неиспользуемые байты, так как параграфы следуют через 16 байт. Число таких неиспользуемых байт варьируется от 0 до 15. Однако в ассемблере предусмотрены средства, обеспечивающие более плотное размещение сегментов.

Тип выравнивания PARA показывает, что логический

сегмент должен начинаться на границе параграфа, т. е. на адресе, кратном 16. Этот атрибут принимается и по умолчанию без явной спецификации *типа выравнивания*. Очевидно, что здесь возможна потеря 0—15 байт памяти.

Тип выравнивания WORD требует, чтобы сегмент начинался по четному адресу, а тип BYTE означает, что сегмент может начинаться по любому адресу. Последний атрибут соответствует наиболее плотной упаковке логических сегментов, так как между ними не будет никаких промежутков. Следует помнить, однако, что МП К1810 обращается к словам данных с четными адресами за один, а с нечетными — за два цикла шины. Поэтому *тип выравнивания* BYTE не рекомендуется использовать для областей данных.

Атрибут PAGE (страница) соответствует адресу границы, имеющему 16-ричное представление XXX00. Наконец, атрибут INPAGE (в странице) означает, что весь сегмент должен быть размещен в пределах одной страницы и, следовательно, его размер не может превышать 256 байт. Атрибуты PAGE и INPAGE применяются сравнительно редко.

Имя класса. Когда логическому сегменту назначается атрибут *имя класса,* то программа размещения собирает вместе все области с одинаковыми *именами классов.* Пусть, например, программа состоит из областей кода, констант, данных и стека, причем каждая область образована объединением логических сегментов с именами P_CODE, P_CONST, P_DATA и P_STACK соответственно. Если желательно разместить код и константы в постоянном запоминающем устройстве, то можно указать, что области P_CODE и P_CONST следуют друг за другом, введя для соответствующих сегментов общее *имя класса,* например 'ROM'. Аналогичным образом возможно собрать области P_DATA и P_STACK, назначив этим сегментом одно и то же *имя класса,* например 'RAM'.

Отметим, что атрибут *имя класса* просто указывает, что некоторые (уже объединенные) области должны быть размещены друг за другом в физической памяти. В отличие от атрибута *тип объединения* данный атрибут не обеспечивает адресацию областей при одном и том же содержимом сегментного регистра.

Группы. Предположим, что желательно адресовать различные области при одном и том же содержимом сегментного регистра. Например, размеры областей данных и сте-

ка позволяют разместить их в одном физическом сегменте памяти. Если объединить эти области таким образом, чтобы их можно было адресовать с одним базовым значением, то в регистры DS и SS загружались бы одинаковые значения, а в качестве указателей переменных и элементов в стеке использовались бы только смещения от общей базы. Именно такое объединение различных программных областей реализуется с помощью групп.

Группа — это множество программных областей, объединенных так, что они разделяют (или коллективно используют) общую базу. Отдельные области, например P_DATA и P_STACK, в группе являются смежными. По существу группа представляет собой «объединение объединений», так как группа образована из отдельных областей, которые, в свою очередь, построены из отдельных логических сегментов.

Информация, необходимая ассемблеру для объединения в группу нескольких областей, содержится в директиве GROUP (группа, сгруппировать), которая имеет следующий формат:

<имя группы> GROUP *<имя сегмента>* ,[*<имя сегмента>*]...

Здесь *имена сегментов* относятся к объединенным логическим сегментам, которые сами объединяются в группу. В дальнейшем *имя группы* можно использовать так же, как и имя сегмента.

Поясним понятие группы на простом примере. Предположим, что программа использует три различные области данных: константы в сегментах P_CONST с типом объединения PUBLIC, изменяющиеся данные (переменные) в сегментах P_DATA с типом объединения PUBLIC и стек, образованный из сегментов P_STACK с типом объединения STACK. Пусть размеры всех этих областей таковы, что все они могут быть размещены в одном физическом сегменте.

Если не использовать группы, то каждая из областей P_CONST, P_DATA и P_STACK будет иметь свою индивидуальную базу. Обращение к данным не представляет трудностей, так как для адресации области P_DATA можно воспользоваться регистром DS, области P_CONST — регистром ES, а области P_STACK — регистром SS, как это показано на рис. 3.7. Но для того, чтобы запомнить адрес переменной или передать адрес процедуре, потребуется два слова: база и смещение (здесь под переменной

несколько более широко понимается единица программных данных, наименованная или нет, которая находится в областях данных или стека).

Предположим теперь, что области P_CONST, P_DATA и P_STACK разделяют общую базу (рис. 3.8). Такой прием обеспечивает два положительных качества. Во-первых, в

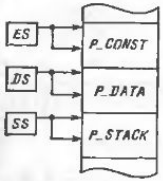


Рис. 3.7. Три различные области данных

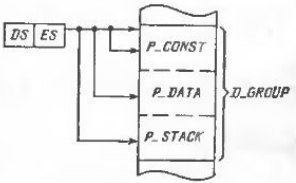


Рис. 3.8. Группа, состоящая из трех областей

сегментные регистры DS и SS можно загрузить общую базу и использовать любой режим адресации (с выбором сегментного регистра по умолчанию) для доступа ко всем трем областям. Во-вторых, для указания адреса переменной достаточно только смещения с учетом того, что базой всегда является содержимое регистра DS или SS.

Такое удобное объединение областей достигается следующей директивой GROUP:

D_GROUP GROUP P_CONST, P_DATA, P_STACK

Напомним, что при объединении логических сегментов с одним и тем же именем это же самое имя обозначает базу объединенных сегментов. Так же общую базу всех областей в группе обозначает *имя группы*, в данном случае D_GROUP. Конечно, база группы и база сегмента в группе в общем случае не совпадают. Поэтому при организации групп сегментные регистры необходимо инициализировать на значения *имен групп*, а не имен сегментов. Кроме того, в директивах ASSUME также должны фигурировать имена групп для трех сегментных регистров, которые будут содержать базу группы.

Смещения переменной или метки относительно их сегмента и относительно их группы являются различными, так как они отсчитываются от различных базовых значений.

Оператор OFFSET (см. § 3.5) всегда возвращает смещения переменной или метки относительно их сегмента. Однако при наличии групп необходимы смещения относительно базы группы. Для получения такого смещения перед именами, к которым производится обращение, указывается оператор группы. Он представляет собой имя группы с последующим двоеточием и находится перед именем переменной или метки. Пусть, например, VAL есть переменная в сегменте P_DATA, который находится в группе

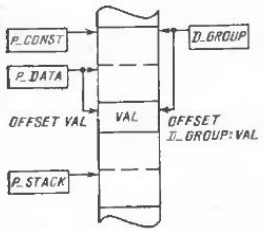


Рис. 3.9. Базы и смещения в группе

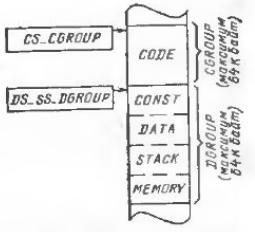


Рис. 3.10. Структура памяти в модели

D-GROUP. Тогда значением OFFSET VAL является смещение VAL от базы сегмента P_DATA, а значением OFFSET D_GROUP : VAL — смещение VAL от базы группы. На рис. 3.9 иллюстрируются различия между базами сегмента и группы, а также между смещениями переменной VAL относительно базовых значений сегмента и группы.

Пример программы. В заключение приведем пример простой программы, который иллюстрирует применение глобальных и внешних переменных, объединения сегментов, групп и процедур. Программа состоит из модулей MOD_1 и MOD_2 и вычисляет 5-элементный массив средних значений пяти пар чисел. Модуль MOD_1 является основным.

Программа 30. Вычисление массива средних
(Модуль MOD_1 основной)

```
NAME    MOD_1
DGROUP  GROUP   P_DATA, P_STACK
        ASSUME  CS:P_CODE, DS:D_GROUP
```


Продолжение

```
P_DATA  SEGMENT WORD PUBLIC
  MEANS     DW      5DUP (0)          ; Массив сред-
  EXTRN     PAIRS:WORD, COUNT:ABS     ;  них
P_DATA  ENDS
P_STACK SEGMENT WORD STACK
        DW      6 DUP (?)
  STK_TOP   LABEL WORD
P_STACK ENDS
P_CODE  SEGMENT BYTE PUBLIC
  EXTRN     AVRG:NEAR
  BEGIN:    MOV     AX, D_GROUP       ; Инициализа-
            MOV     DS, AX            ;  ция сег-
            MOV     SS, AX            ;  ментных
                                      ;  регистров
            MOV     SP, OFFSET D_GROUP:STK_TOP
            MOV     CX, COUNT         ; Счетчик пар
            MOV     SI, 0             ; Индексные ре-
                                      ;  гистры
            MOV     DI, SI
  NEXT:     MOV     AX, PAIRS [SI]
            MOV     BX, PAIRS [SI+2]
            CALL    AVRG              ; Усреднить
            MOV     MEANS [DI], DX    ; Запомнить
            ADD     SI, 4             ; Продвинуть
            ADD     DI, 2             ;  указатели
            LOOP    NEXT              ; Зациклить
P_CODE  ENDS
END BEGIN
```

Модуль MOD_2

```
NAME    MOD_2
D_GROUP GROUP P_DATA, P_STACK
        ASSUME CS:P_CODE, DS:D_GROUP
P_DATA  SEGMENT         WORD PUBLIC
  PAIRS     DW      200, 80           ; Массив
            DW      124, 141          ; из 5 пар
            DW      65, 1408          ;  чисел
            DW      1831, 7
            DW      39, 2508
  COUNT     EQV     5                 ; Число пар
```

```
P_DATA    ENDS
P_STACK   SEGMENT WORD STACK
                  DW      2 DUP (?)
P_STACK   ENDS
P_CODE    SEGMENT BYTE PUBLIC
   PUBLIC         AVRG
   AVRG           PROC NEAR
                  MOV DX,AX
                  ADD DX,BX
                  RCR DX,1
      AVRG        ENDP
   P_CODE    ENDS
END
```

Проанализируем сначала структуру программы в терминах логических сегментов и групп. Директивы SEGMENT/ENDS и D_GROUP одинаковы в обоих модулях и имеют следующий вид:

```
D_GROUP  GROUP P_DATA, P_STACK
P_DATA   SEGMENT WORD PUBLIC
         .
         .
         .
P_DATA   ENDS
P_STACK  SEGMENT WORD STACK
         .
         .
         .
P_STACK  ENDS
P_CODE   SEGMENT BYTE PUBPIC
         .
         .
         .
P_CODE   ENDS
```

В обоих модулях имеются:

сегмент данных P_DATA с типом объединения PUBLIC и выравненный на границе слова;

сегмент стека P_STACK с типом объединения STACK и выравненный на границе слова;

сегмент кода P_CODE с типом объединения PUBLIC и типом выравнивания BYTE (без выравнивания).

Директива GROUP сообщает, что области данных и стека объединяются в группу с именем D_GROUP. Следовательно, содержимое сегментов P_DATA и P_STACK адресуется с использованием смещений относительно общего базового значения.

Директивы ASSUME в обоих модулях одинаковы:

```
ASSUME  CS:P_CODE, DS:D_GROUP
```

Поскольку области данных объединены в группу, регистр DS содержит указатель базы группы, а не области данных. Переменные адресуются смещениями относительно базы группы, поэтому указыаать ASSUME DS : P_DATA было бы неправильным. Область кода не включена ни в какую группу, поэтому CS содержит базу P_CODE, что отражено в директиве ASSUME CS : P_CODE.

Фрагмент инициализации сегментных регистров и SP в модуле MOD_1 начинается с метки BEGIN и содержит четыре команды. В регистры DS и SS загружается база группы, а в указатель стека SP — смещение вершины стека относительно базы группы. Инициализацию SP реализует команда MOV SP, OFFSET D_GROUP : STK_TOP. Отметим, что указание в этой команде операнда OFFSET STK_TOP было бы ошибочным, так как тогда SP содержал бы смещение вершины пустого стека относительно базы области стека P_STACK, а не базы группы.

Собственно вычислительные действия рассматриваемой программы довольно просты. В модуле MOD_2 сегмент данных P_DATA содержит массив из пяти пар переменных типа WORD, определенных с помощью директив DW. Переменная PAIRS относится к первому из 10 слов. Директива EQU определяет COUNT как константу, т. е. именованное число. Так как к именам PAIRS и COUNT требуются обращения извне модуля MOD_2, они объявлены глобальными директивой PUBLIC.

Сегмент P_DATA в модуле MOD_1 просто резервирует пять слов, в которые будут помещаться средние значения пар чисел PAIRS. Здесь имена PAIRS и COUNT объявлены как внешние. Запись директивы

```
EXTRN  PAIRS:WORD, COUNT:ABS
```

внутри пары SEGMENT/ENDS сегмента P_DATA сообщает ассемблеру, что переменная PAIRS определена в другом сегменте с именем P_DATA, который будет объединяться с данным сегментом. Другими словами, из этой директивы EXTRN ассемблер узнает, что переменная PAIRS находится в области P_DATA. Константа COUNT не имеет адреса в формате база:смещение, поэтому ее можно указать директивой EXTRN в любом месте модуля.

Массив MEANS содержит пять слов. Программа нахо-

дит средние пять пар чисел PAIRS и запоминает их в последовательных словах массива MEANS.

Фактическое вычисление средних осуществляет процедура AVRG типа NEAR, поэтому при ее вызове в стеке запоминается только содержимое PC. Зная об этом, ассемблер сформирует правильную комаиду возврата RET. Отметим, что процедура AVRG в модуле MOD_2 объявлена глобальной, так как она вызывается из модуля MOD_1. В связи с этим в сегменте P_CODE модуля MOD_1 имеется директива EXTRN AVRG : NEAR. Она сообщает, что AVRG является меткой типа NEAR, которой в данном случае является точка входа процедуры. Вновь подчеркнем размещение директивы EXTRN внутри пары SEGMENT/ENDS сегмента P_CODE — это сообщение ассемблеру о том, что AVRG принадлежит сегменту P_CODE.

Собственно программа вычисления средних находится в сегменте P_CODE модуля MOD_1 и особых пояснений не требует. Регистр CX выполняет функцию счетчика циклов, а регистры SI и DI используются для индексирования. Регистр SI содержит индекс пары чисел из массива PAIRS, поэтому при прохождении по циклу осуществляется инкремент SI на 4. Выражение PAIRS[SI] адресует первое число пары, а выражение PAIRS[SI+2] — второе. Регистр DI содержит индекс элемента маснва MEANS и увеличивается на 2 при проходе по циклу.

До вызова процедуры AVRG в регистры AX и BX загружаются ее параметры — два усредняемых числа. Процедура AVRG является функцией и возвращает среднее двух чисел в регистре DX. Результат усреднения из регистра DX передается в текущий элемент массива MEANS.

Модели вычислений. Директивы ассемблера обеспечивают разнообразные способы объединения сегментов и образования групп. Правило, которое устанавливает, каким образом объединяются сегменты, и, возможно, как они включаются в группы, называется моделью вычислений. В общем, модели вычислений рассчитаны на языки высокого уровня и выбираются с помощью управляющих конструкций языка. Однако выбранная модель вычислений влияет иа содержание директив SEGMENT и GROUP языка ассемблера, а также на соглашения о взаимодействиях процедур, в частности на передачи в качестве параметров адресов — полных (в формате база:смещение) или коротких (только смещения).

Наиболее простая модель вычислений, которая называется SMALL (малая), характеризуется следующей структурой: код в одном физическом сегменте, а данные и стек — в другом. Следовательно, ее можно использовать для программ, которые требуют не более 64K байт для кода и 64K байт для объединенных данных и стека. Основное достоинство модели SMALL заключается в том, что все указатели являются только смещениями, а содержимое сегментных регистров CS, DS и SS фиксировано, причем регистры DS и SS содержат одно и то же значение. В команде передачи управления модифицируется только программный счетчик PC, а для адресации переменной или элемента в стеке достаточно 16-битного смещения. Таким образом, модель SMALL обеспечнвает наиболее компактные по длине и быстро выполияемые программы.

Сегменты кода, даиных и стека и модели SMALL наименованы соответственно CODE, DATA и STACK. Для данных имеются еще два сегмеита CONST и MEMORY, но они редко применяются в ассемблерных программах.

Программа в модели SMALL разделена на две группы: сегмент кода является единственным сегментом в группе CGROUP с базой в регистре CS, а остальные сегменты входят в группу DGROUP, база которой находится в регистрах DS и SS (рис. 3.10).

Если программист знает, что его программа удовлетворяет требованиям модели SMALL, он может воспользоваться так называемым шаблоном исходного ассемблерного модуля, который имеет следующую структуру:

```
NAME      <имя модуля>
CGROUP    GROUP CODE
DGROUP    GROUP CONST, DATA, STACK MEMORY
ASSUME    CS:CGROUP, DS:DGROUP, SS:DGROUP
CONST     SEGMENT PUBPIC 'CONST'

        <Здесь размещаются константы>

CONST     ENDS
DATA      SEGMENT PUBLIC 'DATA'
          EXTRN  <внешние переменные>

        <Здесь определяются данные>

DATA      ENDS
STACK     SEGMENT STACK 'STACK'

        <Директива DW резервирует стек>

STACK     ENDS
MEMORY    SEGMENT MEMORY 'MEMORY'
```

```
<Специальный сегмент данных>
MEMORY      ENDS
CODE        SEGMENT PUBLIC 'CODE'
            EXTRN   <внешние переменные и метки>

<Командные операторы>
CODE        ENDS
END         <пусковой адрес>    ; Только в основном модуле
```

Прописными буквами в шаблоне даны все директивы, которые, необходимо скопировать в исходном модуле. Предложения в угловых скобках являются поясняющими и в реальном модуле заменяются конструкциями языка ассемблера. Шаблон содержит директивы SEGMENT/ENDS для всех сегментов, которые применяются в языке высокого уровня. Директивы SEGMENT для сегментов CODE, DATA, STACK, а также директивы GROUP и ASSUME показаны так, как они должны появиться в исходном модуле. Атрибуты выравнивания в шаблоне приняты по умолчанию PARA, но программист может задать свои типы выравниваня.

При необходимости программист может определить дополнительные сегменты, а ненужные сегменты опустить. Если удаляется сегмент, входящий в группу, его имя нужно убрать из списка сегментов в директиве GROUP. Пусть, например, в исходном модуле не требуются сегменты CONST и MEMORY. Тогда директива GROUP для группы DGROUP примет следующий вид:

```
DGROUP  GROUP  DATA, STACK
```

При использовании любой модели вычислений важно правильно инициализировать сегментные регистры и SP. В модели SMALL регистр CS содержит базу CGROUP, поэтому в директиве END основного модуля должна находиться метка, содержащаяся в сегменте CODE и соответствующая фрагменту инициализации регистров. В регистры DS и SS необходимо загрузить базу DGROUP. Инициализацию регистров выполняют следующие команды:

```
CODE SEGMENT PUBLIC 'CODE'
     BEGIN: MOV AX, DGROUP
            MOV DS, AX
            MOV SS, AX
            MOV SP, OFFSET DGROUP:STK_TOP
            .
            .
            .
CODE ENDS
END BEGIN
```


Поскольку сегментные регистры содержат базы групп, в операторах OFFSET модулей модели SMALL необходимо всегда указывать оператор группы. Это требование учтено при инициализации указателя стека — в него загружается смещение относительно базы группы DGROUP, а не относительно базы сегмента стека (для чего потребовался бы оператор OFFSET STK_TOP). Смещения переменных всегда берутся относительно базы DGROUP, а смещения меток — относительно базы CGROUP.

Оператор группы необходим также в тех случаях, когда в поле операнда директивы распределения данных фигурируют имена переменных или меток. Если, например, имя VAL_1 представляет собой переменную в сегменте DATA, то правильный относительный адрес ее дает директива DW DGROUP : VAL_1, а не DW VAL_1.

Модель SMALL предполагает неизменное содержимое регистра CS. Следовательно, все метки и процедуры должны иметь тип NEAR, а при передачах управления модифицируется содержимое только программного счетчика PC.

Все адреса (указатели) в модели SMALL представляются одним словом. Адрес переменной в области данных или в стеке есть смещение относительно базы DGROUP, а адрес метки в области кода — смещение относительно базы CGROUP. Следовательно, передаваемый процедуре указатель всегда состоит из одного слова, являясь смещением относительно DGROUP или CGROUP.

В некоторых применениях невозможно удовлетворить ограничения модели SMALL и приходится переходить к более сложным и менее эффективным моделям вычислений. Но иногда приемлемым решением является некоторое расширение модели SMALL с сохранением максимальной эффективности большей части кода. Пусть, например, при разработке программы размер кода не превышает 64K байт, но из-за большой таблицы данных в 32K байт размер объединенной области данных и стека оказывается больше 64K байт. Если таблицу вынести из DGROUP, программа удовлетворяет ограничениям модели SMALL.

В ассемблерном модуле программисту разрешено определять дополнительные сегменты данных вне DGROUP, поэтому большую таблицу данных можно разместить в отдельном сегменте, например TABLE. К элементам TABLE в ассемблерном модуле возможно обращаться либо путем изменения содержимого регистра DS, либо адресацией через регистр ES. Если к элементам таблицы должна обра-

шаться программа на языке высокого уровня, то требующийся элемент следует переслать в буферную область внутри DGROUP с помощью специальной процедуры, написанной на языке ассемблера.

3.8. Примеры ассемблерных программ

Чтобы закрепить навыки программирования на языке ассемблера, в данном параграфе приведены ассемблерные программы, несколько более сложные по сравнению с программами в гл. 2. Рекомендуется внимательно разобраться во всех имеющихся программах и при необходимости составить их подробные схемы.

1. При вводе числовых 16-ричных данных с терминала каждая цифра представляется в коде ASCII. Цифрам 0—9 соответствуют байты с 16-ричными кодами 30—39, а «буквенные» цифры A—F имеют коды 41—46. После ввода данных возникает задача преобразования двух символов кода в 16-ричный эквивалент. Приводимая ниже подпрограмма CONV предназначена для решения этой задачи в предположении, что регистр SI адресует символы в памяти (первый символ — старший) и что полученный результат возвращается в регистр AL. Если преобразуемый код символа выходит за допустимый диапазон (30—39, 41—46), подпрограмма возвращается с установленным флажком ZF. Проверку диапазона и приведение кода символа в тетраде осуществляет вложенная подпрограмма TRANS. Она получает код символа в регистре и возвращает соответствующую тетраду в младших битах этого же регистра. Если код символа находится вне допустимого диапазона, подпрограмма TRANS возвращается с установленным флажком ZF.

Программа 31. Преобразование двух символов в 16-ричный эквивалент

```
CONV    PROC    NEAR
        PUSH    CX                  ; Запомнить (CX)
        MOV     AL, BYTE PTR[SI]    ; Первый символ
                                    ;  в AL
        CALL    TRANS               ; Преобразовать его
        JZ      ERROR               ; Ошибка?
        MOV     CL,4                ; Нет, сдвинуть
        SAL     AL,CL               ;  в старшую
                                    ;  тетраду
        MOV     AH,AL               ; Передать в AH
        MOV     AL,BYTE PTR [SI+1]  ; Второй символ
        CALL    TRANS               ; Преобразовать
        JZ      ERROR               ; Ошибка?
```


Продолжение

```
        OR      AL,AN               ; Объединить тет-
                                    ;  рады
        OR      AH,0FFH             ; Сбросить флажок
                                    ;  ZF
        POP     CX                  ; Восстановить CX
        RET                         ; Нормальный воз-
                                    ;  врат
ERROR:  RET                         ; Ошибка
CONV    ENDP
        ;
TRANS   PROC    NEAR
        SUB     AL, 30H             ; Меньше кода 0?
        JL      FAULT               ; Да, ошибка
        CMP     AL, 0AH             ; Больше цифры
                                    ;  9?
        JL      DONE                ; Нет, возврат
        SUB     AL, 07H             ; Коррекция на A—
                                    ;  F
        CMP     AL, 10H             ; Больше цифры
                                    ;  F?
        JGE     FAULT               ; Да, ошибка
DONE:   RET                         ; Нормальный воз-
                                    ;  врат
FAULT:  XOR     AH, AH              ; Установить ZF
        RET                         ; Ошибка
TRANS   ENDP
```

2. Следующая программа показывает общую организацию обработки четырехуровневых прерываний с помощью очереди FIFO («первый пришел, первый ушел») [8]. Предположим, что прерывания в системе могут генерировать четыре периферийных устройства, запросы которых подаются в программируемый контроллер прерываний, который, в свою очередь, генерирует сигнал INT прерывания микропроцессора. При подтверждении прерывания в микропроцессор вводится тип прерывания, индентифицирующий прерывающее устройство. Типы прерываний представлены 16-ричными кодами 32—35. Напомним, что, реагируя на прерывание, МП включает в стек содержимое регистров флажков, CS и PC, а затем осуществляет косвенный межсегментный переход через таблицу указателей (векторов) прерываний. Таблица содержит полные указатели процедур обслуживания прерываний, а адрес используемого указателя определяется путем умножения типа прерывания на 4.

Достаточно подробная схема обработки прерываний показана на рис. 3.11. Первые операторные вершины особых пояснений не требуют, так как они представляют собой стандартную реакцию на прерывание. Далее проверяется

состояние двоичной переменной FLAG, которое показывает, чем был занят МП до возникновения прерывания. Если флаг установлен (FLAG=FFH), то МП обрабатывал предшествующее прерывание. В этом случае код идентификации прерывающего устройства помещается в очередь ожидающих обработки прерываний, а МП осуществляет возврат из прерываний. Когда флаг сброшен (FLAG==00H), МП выбирает для обработки ожидающий запрос

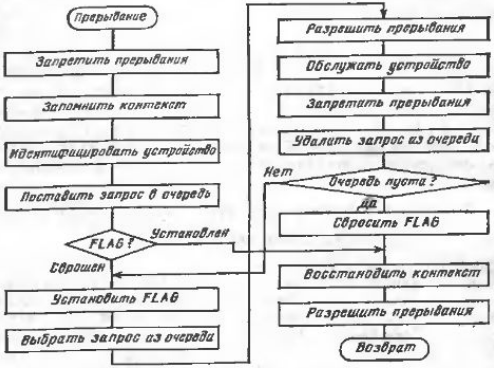


Рис. 3.11. Схема обработки прерываний

(код идентификации) из очереди FIFO и выполняет конкретные действия по обслуживанию соответствующего периферийного устройства (эти действия показаны на схеме операторной вершиной «Обслуживание устройства», а в программе не приведены, так как они определяются спецификой устройства). Отметим, что обработка происходит с разрешенными прерываниями, а по ее окоичании прерывания кратковременно запрещаются для выполнения служебных действий. Эти действия заключаются в удалении запроса из очереди и проверке ее состояния. Если в очереди аапросов нет, МП восстанавливает контекст и возвращается в прикладную программу, причем при извлечении из стека состояний флажков прерывания будут разрешены. Если

очередь не исчерпана, МП продолжает обработку ожидающих запросов прерываний.

Некоторый интерес представляет организация очереди FIFO. Это блок смежных ячеек памяти из 256 байт, который может размещаться в любой области адресуемой памяти. Указателем (внутрисегментным) текущего элемента очереди служит переменная HEAD. Циклическая (круговая) структура очереди достигается тем, что в инкремеите указателя участвует только его младший байт. Этим обеспечивается автоматический переход от наибольшего адреса блока очереди к наименьшему.

В программе принято следующее распределение регистров: AX — рабочий регистр, BX — указатель начала очереди, DI — код идентификации периферийиого устройства, DS содержит адрес сегмента, в котором находится очередь.

Содержимое этих регистров образует тот контекст, который необходимо сохранять в стеке после восприятия запроса прерывания и восстанавливать после обслуживания запроса.

Программа 32. Обслуживание многоуровневых прерываний

```
                        NAME SERVICE

; Символические имена кодов идентификации
ID_32       EQU     0H          ; Коды идентификации
ID_33       EQU     02H         ;    устройств в соответ-
ID_34       EQU     04H         ;    ствии с их типами
ID_35       EQU     06H         ;    прерываний
ON          EQU     0FFH        ; Состояние флага — уста-
                                ;    новлен
; Таблица указателей прерываний
PNTRS       SEGMENT
            ORG     128H        ; Точное начало
TYPE_32     DD      DEV_32      ; Таблица указателей
TYPE_33     DD      DEV_33      ;    процедур прерыва-
TYPE_34     DD      DEV_34      ;    ний соответствую-
TYPE_35     DD      DEV_35      ;    щих устройств
PNTRS       ENDS
; Сегмент, содержащий очередь
DATA        SEGMENT WORD PUBLIC 'DATA'
QUEUE       DW 128 DUP(0)       ; Область очереди
FLAG        DB 2 DUP(0)         ; Флаг состояния МП
HEAD        DW ?                ; Указатель начала оче-
DATA        ENDS                ;    реди
; Собственно программа реакции на прерывание
CODE        SEGMENT PARA PUBLIC 'CODE'
            ASSUME CS:CODE, DS:DATA
```

```
                ; Процедура прерывания устройства 32
DEV_32    PROC      FAR
          PUSH      BX              ; Запомнить в стеке со-
          PUSH      AX              ;   держимое регистров
          PUSH      DI              ;   (контекст)
          PUSH      DS
LOAD_ID:  MOV       DI, ID_32       ; Код идентификации
          MOV       AX,SEG QUEUE    ; В DS сегментный адрес
          MOV       DS, AX          ;   очереди
          MOV       AX, HEAD        ; Указатель «головы» оче-
          MOV       BX,AX           ;   реди в регистре BX
          MOV       [BX],DI         ; Поставить запрос в оче-
                                    ;   редь
          ADD       AL, 2           ; Скорректировать указа-
                                    ;   тель
          MOV       HEAD, AX        ; Запомнить указатель
          MOV       AL,ON           ; Проверить и установить
          XCHG      AL,FLAG         ;   флаг состояния
          TEST      AL,ON
          JZ        CONT            ; Продолжить обработку
          POP       DS              ; Восстановить контекст,
          POP       DI              ;   запрос прерывания
          POP       AX              ;   находится в очереди
          POP       BX
          IRET                      ; Возврат из прерывания
DEV_32    ENDP

; Далее необходимо запрограммировать три
; аналогичные процедуры DEV_ 33, DEV_ 34 и
; DEV_ 35, которые отличаются от приведенной
; только (и только!) командой загрузки кода
; идентификации устройства. Эта команда в процедуре
; DEV_ 32 имеет метку LOAD_ID. В других
; процедурах на этом месте должны быть команды
; MOV DI, DEV_ 33, MOV DI, DEV_ 34 и
; MOV DI, DEV_35 соответственно

; Обслуживание прерывания устройства, код
; идентификации которого считывается из очереди
; в регистр DI

NEXT      PROC      FAR
CONT:     MOV       DI,[BX]         ; Код идентификации в DI
          STI                       ; Разрешить прерывания

; Здесь должны быть команды, осуществляющие
; конкретные действия по обслуживанию устройства
; в соответствии с кодом идентификации в
; регистре DI. Приводимые ниже
; команды являются заключительными для
; обслуживания

          CLI                       ; Запретить прерывания
          ADD       BL,2            ; Удалить запрос из очереди
          CMP       BX,HEAD         ; Очередь пуста?
          JNE       CONT            ; Нет, продолжить обработку
```



```
          XOR       AX, AX          ; Да, сбросить флаг обслужи-
          MOV       FLAG, AL        ;   вания
          POP       DS              ; Восстановить
          POP       DI              ;   контекст
          POP       AX
          POP       BX
          IRET                      ; Возврат в программу
NEXT      ENDP
CODE      ENDS
          END
```

3. Следующие две простые программы показывают возможности МП К1810 по управлению черно-белым символьным дисплеем [9]. На экран дисплея выводятся 25 строк текста по 80 символов в каждой, т. е. всего 2000 символов. Формат экрана, показанный на рис. 3.12, соответствует иден-

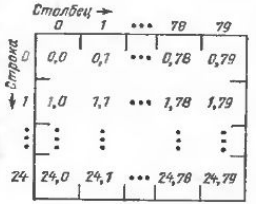


Рис. 3.12. Формат экрана 25×80

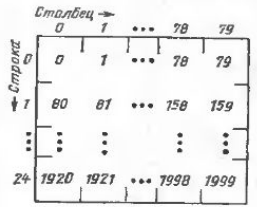
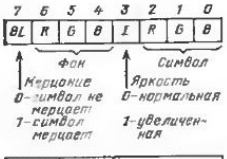


Рис. 3.13. Символьные позиции на экране

тификации символьных позиций номерами строк и столбцов. Символьные позиции можно также пронумеровать целыми числами от 0 до 1999 в соответствии с рис. 3.13; в этом случае для указания конкретной символьной позиции достаточно 11 бит.

Индицируемые символы представляются байтами, имеющими 16-ричные коды от 00 до FF. С каждым символом ассоциируется байт атрибутов, определяющих вид индикации символа. Атрибуты обеспечивают индикацию светлого символа на темном фоне (нормальное видеоизображение), темного символа на светлом фоне (обратное видеоизображение), вывод символа с повышенной яркостью, вывод мерцающего символа и подчеркнутого символа. Формат байта атрибутов приведен на рис. 3.14.

Выводимые на экран данные хранятся в оперативной памяти (видео-ЗУПВ) емкостью 4К байт, имеющей физический начальный адрес B0000. Если загрузить этот адрес в сегментный регистр, то адреса байт будут представлять собой смещения в диапазоне от 0 до 4К—1. Байт с четным адресом содержит код символа, а соседний байт с большим нечетным адресом — атрибуты данного символа. Таким образом, слово с четным адресом дает всю информацию для одной символьной позиции. Бит, идентифицирующий тип информации в байте, добавляется справа к 11-битному адресу (0-символ, 1-атрибуты).



Фон R G B	Символ R G B	Атрибут
0 0 0	0 0 0	Пустая позиция
1 1 1	1 1 1	Белый прямоугольник
0 0 0	1 1 1	Нормальное изображение
0 0 0	0 0 1	Нормальное изображение с подчеркиванием
1 1 1	0 0 0	Инверсное изображение

Рис. 3.14. Формат байта атрибутов

Приводимая ниже программа осуществляет вывод на экран всех символов. Для лучшей наглядности между соседними символами в строке введено по одному пробелу, а между соседними текстовыми строками — по одной пустой строке (строка из 80 пробелов).

Программа 33. Индикация символов

```
STACK   SEGMENT PARA STACK 'STACK'
        DB      256 DUP(0)      ; Определение стека
STACK   ENDS
;
CODE    SEGMENT PARA PUBLIC 'CODE'
  START PROC    FAR
; Стандартное начало программы
        ASSUME  CS:CODE
        PUSH    DS              ; Сохранить в стеке сегментный адрес и нулевое смещение
        MOV     AX,0            ;   PSP для возврата
        PUSH    AX
; Гашение (очистка) дисплея
        MOV     AX, 0B000H      ; Сегментный адрес
        MOV     ES,AX           ;   памяти дисплея в ES
        MOV     DI,0            ; Начальное смещение
```


Продолжение

```
        MOV     AL, ''          ; Символ пробела
        MOV     AH, 07H         ; Атрибуты нормальной индикации
        MOV     CX, 2000        ; Счетчик слов
        CLD                     ; Направление — вперед
REP     STOSW                   ; Погасить дисплей
; Вывод на экран 256 символов
        MOV     AL,0            ; Код символа в AL
        MOV     AH,0            ; Номер столбца в AH
        MOV     DI,160          ; Пропустить одну строку
; Основной программный цикл
AGAIN:  MOV     ES:[DI],AL      ; Передать символ в память
        ADD     DI,4            ; Перейти на следующую позицию
        ADD     AH,2            ; Следующий столбец
        CMP     AH,80           ; Конец строки?
        JB      LINE            ; Нет
        ADD     DI,160          ; Да, начать новую строку
        MOV     AH,0            ; Сбросить счетчик столбцов
LINE:   CMP     AL,255          ; Все символы?
        JE      FINISH          ; Да, закончить
        INC     AL              ; Нет, следующий символ
        JMP     AGAIN           ; Повторить основной цикл
; Индикация закончена, ожидать клавишного ввода
FINISH: MOV     AH,0
        INT     16H             ; Ожидать клавишного ввода
; При возврате необходимо погасить экран
        MOV     DI,0            ; Начальное смещение
        MOV     AL, ' '         ; Символ пробела
        MOV     AH, 07H         ; Атрибуты нормальной индикации
        MOV     CX, 2000        ; Счетчик слов
REP     STOSW                   ; Погасить экран
        RET                     ; Возврат в операционную
START   ENDP                    ;   систему
CODE    ENDS
        END     START
```

Программа оформлена как подпрограмма типа FAR. Объясняется это тем, что в среде операционной системы все прикладные программы считаются подпрограммами, которые вызываются операционной системой.

Первые три исполняемых оператора связаны с наличием так называемого префикса программного сегмента (PSP). Когда операционная система передает управление

Таблица 3.1. Префикс программного сегмента

Поле	Длина поля, байт	Описание поля
0	2	Команда INT 20H
2	2	Общий размер памяти (инкрементами по 16 байт)
5	5	Команда JMP типа FAR к управлению функциями операционной системы
5CH	16	Первый параметр в виде управляющего блока файла
5CH	16	Второй параметр в виде управляющего блока файла
80H	1	Длина параметра
81H	?	Параметр

программе, поля в PSP инициализированы в соответствии с табл. 3.1. Первые два байта PSP содержат команду INT 20H. Когда прикладная программа завершена, она должна выполнить данную комаиду для возврата управления операционной системе. Для этого приходится запомнить в стеке сегментный адрес PSP, передаваемый операционной системой в регистре DS. Требуемое действие осуществляет комаида PUSH DS. Следующие две команды (MOV. AX,0 и PUSH AX) включают в стек нулевое смещение. Благодаря этим действиям команда RET межсегментного возврата, заканчивающая программу, передает управление команде INI 20H, находящейся в PSP, а она осуществляет возврат в операционную систему.

Префикс программного сегмента позволяет также передать прикладной программе информацию в виде параметров. Параметры определяются в строке приказов, которая вызывает прикладную программу. Символы, образующие параметры, операционная система помещает в PSP со смещением 81. Длина цепочки параметров запоминается со смещением 80. Кроме того, параметры часто представляют собой имена файлов, поэтому они размещаются в PSP со смещениями 5C и 6C в виде управляющих блоков файлов.

Следующие семь команд осуществляют гашение (очистку) экрана путем инициализации сегментиого регистра ES на начало видео-ЗУПВ и записи во все символьные байты кода пробела. Такая инициализация ES требует, чтобы все дальнейшие обращения к видео-ЗУПВ осуществлялись через этот сегментный регистр. Отметим, что в регистр AH, который определяет все атрибутные байты, загружается

код 07. Как видно из рис. 3.14, такое значение атрибута соответствует нормальному изображению символов.

Следующие три команды инициализируют переменные основного цикла. Текущий код выводимого символа формируется в регистре AL, а текущий номер столбца находится в регистре AH. Регистр DI содержит текущее смещение в области видео-ЗУПВ. Первая символьная позиция индикации находится в первом столбце второй строки (позиция 80, ячейка 160 в видео-ЗУПВ).

Команда с меткой AGAIN, начинающая основной цикл, передает текущий код символа в видео-ЗУПВ. Байт атрибутов задавать не нужно, так как все символы индицируются в нормальном изображении, которое было определено при гашении экрана.

Команда ADD DI, 4 осуществляет переход на следующую символьную позицию. Инкремент на 4 объясняется тем, что для каждой символьной позиции в видео-ЗУПВ имеются 2 байта и в программе предусмотреи пропуск столбца между индицируемыми символами. Пропускаемый столбец всегда содержит код пробела, заданный при гашении экрана. При достижении конца текущей строки содержимое регистра DI увеличивается на 160 для перехода к очередной строке с пропуском следующей по порядку строки.

После достижения символьного кода 255 (или FF в 16-ричной системе) программа ожидает клавишный ввод, для чего предусмотрена команда INT 16H. Данная команда вызывает утилиту операционной системы, которая ожидает, когда пользователь нажмет любую клавишу. Когда это происходит, управление передается команде, находящейся за командой INT 16H. Как видно из текста программы, пять команд выполняют гашение экрана, а заключительная команда RET межсегментиого возврата выполняет возврат в операционную систему через стек и команду INT 20H в PSP.

Еще одна программа выводит на экран букву A с восемью различными атрибутами. Буква индицируется в центре экрана по вертикали, причем между соседними буквами имеются две пустые строки. Как и в предыдущей программе, при нажатии любой клавиши производится гашение экрана и возврат управления операционной системе.

После задания стека с глубиной 256 байт в сегменте данных с помощью директив DB определяются восемь атрибутов символов.

Программа 34. Показ атрибутов символов

```
;
STACK   SEGMENT     PARA STACK 'STACK'
        DB          255 DUP(0)      ;Определение стека
STACK   ENDS
;
DATA    SEGMENT PARA PUBLIG 'DATA'
  ATTRS DB          07H             ;Нормальное изображение
        DB          0FH             ;Нормальное с большей яркостью
        DB          8FH             ;Яркость и мерцание
        DB          87H             ;Нормальное с мерцанием
        DB          01H             ;Нормальное с подчеркиванием
        DB          70H             ;Инверсное изображение
        DB          0F0H            ;Инверсное с мерцанием
        DB          0F8H            ;Инверсное, яркое с мерцанием
  DATA  ENDS
;
CODE    SEGMENT     PARA PUBLIG 'CODE'
  START PROC        FAR
        ASSUME      CS:CODE
        PUSH        DS              ;Сохранить в стеке сегментный
        MOV         AX,0            ;адрес и нулевое
        PUSH        AX              ;смещение PSP для возврата
        MOV         AX,DATA         ;Инициализации регистра DS
        MOV         DS,AX
        ASSUME      DS:DATA
; Гашение (очистка) дисплея
        MOV         AX,0B000H       ;Сегментный адрес видео-ЗУПВ
        MOV         ES, AX          ;дисплея в регистре ES
        MOV         DI, 0           ;Начальное смещение
        MOV         AL,             ;Символ пробела
        MOV         AH, 07H         ;Атрибут нормального изображения
        MOV         CX, 2000        ;Счетчик слов
        CLD                         ;Направление — вперед
  REP   STOSW                       ;Погасить дисплей
; Вывод буквы А с различными атрибутами
        MOV         CX,8            ;Счетчик атрибутов
        MOV         DI,240          ;Начальное смещение в видео-
                                    ;ЗУПВ (центр второй строки)
```


Продолжение

```
        MOV         BX, OFFSET ATTRS ;Указатель атрибутов
        MOV         AL, 'A'         ;Индицируемый символ
; Основной цикл индикации
LOOPD:  MOV         AH, [BX]        ;Очередной байт атрибутов
        MOV         ES:[DI], AX     ;Передать символ и атрибут
        ADD         DI, 320         ;Сместиться на две строки
        INC         BX              ;Следующий байт атрибутов
        LOOP        LOOPD           ;Повторить с новым атрибутом
; Индикация закончена, ожидать клавишного ввода
        MOV         AH,0
        INT         16H             ;Ожидать клавишного ввода
; При возврате необходимо погасить экран
        MOV         DI, 0           ;Начальное смещение
        MOV         AL, ' '         ;Символ пробела
        MOV         AH, 07H         ;Нормальное изображение
        MOV         CX, 2000        ;Счетчик слов
REP     STOSW                       ;Погасить экран
        RET                         ;Возврат в операционную систему
START   ENDP
CODE    ENDS
        END         START
```

Собственно программа имеет стандартное начало (пролог), после которого в регистр DS загружается сегментный адрес данных. Затем, как и в предыдущей программе, осуществляется гашение экрана.

После инициализации переменных основного цикла в центральной части экрана по вертикали индицируется буква А с восемью атрибутами.

Заключительная часть данной программы полностью аналогична соответствующему фрагменту предыдущей программы.

4. Рассмотрим две программы, показывающие возможности МП К1810 по обработке двоичных наборов, а также иллюстрирующие его команды сдвига [10].

Пусть в памяти находится двоичный набор (или двоичная цепочка), выравненный на границе слова. Задача заключается в том, чтобы проверить, установить или сбросить определенный бит в этом наборе. Для подпрограммы,

решающей эту задачу, необходимы три параметра. Первым параметром является начальный адрес набора, и он передается подпрограмме в регистре BX. Второй параметр определяет порядковый номер выделяемого бита — он передается в регистре CX. Наконец, третий параметр представляет собой код функции, определяющий операцию с указанным байтом. Код функции передается в регистре DX и интерпретируется следующим образом:

1 — проверить бит и сообщить о его состоянии в младшем бите регистра AL,

2 — установить бит в 1,

3 — сбросить бит в 0.

Подпрограмма начинается с запоминания в стеке содержимого рабочего регистра SI, а также номера бита и кода функции в предположении, что они должны быть возвращены неизменными. После этого номер передается в регистр AX и путем сдвига вправо на 3 бита (деления на 8) преобразуется в относительный адрес байта, содержащего требуемый бит. Этот адрес передается в индексный регистр SI. Командой MOV AL, [BX] [SI] нужный байт загружается в регистр AL.

Затем команда AND с непосредственным операндом 00000111B образует в регистре CL позиционный номер выделяемого бита в байте. С помощью команды ROR AL,CL этот бит сдвигается в младший бит регистра AL. Заключительные команды осуществляют заданную кодом функции операцию над выделенным битом, при необходимости возвращают байт с потенциально модифицированным битом в набор и восстанавливают содержимое регистров из стека.

Программа 35. Операции с двоичным набором

```
ASEG    SEGMENT
        ASSUME CS:ASEG, DS:ASEG, ES:ASEG, SS:ASEG
BIT     PROC   FAR
        PUSH   SI                ; Запомнить рабочий регистр,
        PUSH   CX                ;    номер бита
        PUSH   DX                ;    и код функции
        MOV    AX, CX            ; Номер бита в AX
        SAR    AX, 1             ; Образовать относительный
        SAR    AX, 1             ;    адрес байта в наборе
        SAR    AX, 1
        MOV    SI, AX            ; Для индексирования в SI
        AND    CL, 00000111B     ; Номер бита в байте
        MOV    AL, [BX][SI]      ; Нужный байт в AL
        ROR    AL, CL            ; Бит справа в AL
        DEC    DX                ; Модифицировать код функции
        DEC    DX                ;    0=установить, 1=сбросить
```


Продолжение

```
        JS     NOCHG             ; Функция — проверить
        MOV    AH, AL            ; Сохранить байт
        OR     AL, 1             ; Установить бит в 1
        XOR    AL, DL            ; Выполнить операцию
        ROL    AL, CL            ; Восстановить исходный байт
        MOV    [BX][SI], AL      ; Передать его в набор
        MOV    AL,AH             ; Для проверки в AL
NOCHG:  AND    AX, 0001H         ; Подавить все ненужные биты
        POP    DX                ; Восстановить запомненные
        POP    CX                ;    регистры
        POP    SI
        RET                      ; Возврат
BIT     ENDP
ASEG    ENDS
        END
```

Следующая программа осуществляет транспонирование квадратной двоичной матрицы, т. е. матрицы, элементами которой являются нули и единицы. Общий вид такой матрицы размером 5×5 и произвольное размещение ее в памяти показаны на рис. 3.15. Как видно, элементы матри-

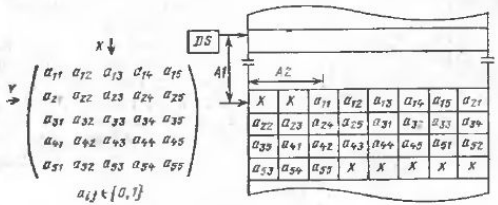


Рис. 3.15. Двоичная матрица и ее размещение в памяти

цы размещаются по строкам подряд. Предполагается, что первый байт матрицы имеет четный адрес.

Транспонирование заключается в обмене элементов a_{ij} и a_{ji} для всех $i \neq j$. Логика алгоритма транспонирования представляется следующим фрагментом на языке высокого уровня:

```
FOR X=1 TO N-1
    Y=X * N
    XB=X+A2
    YB=Y+A2
    WHILE (XB NE YB) DO
        SWAP [XB, YB]
```

Продолжение

```
        XB=XB+N
        YB=YB+1
    END
END
```

Здесь введены следующие обозначения: X — номер столбца; Y — номер строки; XB — позиционное положение столбцового транспонируемого бита в двоичной цепочке, с помощью которой матрица представлена в памяти; YB — аналогично XB, но применительно к строчному транспонируемому биту; A2 — смещение первого элемента матрицы в байте.

Алгоритм состоит из двух циклов. Внутренний цикл WHILE (начало его в программе идентифицируется меткой NEXT) реализует обмен (SWAP) элементов столбца X и соответствующей строки Y до тех пор, пока не будет достигнут диагональный элемент. Внешний цикл FOR, начинающийся в программе с метки LOOPX, осуществляет последовательное прохождение всех столбцов. Таким образом, на первом проходе внешнего цикла будут транспонированы элементы a_{12} и a_{21}, на втором — a_{13} и a_{31}, a_{23} и a_{32} и т. д.

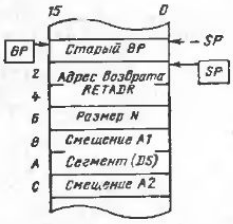


Рис. 3.16. Состояние стека после вызова и образования стекового кадра

Подпрограмме транспонирования в стеке передаются три параметра. Первый параметр (A2) показывает, с какого бита первого байта начинаются элементы матрицы. Второй параметр представляет собой полный начальный адрес матрицы сегмент:смещение, причем смещение обозначается A1. Наконец, третий параметр (N) определяет размер матрицы. Состояние стека после вызова подпрограммы показано на рис. 3.16.

Подпрограмма начинается с определения переданных параметров в более удобной форме, для чего используются директивы EQU. После этого создается стековый кадр — текущее (старое) содержимое регистра BP включается в стек, а затем в регистр BP передается содержимое указателя стека SP. Далее стандартным образом в стеке запоминается содержимое регистров, используемых подпрограммой.

С метки LOOPX начинается внешний цикл (по столбцам), и в регистрах AX и BX образуются значения YB и XB соответственно.

Команда с меткой NEXT определяет начало внутреннего цикла. В стеке запоминаются значения YB и XB, которые понадобятся при новом проходе внутреннего цикла. Следующие команды, начиная с метки ROW и кончая меткой COLUMN перед командой, выполняют следующие действия:

в регистре SI образуется адрес байта, содержащего транспонируемый бит в строке;

в регистре AH формируется байт a_{i}0000000, содержащий транспонируемый бит слева;

в регистре AL образуется байт, содержащий нуль в позиции транспонируемого бита и единицы в остальных битах (например, 111a_{i}1111).

Команды, начинающиеся с метки COLUMN, осуществляют точно такие же действия для транспонируемого бита в столбце. Здесь вместо регистра SI используется регистр BX, вместо регистра AH — регистр DH и вместо AL — регистр DL.

Собственно транспонирование (обмен элементов a_{ij} и a_{ji}) выполняют семь команд, начинающихся с метки EXCHG.

Затем осуществляется модификация значений XB и YB и проверка условия окончания внутреннего цикла, а при выходе из него — инкремент X и организация внешнего цикла. Заключительные команды POP восстанавливают содержимое рабочих регистров, а команда RET8 производит возврат и уничтожение параметров в стеке.

5. Рассмотрим программу, иллюстрирующую применение цепочечных команд МП К1810. Исходными данными для программы являются символьные массив (таблица) и цепочка. Задача состоит в том, чтобы определить, содержится ли заданная цепочка в таблице. При положительном ответе необходимо возвратить адрес элемента таблицы, соответствующий началу найденной цепочки, а при отрицательном — некоторый индикатор несоответствия [8].

Символический адрес начала таблицы обозначим T_START, конечный T_END, а соответствующие адреса отыскиваемой цепочки S_START и S_END. В приводимой программе для определенности принято, что длина таблицы

составляет 120 символов и что таблица инициализнрована на конкретный текст. Принята также конкретная цепочка из 15 символов. В практических применеиинях такие иллюстративные предположения, конечно, не нужны. Программа составлена в общем внде и требует только, чтобы длина цепочки была меньше длины таблицы, что представляется вполне естественным.

Алгоритм поиска заключается в следующем. Сначала берется первый символ цепочки и таблица просматривается на наличие этого символа. При безуспешном поиске задача решена — заданной символьной цепочки в таблице нет. При успешном поиске следует (начиная с найденного элемента таблицы) сравнивать последовательные символы таблицы и цепочки. Если сравнение оказалось успешным для всей цепочки, ответ в задаче положительный — в таблице найдена заданная цепочка, и адрес элемента таблицы, равного первому символу цепочки, является выходным параметром. Если это сравнение оказалось безуспешным, придется снова взять первый символ цепочки и повторить все действия, но уже начиная с элемента таблицы, следующего за тем, который был найден в предыдущем просмотре.

Программа 36. Транспонирование двоичной мвтрицы

```
NAME    TRANSP
  SEGA  SEGMENT  AT  100 H
        ASSUME   CS:SEGA, DS:SEGA, SS:SEGA, ES:SEGA
; Определепие переданных параметров
A2      EQU      WORD PTR [BP+12] ; Смещение первого
                                  ;   бита
AI      EQU      DWORD PTR [BP+8] ; Адрес матрицы
N       EQU      WORD PTR [BP+6]  ; Размер матрицы
RETADR  EQU      DWORD PTR [BP+2] ; Адрес возврата
; Образование стекового кадра для адресации параметров
        PUSH     BP
        MOV      BP,SP
; Запоминание рабочих регистров подпрограммы
        PUSH     AX
        PUSH     BX
        PUSH     CX
        PUSH     DX
        PUSH     SI
        PUSH     DI
        PUSH     DS
        LDS      AX,A1            ; Сегмент матрицы в
                                  ;   DS
; Внешний цикл с индексом X от I до N—I
        MOV      AX,1             ; Счетчик столбцов
```



```
LOOPX:  PUSH     AX               ; Запомнить индекс в
                                  ;   стеке
        MOV      BX,AX            ; Сохранить X для вы-
                                  ;   числения XB
        MUL      N                ; Вычислить начало
                                  ;   строки
        ADD      AX,A2            ; Учесть начало матрицы
        ADD      BX,A2            ; Учесть начало матрицы
; Внутренний цикл — транспонирование элементов
NEXT:   PUSH     AX               ; Запомнить строку
                                  ;   (YB)
        PUSH     BX               ; Запомнить столбец
                                  ;   (XB)
ROW:    MOV      CL,AL            ; Младший байт YB
        ADN      CL,07H           ; Выделить смещение в
                                  ;   байте
        MOV      SI,AX            ; Образовать
        SHR      SI,1             ;   в регистре SI;
        SHR      SI,1             ;   адрес байта, со-
        SHR      SI,1             ;   держащего эле-
                                  ;   мент в строке
        ADD      SI,WORD PTR A1   ; Учесть смещение
                                  ;   матрицы
        MOV      AH,[SI]          ; Байт строки с эле-
                                  ;   ментом
        SNL      AH,CL            ; Сдвинуть в старший
                                  ;   бит
        AND      AH,80H           ; Выделить бит эле-
                                  ;   мента
        MOV      AL,7FH           ; Подготовить байт
                                  ;   маски
        ROR      AL,CL            ; Образовать маску
        MOV      CH,CL            ; Сохранить счетчик
                                  ;   сдвигов строки
COLUMN: MOV      CL,BL            ; Младший байт XB
        AND      CL,07H           ; Выделить смещение
                                  ;   в байте
        SHR      BX,1             ; Образовать в регист-
                                  ;   ре BX
        SHR      BX,1             ;   адрес байта, со-
                                  ;   держащего
        SHR      BX,1             ;   элемент в столб-
                                  ;   це
        ADD      BX,WORD PTR A1   ; Учесть смещение
                                  ;   матрицы
        MOV      DX,[BX]          ; Байт столбца с эле-
                                  ;   ментом
        SHL      DH,CL            ; Сдвинуть в старший
                                  ;   бит
        AND      DH,80 H          ; Выделить бит эле-
                                  ;   мента
        MOV      DL,7 FH          ; Подготовить байт
                                  ;   маски
        ROR      DL,CL            ; Образовать маску
```

```
; Произвести обмен элементов
EXCHG:  SHR     AH, CL          ; Сдвинуть на новое
                                ;   место
        MOV     CL, CH          ; Восстановить счет-
                                ;   чик сдвига
        SHR     DH, CL          ; Сдвинуть на новое
                                ;   место
        AND     [BX], DL        ; Произвести обмен
        OR      [BX], AH        ;   элементов
        AND     [SI], AL
        OR      [SI], DH

; Проверить окончание внутреннего цикла
        POP     BX              ; Восстановить XB и
                                ;   YB
        POP     AX
        ADD     BX, N           ; Модифицировать XB
                                ;   и YB
        INC     AX
        CMP     AX,BX           ; Цикл закончен?
        JNE     NEXT            ; Нет, повторить

; Проверить окончание внешнего цикла
        POP     AX              ; Восстановить индекс
                                ;   X
        INC     AX              ; Перейти к новому
                                ;   значению
        CMP     AX, N           ; Транспонирование
                                ;   закончено?
        JB      LOOPX           ; Нет, повторить для
                                ;   столбца

; Транспонирование закончено
        POP     DS              ; Восстановить преж-
        POP     DI              ;   нее
        POP     SI              ;   содержимое ра-
                                ;   бочих
                                ;   регистров
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        POP     BP
        RET     8

                                ; Возврат и удаление
                                ;   параметров
SEGA    ENDS

        END
```


В программе, схема которой приведена на рис. 3.17, принято следующее распределение регистров: регистры ES : DI адресуют текущий элемент таблицы, регистры DS : SI являются указателем текущего символа цепочки, регистр CX

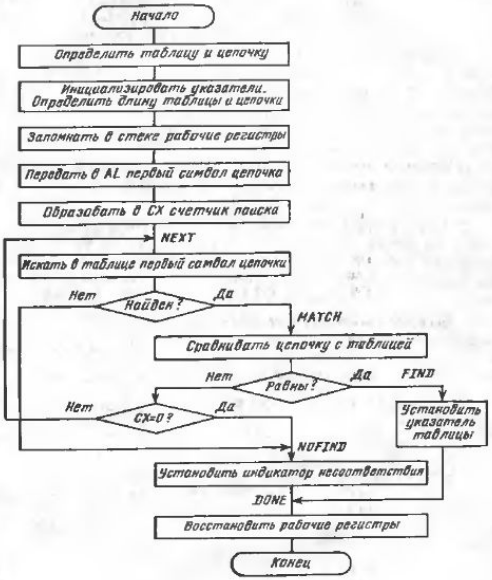


Рис. 3.17. Схема программы поиска цепочки в таблице

выполняет функцию счетчика, регистр BX содержит длину цепочки. Индикатором несоответствия служит регистр DI — если цепочка не найдена, он будет содержать код FFFF.

Программа содержит определения содержимого таблицы и цепочки, для чего применяются директивы DB. Собственно программа, оформленная в виде подпрограммы

типа FAR, начинается с инициализации указателей и вычисления длин таблицы и цепочки. Кроме того, в стеке запоминается содержимое регистров AX и DX, которые используются подпрограммой. Затем организуется цикл поиска в таблице элемента, равного первому символу цепочки. Начало этого цикла идентифицирует метка NEXT. Счетчик цикла в регистре CX равен разности длин таблицы и цепочки, увеличенной на 1. Собственно поиск реализуется цепочечной командой сканирования SCAS с префиксом повторения REPNE. При безуспешном поиске программа переходит на метку NOFIND, загружает в регистр DI код FFFF и заканчивается. При успешном поиске первого символа начинается цикл проверки того, что с данного элемента таблицы начинается отыскиваемая цепочка. Главной командой в этом цикле является цепочечная команда сравнения цепочек CMPS с префиксом REPE.

Если результат сравнения положителен, программа выходит на метку FIND, формирует в регистре DI адрес элемента таблицы, соответствующий началу цепочки, и осуществляет возврат. При безуспешном сравнении программа либо переходит на метку NEXT (новый цикл поиска в таблице первого символа цепочки), либо заканчивается с отрицательным результатом, когда таблица исчерпана.

В приведенной ниже программе, оформленной как исходный модуль, метки соответствуют схеме на рис. 3.17.

Программа 37. Поиск цепочки в таблице

```
        NAME SEARCH
DGROUP  GROUP DATA
DATA    SEGMENT PUBLIC 'DATA'
T_START DB      'OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO'
S1      DB      'OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO'
S2      DB      'HERE OOOOOOOOOOOOOOOOOOOOOOOOO'
S3      DB      'HERE IS A MATCH OOOOOOOOOOOOOO'
T_END   DB      0
S_START DB      'HERE IS A MATCH'
S_END   DB      0
; Полное соответствие в позиции 90
DATA    ENDS
; Собственно программа
CGROUP  GROUP   CODE
        ASSUME  CS:CGROUP, DS:DGROUP,
                ES:DGROUP, SS:DGROUP
        PUBLIC  SRCH
CODE    SEGMENT PUBLIC 'CODE'
SRCH    PROC    FAR
```



```
; Инициализация
        MOV     AX, SEG T_START     ; ES:DI есть
        MOV     ES, AX              ;   указатель
                                    ;   таблицы
        MOV     DI, OFFSET T_START
        MOV     CX, OFFSET T_END    ; Длина табли-
                                    ;   цы
        SUB     CX, OFFSET T_START  ;   в регистре
                                    ;   CX
        MOV     DS, AX              ; DS:SI есть
        MOV     SI, OFFSET S_START  ; указатель це-
                                    ;   почки
        MOV     BX, OFFSET S_END    ; Длина цепоч-
        SUB     BX, OFFSET S_START  ;   ки в реги-
                                    ;   стре BX
        ;
        PUSH    AX                  ; Запомнить
        PUSH    DX                  ;   рабочие
                                    ;   регистры
        PUSH    CX                  ; Запомнить
                                    ;   длину
                                    ;   таблицы
        CLD                         ;   Возраста-
                                    ;   ние адре-
                                    ;   сов при
                                    ;   поиске
        MOV     AL, [SI]            ; Первый сим-
                                    ;   вол цепоч-
                                    ;   ки
        SUB     CX, BX              ; Образовать
                                    ;   значение
        INC     CX                  ;   счетчика
                                    ;   поиска
NEXT:
; Искать первый символ в таблице
REPNE   SCAS    B                   ; Искать, пока
                                    ;   не равны
        JE      MATCH               ; Символ обна-
                                    ;   ружен
NOFIND: MOV     DI, -1              ; Цепочка не
                                    ;   найдена
        JMP     DONE
; Сравнивать всю цепочку
MATCH:  MOV     DX, CX              ; Возможный
                                    ;   счетчик по-
                                    ;   иска
        PUSH    DI                  ; Запомнить по-
                                    ;   зицию в
                                    ;   таблице
```

Продолжение

```
                MOV     CX,BX       ; Длина цепочки в CX
                PUSH    SI          ; Запомнить указатель цепочки
                DEC     DI          ; Учесть длину, равную 1
REPE            CMPS    B           ; Сравнивать таблицу и цепочку

; Результат сравнения — состояния ZF и CX
                POP     SI          ; Восстановить указатель цепочки
                POP     DI          ; Восстановить позицию в таблице
                MOV     CX, DX      ; Восстановить счетчик поиска
                JE      FIND        ; Соответствие найдено
                JCXZ    NOFIND      ; Соответствия нет
                JMP     NEXT        ; Повторить основной цикл
                POP     DI          ; Длина таблицы
FIND:           SUB     DI,DX       ; Адрес соответствия

; Восстановить регистры
DONE:           POP     DX
                POP     AX
                RET                 ; Возврат
SRCH            ENDP
CODE            ENDS
                END
```

6. Таблицы, содержащие фиксированные данные, можно использовать как функциональные преобразователи, в которых по заданному входному значению (аргументу) считывается хранимое в таблице выходное значение (функция). Табличное преобразование применяется в тех случаях, когда функция не имеет аналитического выражения или когда оно существует, но время вычисления превышает допустимое. Конечно, для организации табличных преобразований необходима память соответствующей емкости.

Ниже рассмотрены два типичных применения таблиц [11].

Стандартное вычисление тригонометрических функций путем разложения в ряд требует слишком много времени. Если требования точности невелики (угол задается с точностью до одного градуса в диапазоне от 0 до 360, а значение синуса имеет точность до четвертого десятичного разряда), табличное преобразование оказывается очень быстрым, а размер таблицы невелик. В подпрограмме SINE предполагается, что угол X передается в регистре AX, а значение синуса в прямом коде возвращается в регистре BX. Таблица синусов углов из первого квадранта содержит 91 значение с масштабным коэффициентом 10 000. Таблица размещается в сегменте данных и определяется следующим образом (показаны две первых и последняя строки):

```
TSIN    DW      0,175,349,523,698,875       ; 0—5 градусов
        DW      1045,1219,1392,1564,1736    ; 6—10 градусов
                . . .
        DW      9976,9986,9994,9998,10000   ; 86—90 градусов
```

Получение синуса угла X, находящегося в различных квадрантах, осуществляется по формулам:

первый квадрант	sin X
второй квадрант	sin(180—X)
третий квадрант	— sin(X—180)
четвертый квадрант	— sin(360—X)

Программа 38. Табличное получение синуса

```
SINE    PROC
        PUSH    AX          ; Сохранить угол
        PUSH    CX          ; Освободить рабочий регистр
        SUB     CX,CX       ; Знак положительный
        CMP     AX,181      ; Угол меньше 181 градуса?
        JB      POS         ; Да, синус положительный
        MOV     CX,8000H    ; Нет, синус отрицательный
        SUB     AX,180      ; Образовать X—180
POS     CMP     AX,91       ; Угол меньше 91 градуса?
        JB      GETS        ; Да, обратиться к таблице
        NEG     AX          ; Образовать 180—X
        ADD     AX,180
```

Продолжение

```
GETS    MOV     BX,AX           ; Образовать в BX индекс таб-
        SHL     BX,1              лицы
        MOV     BX,[BX] TSIN    ; Обратиться к таблице
        OR      BX,CX           ; Учесть знак
        POP     CX              ; Восстановить регистры
        POP     AX
        RET
SINE    ENDP
```

Время выполнения подпрограммы SINE без учета команд CALL и RET составляет около 120 тактов синхронизации.

Благодаря тождеству

$$\cos X = \sin (X + 90)$$

подпрограмму SINE можно использовать для получения косинуса угла X.

Программа 39. Табличное получение косинуса

```
COS     PROC
        PUCH    AX          ; Сохранить угол
        ADD     AX,90       ; Образовать X+90
        CMP     AX,360      ; Привести угол в диапазон
        JNA     GETC
        SUB     AX,360
GETC    CALL    SINE        ; Получить косинус
        POP     AX          ; Восстановить угол
        RET
COS     ENDP
```

В некоторых задачах удобно хранить в таблице адреса управляющих программ, процедур обработки прерываний, сообщений и т. д. По аргументу, которым является код нажатой клавиши, тип прерывания или номер сообщения, из таблицы считывается соответствующий адрес. Такая таблица обычно называется таблицей переходов. Этот прием

использован в микропроцессоре К1810ВМ86 для обработки прерываний.

Следующая программа показывает применение таблицы переходов для обслуживания пяти пользователей в многотерминальной системе. Здесь содержимое регистра AL интерпретируется как код идентификации пользователя (0—4) и используется для перехода к процедуре обслуживания соответствующего пользователя. Если код пользователя превращается в индекс таблицы, и осуществляется косвенный переход к процедуре обслуживания пользователя. В таблице хранятся полные адреса сегмент:смещение, которые должны быть определены директивами DD и иметь начальный адрес JUMPT.

Программа 40. Табличный переход к процедуре

```
USER    PROC
        CMP     AL,4            ; Проверить код пользователя
        JA      ERROR           ; Недействительный код
        XOR     AH,AH           ; Сбросить регистр AH
        MOV     DI,AX           ; Передать код в регистр
        SHL     DI,1            ; Образовать индекс таблицы
        SHL     DI,1            ; Перейти
        JMP     FAR PTR JUPT[DI] ; Обработать ошибку
ERROR:  . . .
        . . .
        RET
USER    ENDP
```

ГЛАВА 4

ОСОБЕННОСТИ АППАРАТНЫХ КОНФИГУРАЦИЙ

4.1. Цикл шины

Макропроцессор К1810 взаимодействует с внешней средой с помощью 20-битной шины адреса/данных/состояния и нескольких управляющих сигналов (см. § 1.2). Собственно взаимодействие заключается в выполнении одной из двух операций: МП либо выводит (записывает) данные,

либо вводит (считывает) данные или команды. Конечно, в каждой из этих операций МП должен указывать то устройство, с которым он будет взаимодействовать; другими словами, МП должен адресовать ячейку памяти либо порт (устройство) ввода или вывода.

Для передачи данных или выборки команды МП инициирует так называемый цикл шины. По существу этот термин эквивалентен понятию машинного цикла, которое фигурирует в описании работы МП К580. Однако термин «цикл шины» несколько шире в том смысле, что цикл шины может инициировать не только МП, но, например, и процессор ввода-вывода или арифметический сопроцессор.

Цикл шины представляет собой последовательность событий, в течение которой МП выдает адрес ячейки памяти или периферийного устройства, а затем формирует сигнал записи или считывания, а также соответствующие данные в операции записи. Выбранное устройство воспринимает данные с шины в цикле записи или помещает необходимые данные на шину в цикле считывания. По окончании цикла шины устройство фиксирует записываемые данные или снимает считываемые данные.

Простейшая временна́я диаграмма цикла шины в общем виде представлена на рис. 4.1, а более подробная диаграмма для минимального режима — на рис. 4.2. Звездочки на

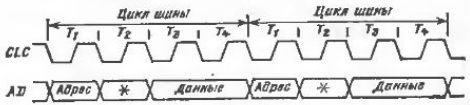


Рис. 4.1. Простая интерпретация цикла шины

диаграмме рис. 4.1 обозначают, что линии шины находятся в высокоимпедансном состоянии при считывании или содержат выводимые данные при записи.

Цикл шины состоит минимум из четырех тактов синхронизации, называемых также состояниями T, которые идентифицируются спадающим фронтом сигнала синхронизации CLC. В первом такте (T_1) микропроцессор выдает на шину адреса /данных/состояния адрес устройства, которое будет источником или получателем информации в текущем цикле шины. Во втором такте (T_2) микропроцессор снимает адрес

с шины и либо переводит тристабильные буфера линий AD15—AD0 в высокоимпедансное состояние, подготавливая их к вводу информации в цикле считывания, либо выдает на них данные в цикле записи. Работа шинных формирователей (если они есть в системе минимальной конфигурации)

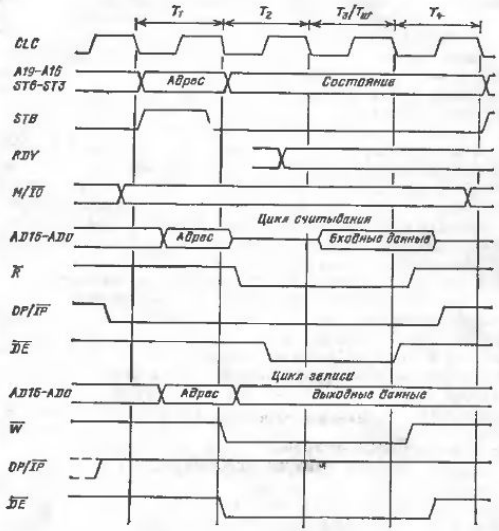


Рис. 4.2. Основные циклы шины

разрешается в тактах T_1 или T_2 в зависимости от системной конфигурации и направления передачи информации. Управляющие сигналы, инициирующие считывание, запись или подтверждение прерываний, всегда выдаются в такте T_2. В максимальной конфигурации (см. ниже) сигнал записи формируется в такте T_3, чтобы гарантировать стабилизацию сигналов данных до действия сигнала.

В такте T_2 старшие четыре линии адреса/состояния переключаются с адреса на состояние ST6—ST3. Сигналы состояния предназначены в основном для диагностических целей. Напомним, что сигналы ST4—ST3 идентифицируют сегментный регистр, который участвует в формировании адреса памяти. Следовательно, с помощью подключенного к линиям этих сигналов дешифратора можно выбрать (разрешить) один из четырех банков памяти, выделенных для каждого сегмента. Такой прием обеспечивает расширение адресуемой памяти до 4М байт и защиту от ошибочной операции записи в один из сегментов, перекрывающийся с другими сегментами.

В течение такта T_3 микропроцессор сохраняет на линиях ST6—ST3 информацию о состоянии. На шине AD в цикле записи сохраняются выводимые данные, а в цикле считывания производится опрос вводимых данных. Если адресованная память или периферийное устройство не может принимать или передавать информацию синхронно с действиями МП, она должна до начала такта T_3 сформировать низкий уровень сигнала готовности RDY. Это заставляет МП ввести после такта T_3 дополнительные такты, называемые тактами (состояниями) ожидания T_w. В тактах T_w на линиях шины действуют такие же уровни сигналов, что и в такте T_3. Когда адресованное медленное устройство получило достаточно времени для завершения операции, оно формирует высокий уровень на входе готовности микропроцессора, что заставляет его перейти к такту T_4, которым заканчивается цикл шины. В этом такте снимаются все управляющие сигналы и выбранное устройство отключается от шины.

Таким образом, цикл шины для памяти или периферийного устройства представляет собой асинхронное действие, состоящее из выдачи адреса для выбора нужного устройства с последующим стробом считывания или данных и стробом записи. По окончании стробов устройство фиксирует записываемые данные или запрещает свои шинные формирователи. Устройство может управлять циклом шины только путем введения состояний ожидания.

Микропроцессор выполняет цикл шины, если ему необходимо осуществить запись или считывание информации. Если циклы шины не требуются, шинный интерфейс реализует холостые состояния T_i. В течение T_i МП сохраняет на линиях ST6—ST3 сигналы состояния от предыдущего цикла шины. Если предыдущий цикл шины был записью, МП

сохраняет на линиях AD записываемые данные до следующего цикла шины. Если МП переведен в холостое состояние после цикла считывания, он не управляет линиями AD до тех пор, пока не потребуется следующий цикл шины.

Микропроцессор выбирает во внутреннюю очередь до 6 байт командного потока (программы), поэтому взаимосвязь между выборкой команды и относящимися к ней передачами данных (операнда и результата) может быть разделена дополнительными циклами шины для выборки других команд. Следовательно, временная диаграмма работы этого МП оказывается сложнее диаграммы микропроцессоров без опережающей выборки команд (в последнем случае циклы шины, связанные с выборкой команды и относящимся к ней передачами данных, следуют строго друг за другом).

В общем, если команда выбрана во внутреннюю очередь, до ее извлечения из очереди и собственно выполнения может быть выбрано несколько дополнительных байт командного потока. Если извлекаемая из очереди команда оказывается командой передачи управления, то все оставшиеся в очереди команды не выполняются. Микропроцессор осуществляет реинициализацию очереди из новой программной области, определяемой адресом перехода. Таким образом, действия шины при выполнении какой-либо команды, конечно, зависят от предыдущих команд, но в каждой конкретной последовательности команд всегда детерминированы.

4.2. Шины адреса и данных

Для правильного функционирования памяти и периферийных устройств обычно требуется стабильный адрес в течение всего цикла шины. Поэтому адрес, выдаваемый на линии AD и A/S в такте T_1, необходимо запомнить (зафиксировать) и использовать зафиксированный адрес для выбора периферийного устройства или ячейки памяти. Специально для демультиплексирования шины адреса/данных/состояния МП формирует сигнал строба STB разрешения регистра-защелки адреса, показанный на рис. 4.2. Он применяется для загрузки адреса в регистры-защелки (рис. 4.3), имеющие большую нагрузочную способность, малое время переключения и тристабильные выходные буфера. При высоком уровне сигнала STB входные сигналы на линиях DI повторяются на выходах DO, а при переходе сигнала STB от высокого уровня к низкому фикси-

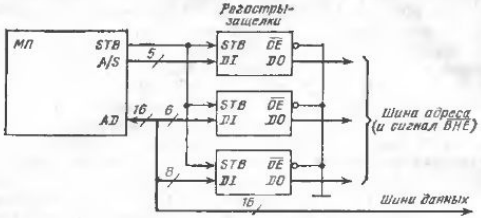
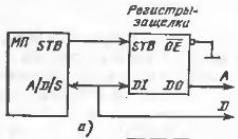


Рис. 4.3. Демультиплексирование шины



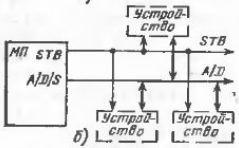
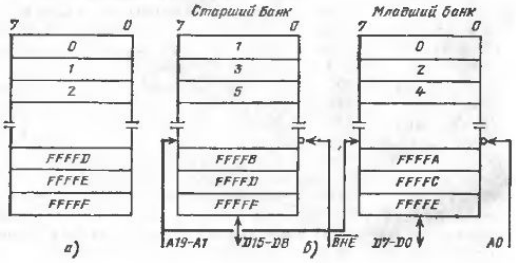


Рис. 4.4. Способы демультиплексирования шины

Рис. 4.5. Логическое и физическое адресные пространства памяти



руются значения сигналов D1. Обычно регистры-защелки имеют входной сигнал $\overline{OE}$ разрешения выхода, который в данном случае подключен на землю. Задержка распространения сигнала регистров-защелок вносит некоторое запаздывание в тракт дешифрирования адреса.

Демультиплексирование шины адреса/данных/состояния осуществляется в процессорном модуле или локально в соответствующих компонентах системы. Оба способа представлены на рис. 4.4, *а* и *б* соответственно. В первом способе реализуется отдельная шина адреса, распределяющая адрес по всей системе. Чтобы оптимизировать производительность системы и обеспечить совместимость с мультипроцессорными конфигурациями, рекомендуется применять именно этот способ, который и предполагается в дальнейшем изложении.

Логическое адресное пространство памяти МП К1810 для программиста представляется линейной последовательностью из 1М байт, которая показана на рис. 4.5, *а*. Любой байт содержит 8-битный элемент данных, а любые два смежных (соседних) байта содержат 16-битный элемент данных — слово. Ограничения на адреса байт и слов не накладываются. Физическое адресное пространство памяти с 16-битной шиной данных D15—D0 реализовано с помощью двух банков памяти по 512К байт каждый, представленных на рис. 4.5, *б*. Один банк, подключенный к младшей половине D7—D0 шины данных, называется младшим; он содержит байты с четными адресами, у которых младший бит адреса A0=0. Второй банк (старший) подключен к старшей половине D15—D8 шины данных и содержит байты с нечетными адресами, у которых A0=1. Конкретный байт в каждом банке выбирается (адресуется) сигналами на линиях A19—A1.

Байты с четными адресами передаются по линиям D7—D0 (рис. 4.6), причем разрешение соответствующего банка осуществляется сигналом на линии A0. При этом высокий уровень выходного сигнала $\overline{BHE}$, формируемого микропроцессором, запрещает старший банк памяти. Такое действие необходимо для того, чтобы операции записи в младший банк не разрушали данных в старшем банке. Так как временная диаграмма мультиплексного сигнала $\overline{BHE}$/ST7 аналогична диаграмме сигналов A19—A16/ST6—ST3, то значение сигнала $\overline{BHE}$ необходимо зафиксировать с помощью строба STB.

Передачи байт с нечетными адресами (рис. 4.7) осуществляются по линиям D15—D8, для чего низкий уровень сигнала $\overline{BHE}$ разрешает старший банк памяти, а высокий уровень сигнала A0 запрещает младший банк.

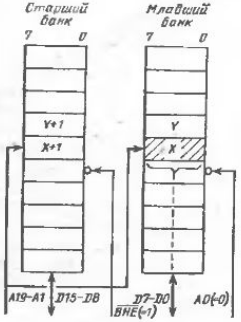


Рис. 4.6. Передача байта с четным адресом

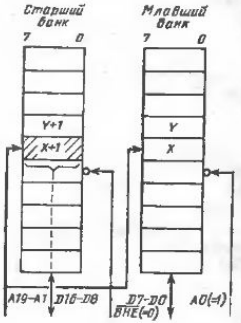


Рис. 4.7. Передача байта с нечетным адресом

Управление передачами данных по соответствующей половине шины данных и формирование требуемых уровней сигнала $\overline{BHE}$ микропроцессор реализует автоматически без специальных указаний программиста. Например, при загрузке байта данных в регистр BL из ячейки памяти с нечетным адресом данные передаются в МП по линиям D15—D8, автоматически направляются на младшую половину внутреннего 16-битного тракта данных и попадают в регистр BL. Такая особенность позволяет осуществлять байтные передачи ввода-вывода из аккумулятора AL в периферийные устройства, подключенные к старшей или младшей половине шины данных.

В обращениях к словам с четными адресами сигналы на линиях A19—A1 выбирают соответствующие байты в каждом банке, а сигналы A0 и $\overline{BHE}$ разрешают оба банка одновременно, что показано на рис. 4.8. Слово передается в одном цикле шины по линиям D15—D0. При обращениях к словам с нечетными адресами (рис. 4.9) сначала по линиям

D15—D8 передается младший байт; сигнал $\overline{BHE}$=0 (старший банк разрешен), а сигнал A0=1 (младший банк запрещен). Для доступа к старшему байту производится инкремент полного адреса A19—A0, а затем выполняется второй цикл шины — старший байт слова передается из младшего банка памяти ($\overline{BHE}$=1, A0=0). Такую последовательность МП реализует автоматически, когда команда требует передачи слова с нечетным адресом. Маршрутизация старшего и младшего байт внутренних 16-битных регистров на соответствующие половины шины данных также выполняется автоматически и незаметна для программиста.

В операциях считывания байта МП переводит в высокоимпедансное состояние (такт T_2) буфера всех линий шины

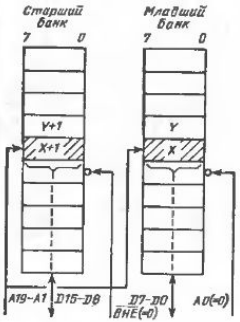
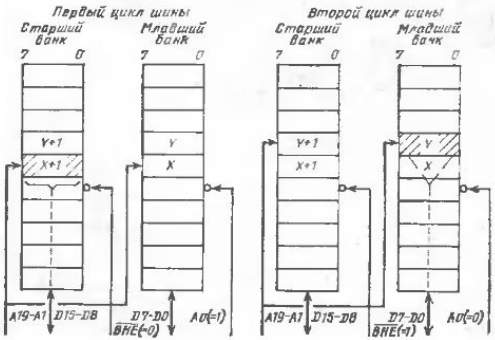


Рис. 4.8. Передача слова с четным адресом

Рис. 4.9. Передача слова с нечетным адресом



D15—D0, хотя данные ожидаются только на одной ее половине. Такое действие упрощает формирование выбирающих (селектирующих) сигналов для устройств, выполняющих только операцию считывания. В операциях записи байта МП управляет всей 16-битной шиной данных, но информация на ненужной половине шины не определена.

Рассмотренные выше принципы передач байт и слов аналогичны и для адресного пространства ввода-вывода.

4.3. Системная шина данных

Имеются две возможные реализации системной шины данных: мультиплексная шина адреса/данных (рис. 4.10, *а*) и шина данных, отделенная от мультиплексной шины с по-

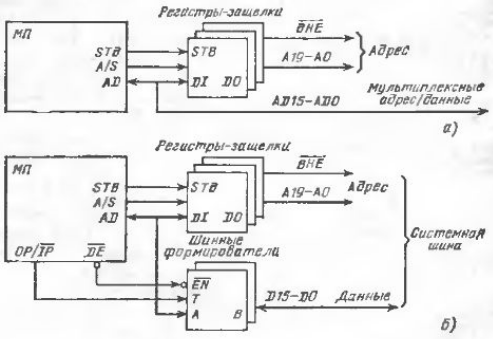


Рис. 4.10. Организация шины данных

мощью двунаправленных шинных формирователей (рис. 4.10, *б*).

Если память или периферийные устройства подключены непосредственно к мультиплексной шине, необходимо гарантировать, что эти устройства не исказят адрес на шине в течение такта T_1. Чтобы избежать такого влияния, устройства должны иметь вход разрешения выдачи $\overline{OE}$, на который подается системный сигнал считывания $\overline{R}$. Временна́я диаграмма работы МП гарантирует, что сигнал $\overline{R}$ не генерируется до тех пор, пока адрес не будет зафиксирован стробом STB. Большинство микросхем имеют сигналы разрешения выдачи $\overline{OE}$ или считывания $\overline{RD}$, обеспечивающие подключение к системной шине.

Интерфейс с мультиплексной шиной устройств, не имеющих управляющего входа $\overline{OE}$, осуществляется несколькими способами, но у каждого из них имеются свои недостатки и ограничения. Например, возможно объединить по И сигнал выбора кристалла $\overline{CS}$ (с выхода дешифратора старших линий адреса) и сигналы считывания или записи. Однако в этом способе время обращения по входу $\overline{CS}$ сокращается до времени обращения по сигналу считывания, а это может потребовать более быстродействующих устройств, если необходима максимальная производительность системы (работа без введения состояний ожидания). Кроме того, необходимо убедиться в том, что для устройств не нарушаются временные соотношения между сигналом $\overline{CS}$ и временами установления записи и сохранения. Другие способы интерфейса связаны с соответствующими ограничениями, поэтому лучшим решением оказывается способ с устройствами, имеющими входной сигнал разрешения выдачи $\overline{OE}$.

Еще одно ограничение мультиплексной шины объясняется малой нагрузочной способностью микропроцессора (ток 2 мА, емкость 100 пкФ), гарантирующей паспортные динамические характеристики. Для удовлетворения требований резистивной и емкостной нагрузок в большинстве систем шину данных необходимо буферировать с помощью шинных формирователей, имеющих высокую нагрузочную способность и тристабильные выходные каскады. Управление работой шинных формирователей (разрешение и направление передачи) осуществляется сигналами OP/$\overline{IP}$ и $\overline{DE}$. Временны́е характеристики этих сигналов гарантируют изолирование мультиплексной шины от системы в течение такта T_1 и устранение шинных конфликтов в циклах считывания и записи. При считывании сигнал $\overline{DE}$ разрешает работу шинных формирователей после того, как МП перевел мультиплексную шину в высокоимпедансное состояние. В цикле записи сигнал $\overline{DE}$ разрешает работу формирователей несколько раньше, но сигнал OP/$\overline{IP}$ гарантирует,

что формирователи находятся в режиме передачи и не будут влиять на микропроцессор.

Расширение системы связано с необходимостью использования второго уровня буферирования для устройств на системной шине с целью уменьшения суммарной нагрузки на шину (рис. 4.11). Обычно данный способ применяется

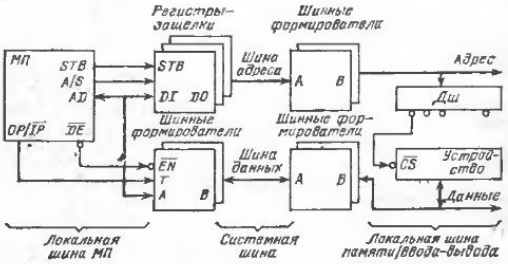


Рис. 4.11. Полностью буферированная шина

в многоплатных системах и для изолирования памяти большой емкости, содержащей значительное число микросхем. В этой конфигурации важную роль играют дополнительная задержка при обращении и управление дополнительными формирователями с учетом характеристик системной шины и подключенных к ней устройств.

4.4. Режимы работы

Важной и интересной особенностью МП К1810 является выбор базовой конфигурации, наиболее соответствующей конкретному его применению. Уровень сигнала на входе MN/$\overline{MX}$ позволяет выбрать одно из двух функциональных определений восьми сигналов микропроцессора и один из двух режимов работы.

Минимальный режим, системная конфигурация которого показана на рис. 4.12, предназначен для малых и средних систем, содержащих одну-две печатные платы с одним микропроцессором. Адресное пространство памяти составляет 1М байт, пространство ввода-вывода — 64К байт и шина

данных имеет 16 линий. Микропроцессор сам формирует все управляющие сигналы (OP/IP, DE, STB, M/$\overline{IO}$, R, $\overline{W}$).

Имеется также простой механизм приостановки действий микропроцессора (сигналы HLD и HLDA), совместимый с имеющимися контроллерами прямого доступа к памяти.

Максимальный режим расширяет системную архитектуру, допуская мультипроцессорные конфигурации и введе-

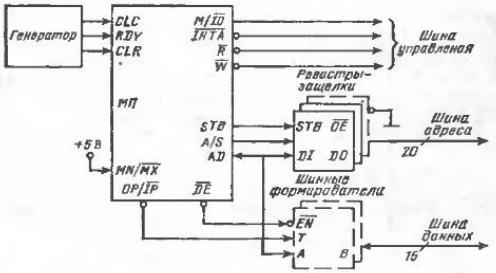


Рис. 4.12. Микропроцессор в минимальном режиме

ние сопроцессора. При наличии контроллера шины выходы микропроцессора, предназначенные для управления шиной в минимальном режиме, переопределяются с целью обеспечения указанных выше расширений и повышения производительности системы. Структурная схема системной конфигурации максимального режима представлена на рис. 4.13.

Переопределение сигнальных линий заключается в следующем:

две приоритетные линии запроса шины RQ/$\overline{E0}$ и $\overline{RQ}$/$\overline{E1}$ позволяют подключать к локальной шине МП несколько процессоров и разделять его интерфейс с системной шиной;

линии состояния очереди QS0, QS1 дают возможность сопроцессору следить за выполнением команд МП;

механизм блокировки шины обеспечивает управление доступом к разделяемым ресурсам;

расширены варианты системных конфигураций с помощью специальных микросхем контроллера шины и арбитра шины.

Сигналы состояния очереди QS, кодирование которых приведено в табл. 1.2, показывают, какая информация извлекается из внутренней очереди команд и когда очередь сбрасывается из-за передачи управления. Проверяя сначала сигналы состояния ST на выборку команд, а затем сигналы QS0 и QS1, можно проследить за выполнением отдельных команд программы. Так как команды выполняются

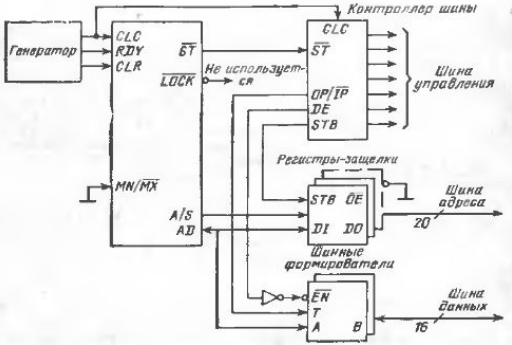


Рис. 4.13. Микропроцессор в максимальном режиме

из внутренней очереди микропроцессора, состояние очереди выдается на каждый такт синхронизации и не связано с действиями циклов шины. Анализируя сигналы на линиях QS, сопроцессор фиксирует выполнение предназначенной для него команды ESC, а при отладке системы с помощью этих линий можно определить момент выполнения команды из конкретной ячейки памяти.

Для управляемого доступа к разделяемым ресурсам предусмотрен выходной сигнал блокировки шины $\overline{\text{LOCK}}$, который формируется при выполнении однобайтного префикса LOCK (код операции F0). Сигнал $\overline{\text{LOCK}}$ становится активным в первом такте синхронизации после выполнения префикса LOCK и сохраняется активным до такта, следующего за завершением заблокированной команды. Этот

сигнал подается в логику арбитража системной шины (см. ниже).

При нормальной работе мультипроцессорной системы приоритет разделяемой системной шины определяется логикой арбитража такт за тактом. Когда МП требует передачи информации по системной шине, он запрашивает доступ к ней через логику арбитража. Если МП имеет достаточный приоритет, он получает управление шиной и выполняет свой цикл шины, а затем либо сохраняет управление шиной, либо сам освобождает ее, либо, наконец, отключается от шины из-за потери приоритета. Механизм блокировки предотвращает потерю микропроцессором управления шиной и гарантирует ему выполнение нескольких циклов шины без вмешательства и возможного искажения

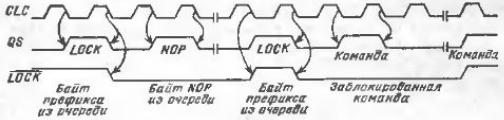


Рис. 4.14. Действие префикса и сигнала блокировки

данных другими процессорами. Классический пример блокировки связан с семафорами и командой обмена XCHG; он был рассмотрен в § 1.7. Временная диаграмма сигнала $\overline{\text{LOCK}}$ приведена на рис. 4.14.

Сигнал $\overline{\text{LOCK}}$ характеризуется следующими особенностями:

между двумя соседними заблокированными командами имеется некоторый интервал, когда сигнал $\overline{\text{LOCK}}$ не активен;

для дешифрирования префикса LOCK и выдачи сигнала $\overline{\text{LOCK}}$ требуются два такта синхронизации;

так как сигналы QS отражают операцию очереди команд в предыдущем такте, выход $\overline{\text{LOCK}}$ становится активным по совпадению с началом команды и остается активным на один такт после команды;

если команды, находящейся после префикса LOCK, нет в очереди, выход $\overline{\text{LOCK}}$ все-таки становится активным, пока производится выборка команды;

при выполнении заблокированной команды устройство шинного интерфейса продолжает выборку команд; сигнал $\overline{LOCK}$ резервирует шину для данного МП, когда он выполняет циклы шины во время заблокированной команды.

При выполнении заблокированной команды запрос приостановки МП по линиям $\overline{RQ}/\overline{E}$ фиксируется, но не подтверждается до завершения команды. Сигнал $\overline{LOCK}$ прямого воздействия на прерывания не оказывает. Например, заблокированная команда останова HLT заставляет игнорировать запросы $\overline{RQ}/\overline{E}$, но позволяет микропроцессору выйти из состояния останова по прерыванию.

Байты префиксов считаются расширением тех команд, которым они предшествуют. Поэтому возникающие при выполнении префиксов прерывания не подтверждаются (если они разрешены) до завершения команды, следующей за префиксами. Исключение составляют те команды, которые допускают обслуживание прерываний при их выполнении, например команды HLT, WAIT или повторяющиеся цепочечные команды.

Сигналы состояния $\overline{ST2}$, $\overline{ST1}$, $\overline{ST0}$ предназначены для взаимодействия с контроллером и арбитром шины. Они сообщают контроллеру шины, когда инициировать цикл шины, какие управляющие сигналы необходимо сформировать и когда закончить цикл шины. Контроллер опрашивает уровни сигналов $\overline{ST}$ в начале каждого такта синхронизации. Для инициирования цикла шины МП переводит сигналы $\overline{ST}$ из пассивного состояния ($\overline{ST}=111$) в одно из семи активных состояний, которые были приведены в табл. 1.1. Это происходит по нарастающему фронту сигнала синхронизации в такте T_4 предыдущего цикла шины или в холостом состоянии. Контроллер фиксирует изменение состояния путем опроса линий $\overline{ST}$ по переходу сигнала синхронизации от высокого уровня к низкому в каждом такте. Он начинает цикл шины путем генерирования строба адреса STB и сигнала $OP/\overline{IP}$ (управляющего направлением передач в шинных формирователях) в такте непосредственно за обнаружением изменения состояния, т. е. в такте T_1. В такте T_2 (или T_3 для обычного сигнала записи) разрешается работа формирователей и генерируются необходимые управляющие сигналы. Когда состояние возвращается в пассивный режим, контроллер заканчивает управление шиной.

Временная диаграмма сигналов состояния представлена на рис. 4.15. Так как МП не возвращает сигналы ST в пассивный режим до восприятия сигнала готовности RDY, контроллер шины будет сохранять управление шиной в течение любого числа тактов ожидания T_w.

Линии $\overline{ST}$ могут использовать другие процессоры, подключенные к локальной шине микропроцессора для контро-

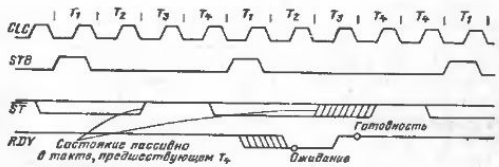


Рис. 4.15. Действие сигналов состояния

ля действий шины и управления контроллером, если он получат управление локальной шиной.

Контроллер генерирует сигналы DE, $OP/\overline{IP}$, STB и семь управляющих сигналов, включающих в себя подтверждение прерывания, считывание и запись в память, считывание и запись ввода-вывода, а также два опережающих сигнала записи в память и записи ввода-вывода. Таким образом, МП освобождается от функций генерирования управляющих сигналов шины.

Опережающие сигналы записи выдаются на один такт синхронизации раньше обычных сигналов записи, чтобы обеспечить более широкий импульс записи. Такой импульс часто требуется для периферийных схем и статической оперативной памяти. Обычные сигналы записи обеспечивают установление данных до сигнала записи, что необходимо для микросхем динамической памяти и периферийных устройств, которые стробируют данные по началу импульса записи. Опережающие сигналы записи не гарантируют действительность данных по началу импульса записи.

Отметим, что сигнал DE в максимальном режиме инвертируется по сравнению с сигналом в минимальном режиме, что в некоторых ситуациях оказывается более удобным.

В максимальном режиме сигналы HLD и HLDA превращаются в более сложные сигналы $\overline{RQ}/E0$ и $\overline{RQ}/E1$. Двунаправленные линии этих сигналов применяются для разделения во времени локальной шины МП между несколькими процессорами, например, арифметическим сопроцессором или процессором ввода-вывода. Трехфазная процедура запроса—разрешения—освобождения была достаточно под-

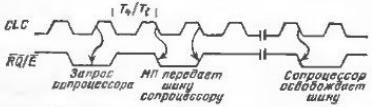


Рис. 4.16. Трехфазный цикл запроса (разрешения) освобождения

робно рассмотрена в § 1.7, а поясняющая ее временна́я диаграмма приведена на рис. 4.16.

Запаздывание разрешения доступа к шине при выполнении микропроцессором обычной команды, т. е. команды без префикса блокировки LOCK, составляет 3—10 тактов. Минимальное значение запаздывания получается для команд, не требующих обращения к памяти. Когда МП сформировал сигнал $\overline{LOCK}$, время запаздывания увеличивается. Сигнал $\overline{LOCK}$ вырабатывается в цикле подтверждения прерываиия или при выполненнн команды с префиксом LOCK. При выполнении заблокироваиной команды обмена XCHG запаздывание достигает 39 тактов синхронизацни. Еще большее запаздывание получается при выполненин заблокированных команд умножения и деления, а также цепочечных команд, поэтому их блокировка не рекомендуется.

Микропроцессор возобновляет управление шиной через два такта снихронизации после прнема от другого процессора снгнала освобождения шины (или снятия сигнала HLD в минимальном режиме).

Хотя максимальный режим рассчитан на мультипроцессорную среду, его следует применять и в сложных изделиях с одним мнкропроцессором. Высокая нагрузочная способность контроллера шины и его временны́е характеристикн обеспечивают лучшую производительность большой системы по сравненню с пронзводнтельностью в минимальном режиме.

4.5. Мультипроцессорные системы

Микропроцессор К1810 рассчитан на применение в мультипроцессорных системах, имеющих структуру с разделяемой системной шиной, которая иллюстрируется на рис. 4.17. Все процессоры системы, находящиеся в процессорных модулях, взаимодействуют друг с другом и разделяют ресурсы через системную шину. В качестве последней может использоваться стандартная шина или расширение системной шины, рассмотренное в § 4.3. При разделении шины необходима специальная схема арбитража, предназначенная для управления доступом к системной шине. Когда любой процессор асинхроино запрашивает доступ к разделяемой шине, схема арбитража, называемая также арбитром шины, учитывает приоритеты про-

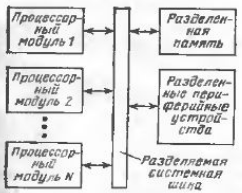
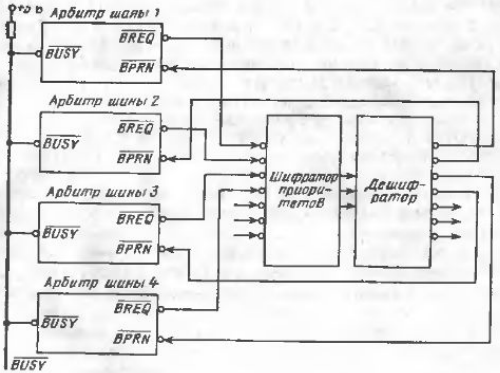


Рис. 4.17. Мультипроцессорная система

Рис. 4.18. Параллельный учет приоритетов



цессоров (при нескольких одновременных запросах) и разрешает доступ к шине процессору с наибольшим приоритетом. Получив разрешение, процессор реализует требуемый ему цикл шины и либо освобождает шину, либо ожидает приказа освободить ее.

Арбитр шины, выполненный по биполярной технологии и имеющий высокое быстродействие, обеспечивает арбитраж шины в мультипроцессорных системах с несколькими ведущими («запросчиками») шины, т. е. устройствами, которые могут запрашивать системную шину и управлять ею. Микропроцессор К1810, работающий в максимальном режиме, не заботится о наличии арбитра шины и формирует сигналы так, как будто он монопольно распоряжается системной шиной. Если шина не доступна, арбитр шины предотвращает доступ к ней контроллера шины, формирователей шины данных и регистров адреса, переводя их выходные каскады в высокоимпедансное состояние, и заставляет МП перейти в состояние ожидания, устанавливая сигнал RDY=0. Микропроцессор остается в этом состоянии до тех пор, пока арбитр не разрешит доступ к системной шине, т. е. не подключит к ней контроллер шины, формирователи шины данных и регистры адреса. Когда началась передача информации, в МП возвращается сигнал подтверждения передачи для указания готовности ведомого устройства, к которому производится обращение. Затем МП завершает свой цикл передачи. Таким образом, арбитр шины служит для мультиплексирования ведущих шины на системную шину и решения проблем конкуренции между ними.

Так как системную шину могут запрашивать несколько ведущих, необходимо учитывать степень их относительной важности. Поэтому все режимы работы арбитра шины базируются на понятии приоритета — в любой момент времени одно из ведущих шины имеет приоритет над другими.

Арбитр шины имеет около 20 входных и выходных управляющих сигналов. Для понимания способов учета приоритетов ведущих шины достаточно рассмотреть следующие сигналы:

$\overline{\text{BREQ}}$ — запрос шины. Выходной сигнал (при использовании способа параллельного учета приоритетов), который генерирует арбитр для запроса использования системной шины;

$\overline{\text{BPRN}}$ — вход приоритета шины. Сигнал, подаваемый в арбитр шины для сообщения ему о том, что он может использовать системную шину по очередному спаду сигнала

шинной синхронизации (BCLC), свидетельствует о том, что данный арбитр имеет приоритет над всеми арбитрами, запрашивающими системную шину. Снятие сигнала $\overline{\text{BPRN}}$ сообщает арбитру, что он потерял приоритет;

$\overline{\text{BPRO}}$ — выход приоритета шины. Данный сигнал используется при последовательном способе учета приоритетов, когда выход $\overline{\text{BPRO}}$ некоторого арбитра подается на вход $\overline{\text{BPRN}}$ следующего арбитра с меньшим приоритетом;

$\overline{\text{BUSY}}$ — занятость. Двунаправленная линия системной шины, к которой подключаются соответствующие выходы (типа открытый коллектор) всех арбитров шины и сигнал на которой сообщает арбитру о доступности шины (если $\overline{\text{BUSY}}$=1, шина доступна). Когда арбитр использовал шину, он снимает сигнал $\overline{\text{BUSY}}$, выключая транзистор выходного каскада, что позволяет другим арбитрам получать доступ к системной шине.

При параллельном способе учета приоритетов, показанном на рис. 4.18, имеются отдельные линии запроса шины $\overline{\text{BREQ}}$ для каждого арбитра, находящегося в системе. Сигналы $\overline{\text{BREQ}}$ подаются в шифратор приоритетов, который формирует двоичный адрес (номер) активной линии $\overline{\text{BREQ}}$ с наибольшим приоритетом. Полученный адрес подается на дешифратор для выбора соответствующей линии $\overline{\text{BPRN}}$, активный сигнал на которой возвращается в запрашивающий арбитр шины с наибольшим приоритетом. Этот арбитр разрешает своему ведущему доступ к системной шине, как только она будет доступна ($\overline{\text{BUSY}}$=1). Одновременно с доступом к шине выбранный арбитр формирует сигнал $\overline{\text{BUSY}}$=0, отключая все остальные арбитры от шины.

Способ последовательного учета приоритетов реализуется путем сцепления (организации приоритетной цепочки) арбитров шины. При этом выход $\overline{\text{BPRO}}$ арбитра с бо́льшим приоритетом подключается на вход $\overline{\text{BPRN}}$ соседнего арбитра с меньшим приоритетом. На рис. 4.19 приоритеты арбитров убывают сверху вниз. Если арбитр шины не запрашивает шину, он пропускает сигнал $\overline{\text{BPRN}}$ на выход $\overline{\text{BPRO}}$. Когда арбитр запрашивает шину, ему разрешается доступ к ней только при условии, что сигнал $\overline{\text{BPRN}}$=0 (и, конечно, если $\overline{\text{BUSY}}$=1).

Способ циклического учета приоритетов аналогичен способу параллельного анализа приоритетов, но приоритеты

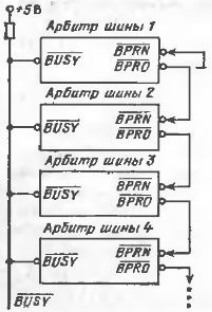


Рис. 4.19. Последовательный учет приоритетов

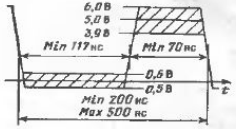


Рис. 4.20. Сигналы синхронизации

арбитров динамически переназначаются. Шифратор приоритетов заменяется более сложной схемой, которая назначает только что обслуженному арбитру низший приоритет и соответственно корректирует приоритеты остальных арбитров. При этом способе ранжирования модулей все арбитры имеют примерно одинаковые шансы на использование системной шины.

Из рассмотренных способов учета приоритетов предпочтение обычно отдается первому, так как он допускает использование значительного числа арбитров шины и не требует сложных дополнительных схем.

4.6. Синхронизация

Для правильной работы МП К1810 всегда необходимы сигналы синхронизации CLC, к которым предъявляются довольно жесткие требования. Максимальная длительность фронтов составляет 10 нс, диапазон напряжения низкого уровня составляет $-0{,}5 \div +0{,}6$ В, а диапазон напряжения высокого уровня $3{,}9 \div (E_{пит}+1)$ В. Максимальная частота сигналов синхронизации 5 МГц. Так как в микропроцессоре имеются динамические элементы, не допускается уменьшение частоты синхронизации ниже 2 МГц. Следовательно, осуществить работу МП по командам или тактам путем запрещения синхронизации невозможно. По мере приближения частоты сигналов синхронизации к максимальной коэффициент заполнения импульсов должен приближаться к $^1/_3$. Вид сигналов синхронизации показан на рис. 4.20.

Формирование сигналов синхронизации с оптимальным коэффициентом заполнения и требуемыми уровнями напряжений обеспечивает микросхема генератора синхронизации (или генератора тактирующих импульсов). Частота выходных сигналов CLC генератора определяется либо частотой внешнего сигнала, имеющего коэффициент заполнения $^1/_2$, либо подключаемым к генератору кварцевым резонатором. При этом частота входных сигналов и основная частота кварца должны быть в 3 раза выше требуемой частоты синхронизации МП, так как выходной сигнал CLC получается путем деления внутренним счетчиком исходной частоты на 3.

Когда общий источник применяется для управления несколькими генераторами, распределенными по системе, к каждому из них должна подходить отдельная линия от источника. Для минимизации помех такие линии реализуются витыми парами, причем их проводники земли соединяют земли источника и приемника.

Чтобы уменьшить перекос сигналов синхронизации, линии связи со всеми генераторами должны иметь одинаковую длину. Способ образования основного источника для нескольких генераторов показан на рис. 4.21. Сигналы на выходе OSC генератора имеют частоту, определяемую резонансной частотой кварца; они используются в качестве внешнего источника (EFI — вход внешней частоты) всех других генераторов в системе.

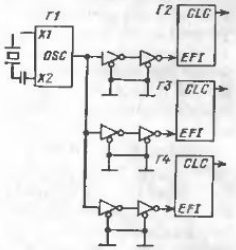


Рис. 4.21. Подключение нескольких генераторов

Выходной сигнал OSC представляет собой инверсию сигнала задающего генератора. Поэтому сигнал выхода OSC одного генератора нельзя подавать на вход EFI другого генератора, если их сигналы CLC подаются на два микропроцессора, которые должны работать синхронно. Разброс времен задержки сигналов EFI и CLC в различных генераторах может достигать 35—45 нс. Но если все генераторы имеют одинаковое напряжение питания и работают в одинаковых температурных условиях, этот разброс удается уменьшить до 15—25 нс.

Так как состояние внутреннего счетчика при включении питания не определено, в генераторе предусмотрен вход внешней синхронизации CSYNC, обеспечивающий синхро-

низацию генератора с внешним событием. При подаче на вход CSYNC напряжения высокого уровня на выходе CLC устанавливается также напряжение высокого уровня. Когда сигнал на входе CSYNC имеет низкий уровень, следующий положительный импульс от источника частоты запускает формирователь импульсов синхронизации CLC. Сигнал CSYNC должен быть активным минимум два периода сигналов источника. Необходима также схема синхронизации сигнала CSYNC с сигналами внешней частоты, если в системе применяется несколько генераторов.

Один генератор не следует применять для формирования сигналов CLC нескольких процессоров, которые не разделяют мультиплексную шину. Это объясняется тем, что генератор синхронизирует сигнал готовности RDY и может сформировать этот сигнал только для одного процессора. Если локальную шину разделяют несколько процессоров, например МП и арифметический сопроцессор, их следует синхронизировать одним и тем же сигналом, чтобы ускорить операции передачи управления шиной. В данном случае шиной всегда будет управлять только один процессор, что позволяет использовать общий сигнал готовности.

4.7. Сброс микропроцессора

Приведение МП в начальное состояние осуществляется сигналом высокого уровня на входе сброса CLR. Минимальная продолжительность сигнала CLR составляет четыре такта синхронизации, но при включении напряжения питания импульс сброса должен быть не менее 50 мкс. Так как в МП производится синхронизация сигнала CLR импульсами CLC, сигнал внутреннего сброса активен минимум через один такт после действия внешнего сигнала сброса. Запросы на входах NMI и $\overline{RQ}/\overline{E}$, возникающие во время внутреннего сброса, не подтверждаются. Запросы HLD в минимальном режиме или $\overline{RQ}/\overline{E}$ в максимальном режиме, появляющиеся сразу после внутреннего сброса, подтверждаются до выборки первой команды.

Состояния линий МП после сброса приведены в табл. 4.1.

Когда МП обнаруживает сигнал сброса, мультиплексная шина адреса/данных/состояния переводится в высокоимпедансное состояние. Другие тристабильные сигналы переводятся в неактивное состояние через один интервал низкого уровня сигнала CLC, а затем переходят в высокоимпедансное состояние. В минимальном режиме сигналы STB и HLD

Таблица 4.1. Состояния внешних линий МП после сигнала сброса

Линия	Состояние
AD, A/S, $\overline{BHE}$/ST7	Высокоимпедансное
$\overline{ST2}$ (M/$\overline{IO}$), $\overline{ST1}$ (OP/$\overline{IP}$), $\overline{ST0}$ ($\overline{DE}$), R, $\overline{LOCK}$ ($\overline{W}$), $\overline{INTA}$	Переводится к высокому уровню, затем высокоимпедансное
STB, HLDA, QS0, QS1	Низкий уровень
$\overline{RQ}/\overline{E0}$, $\overline{RQ}/\overline{E1}$	Высокий уровень

становятся неактивными, но не переводятся в высокоимпедансное состояние. В максимальном режиме сигналы на линиях $\overline{RQ}/\overline{E}$ сохраняются неактивными, а сигналы состояния очереди показывают отсутствие действий. Сигналы QS не фиксируют сброс очереди, поэтому все контролирующие очередь схемы должны сбрасываться при действии системного сброса.

Контроллер шины на входах $\overline{ST}$ имеет внутренние резисторы, подключенные к источнику +5 В; высокий уровень на этих входах (когда МП перевел шину в высокоимпедансное состояние) заставляет контроллер интерпретировать последовательность сброса как пассивное состояние. Если сброс возникает во время цикла шины, переход сигналов $\overline{ST}$ в пассивное состояние закончит цикл шины.

В мультипроцессорных системах сигнал сброса должен быть общим для всех процессоров и должен удовлетворять требованиям, предъявляемым к сигналу сброса МП. Кроме того, сигнал системного сброса необходимо подавать на вход инициализации арбитра шины.

Сигнал сброса для МП формирует микросхема генератора синхронизации, имеющая на входе сброса триггер Шмитта с гистерезисными свойствами. Чтобы автоматически формировать сигнал сброса при включении напряжения питания, на этот вход подключается RC-цепочка, постоянная времени которой рассчитывается из условия формирования импульса сброса с длительностью не менее 50 мкс. На этот же вход обычно подключается кнопка ручного системного сброса (параллельно конденсатору). Генератор синхронизирует вход сброса с сигналом CLC и вырабатывает сигнал CLR. Действие сброса не влияет на схемы синхронизации в генераторе.

4.8. Сигнал готовности

Сигнал готовности RDY применяется в системе для адаптации памяти и периферийных устройств, которые не могут передавать информацию с максимальной пропускной способностью шины МП. Для введения состояния ожидания T_w в цикле шины сигнал RDY должен быть неактивным (иметь низкий уровень) к концу такта T_2. Чтобы предотвратить введение состояния T_w, сигнал RDY должен быть активным внутри указанного времени установления до положительного перехода синхронизации в такте T_3. В зависимости от состава и характеристик системы имеются два варианта реализации готовности.

Классический вариант реализации готовности — заставить систему «нормально не быть готовой». Когда выбранное устройство воспринимает сигнал $\overline{R}$, $\overline{W}$ или $\overline{INTA}$ и имеет достаточно времени для выполнения требуемых действий, оно активизирует сигнал RDY, позволяя микропроцессору закончить цикл шины. Такая реализация характерна для большой мультипроцессорной системы и системы, в которой задержки распространения, задержки доступа к шине и характеристики устройств приводят к обязательному замедлению работы системы. Для достижения максимальной производительности системы устройства, которые могут работать без введения состояний ожидания, должны возвращать сигнал RDY в заданном интервале. Невозможность отреагировать своевременно приведет только к введению одного или нескольких состояний ожидания T_w.

Второй вариант заключается в реализации «нормально готовой» системы с предположением о том, что все устройства работают с максимальным быстродействием микропроцессора. Устройства, не удовлетворяющие этому требованию, должны запрещать RDY к концу такта T_2, чтобы гарантировать введение состояния ожидания. Такая реализация характерна для небольших однопроцессорных систем и не требует схем, необходимых для управления сигналом готовности. Если устройство, требующее введения состояния ожидания, не сформирует сигнал RDY к концу такта T_2, это приведет к преждевременному окончанию цикла шины. Следовательно, при использовании этого варианта необходимо тщательно проанализировать временные диаграммы устройств.

В зависимости от принятого варианта реализации системы МП имеет два временных требования к сигналу готовности. В «нормально готовой системе» для введения состояния ожидания сигнал RDY должен быть запрещен минимум за 120 нс до нарастающего фронта сигнала CLC в такте T_3 (при частоте синхронизации 5 МГц). Для гарантии правильной работы МП сигнал RDY не должен изменять уровень при действии низкого уровня сигнала CLC в такте T_3. В «нормально не готовой системе» во избежание состояний ожидания сигнал RDY должен быть активным минимум за 35 нс до нарастающего фронта сигнала синхронизации в такте T_3.

Для формирования стабильного сигнала RDY, который удовлетворяет временным требованиям, сигналы готовности от устройств подаются в микросхему генератора синхронизации. Она имеет два входа готовности (RDY1 и RDY2) и формирует синхронизированный выходной сигнал готовности, который подается в микропроцессор.

Примечание. В «нормально не готовой» системе программист не должен помещать исполняемые команды в последние 6 байт физической памяти. Так как МП выбирает команды с опереженем, он может попытаться обратиться к несуществующей памяти при выполнении программы, находящейся в конце физической памяти. Если обращение к несуществующей памяти не разрешит сигнал готовности RDY, система «зависнет» в бесконечном ожидании.

4.9. Интерфейс ввода-вывода

Микропроцессор К1810 может взаимодействовать с 8- и 16-битными периферийными устройствами, используя либо специальные команды ввода-вывода, либо ввод-вывод, отображенный на память. Допускается подключать периферийные устройства к локальной шине микропроцессора или к буферированной системной шине.

Команды ввода-вывода позволяют размещать периферийные устройства в отдельном адресном пространстве, а ввод-вывод, отображенный на память, дает возможность применять для ввода-вывода любую команду, связанную с обращением к памяти. Программист осуществляет доступ к отдельному адресному пространству ввода-вывода только с помощью команд IN и OUT, которые передают данные между устройством и аккумуляторами AX (слово) и AL (байт). Первые 256 байт пространства ввода-вывода прямо адресуются в командах ввода-вывода, а все 64К байт — косвенно через регистр DX. Последний способ особенно удобен в процедурах, предназначенных для обслуживания несколь-

ких идентичных периферийных устройств, так как адрес нужного устройства можно передать процедуре как параметр.

Периферийные устройства, в которых основным элементом данных является байт (их можно назвать байтными или 8-битными устройствами), допускается подключать к старшей и младшей половинам шины данных, равномерно распределяя нагрузку на шину. Если устройство подключено к линиям D15—D8, все назначаемые ему адреса ввода-вывода должны быть нечетными (A0=1), а если оно подключено к линиям D7—D0, четными (A0=0). Так как для конкретного устройства сигнал A0 всегда будет иметь логические 0 или 1, его нельзя использовать для выбора регистров в данном устройстве. Когда устройству на старшей и младшей половинах шины назначены адреса, различающиеся только битом A0, в дешифрировании сигналов, определяющих выбор устройства (сигнал DS), должны участвовать сигналы $\overline{BHE}$ и A0. Это необходимо для того, чтобы запись в одно устройство не приводила к ошибочной записи в другое. Несколько способов формирования сигналов DS для периферийных устройств представлены на рис. 4.22, где входы E являются входами разрешения работы дешифратора, а входы A_{2-0} — информационными входами дешифратора.

На рис. 4.22, *а* показаны отдельные дешифраторы для генерирования сигналов выбора DS байтных периферийных устройств с четными и нечетными адресами. Если выполняется передача слова в устройство с четным адресом, то выбирается и устройство с соседним нечетным адресом. Этим обеспечивается доступ к устройствам отдельно при передачах байт или одновременно, как к 16-битному устройству для передач слов. В способе, показанном на рис. 4.22, *б*, ограничивается выбор периферийного устройства для передачи байт, но передача слова по нечетному адресу заставит МП выполнить две передачи байт, которые дешифратор не обнаружит. В третьем способе, показанном на рис. 4.22, *в*, применяется один дешифратор для генерирования сигналов DS устройств с четными и нечетными адресами в пределах байт, но при передаче слова по четному адресу выбирается только 8-битное устройство с четным адресом.

Если в системе используется более 256 байт пространства ввода-вывода или ввод-вывод, отображенный на память, могут потребоваться дополнительные дешифраторы, кроме показанных на рис. 4.22. Для этого можно применить

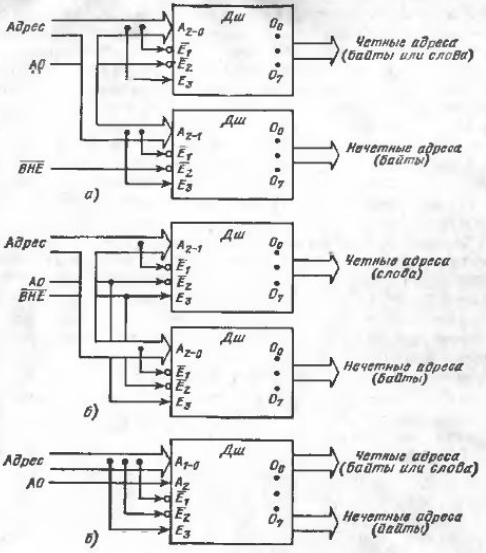


Рис. 4.22. Способы выбора устройств

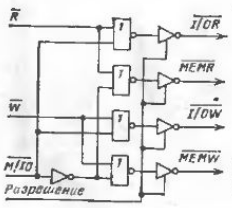


Рис. 4.23. Формирование отдельных сигналов для памяти и ввода-вывода

ТТЛ-дешифраторы или программируемые постоянные запоминающие устройства (ППЗУ). Биполярные ППЗУ несколько медленнее дешифраторов, но обеспечивают полное дешифрирование в одном корпусе и допускают введение нового ППЗУ при реконфигурации системной карты ввода-вывода без модификаций схемной платы или монтажа.

С целью эффективного использования шины и упрощения выбора устройств 16-битным периферийным устройствам следует назначать четные адреса. Для гарантии выбора устройства только для передач слов в выборе устройства должны участвовать сигналы A0 (A0=0) и $\overline{BHE}$ ($\overline{BHE}$=0).

Поскольку в минимальном режиме МП формирует общие сигналы считывания и записи для памяти и ввода-вывода, при перекрытии их адресных пространств выбор устройства должен определяться сигналом M/$\overline{IO}$. Это ограничение на формирование сигналов DS можно снять, если:

адреса памяти и ввода-вывода в системе не перекрываются;

весь ввод-вывод отображен на память;

из сигналов $\overline{R}$, $\overline{W}$ и M/$\overline{IO}$ формируются отдельные управляющие сигналы считывания/записи памяти и ввода-вывода, как показано на рис. 4.23.

В системе, в которой МП работает в максимальном режиме, контроллер шины вместо сигнала M/$\overline{IO}$ генерирует отдельные управляющие сигналы для памяти и ввода-вывода. Периферийные устройства размещаются в пространствах памяти или ввода-вывода путем подключения их к линиям соответствующих сигналов.

ЗАКЛЮЧЕНИЕ

В книге рассмотрены архитектура и программирование на языке ассемблера нового 16-битного микропроцессора К1810ВМ86. Безусловно, этому микропроцессору с его широкими возможностями обеспечено применение в тех прикладных областях, где операционных средств и(или) производительности 8-битных микропроцессоров недостаточно. Можно сказать, что микропроцессор К1810ВМ86 поднимает проектирование микросистем на новый, более высокий уровень, превращая в реальность то, что казалось невозможным всего несколько лет назад.

Изучение внутренней организации, режимов адресации и системы команд микропроцессора, а также элементов языка ассемблера стимулируется несколькими причинами. Во-первых, знание этого материала позволит разрабатывать законченные ассемблерные программы, а также эффективные ассемблерные фрагменты критических секций программ на языках высокого уровня. Во-вторых, программисту на языке ассемблера доступны все ресурсы той конкретной системы, для которой создается программа. В-третьих, изучение этих вопросов поможет разобраться в особенностях применения и программирования новых микросхем, рассчитанных на работу с микропроцессором К1810ВМ86. В первую очередь это относится к арифметическому сопроцессору К1810ВМ87, программирование которого осуществляется на языке ассемблера и который может работать только совместно с микропроцессором К1810ВМ86, выступающим в качестве центрального процессора. Наконец, в разработках новых микропроцессоров наряду с введением дополнительных возможностей сохраняется совместимость с микропроцессором К1810ВМ86 (ради мобильности имеющегося программного обеспечения) и знание его поможет при знакомстве с новыми изделиями.

Автор надеется, что книга окажется полезной для инженерно-технических работников и студентов вузов, специализирующихся по вычислительной технике, в деле освоения микропроцессора К1810ВМ86 и применения его в разработках новых микросистем. Кроме того, она будет содействовать в освоении нескольких профессиональных персональных компьютеров, в которых применяется микропроцессор К1810ВМ86.

Список литературы

1. **Уокерли Д.** Архитектура и программирование микро-ЭВМ. Т. 1. М.: Мир, 1984. Т. 2. М.: Мир, 1984.

2. **Григорьев В. Л.** Программное обеспечение микропроцессорных систем. М.: Энергоатомиздат, 1983.

3. **An Introduction** to ASM-86. — Intel Corp., 1981.

4. **MCS-86** assembler language reference manual. Intel Corp., 1982.

5. **Rector R., Alexy G.** The 8086 book. McGraw-Hill, 1980.

6. **Morse S. P.** Intel microprocessors — 8008 to 8086//Computer. 1980. Vol. 26. № 10.P.42—60.

7. **Morgan C. L., Waite M.** 8086/8088 16-bit microprocessor primer. BYTE/McGraw-Hill, 1982.

8. **Patstone W.** 16-bit microprocessors benchmarks — an update with explanations.//EDN. 1981, Vol. 26. № 18. P. 169—203.

9. **Willen D. C., Krantz J. I.** 8088 assembler language programming: the IBM PC. — Howard W. Sams, 1983.

10. **Grappel R. D., Hemenway J. E.** A tale your microprocessor: benchmarks quantify performance//EDN. 1981. Vol. 26. № 17. P. 179—265.

11. **L. J. Scanlon.** 8086/88 assembly language programming. Prentice-Holl, 1984. — 213 p.

ОГЛАВЛЕНИЕ

ПРОИЗВОДСТВЕННОЕ ИЗДАНИЕ

Вячеслав Леонидович Григорьев

Программирование однокристальных микропроцессоров

Редактор **В. В. Сташин**
Редактор издательства **А. И. Гускова**
Художественный редактор **Т. А. Дворецкова**
Технический редактор **В. В. Хапаева**
Корректор **М. Г. Гулина**
ИБ № 2123

Сдано в набор 11.12.86. Подписано в печать 27.05.87. Т-12141. Формат 84×108¹/₃₂. Бумага типографская № 2. Гарнитура литературная. Печать высокая. Усл. печ. л. 15,12. Усл. кр.-отт. 15,12. Уч.-изд. л. 16,56. Тираж 60 000 экз. Заказ № 736. Цена 1 р. 20 к.

Энергоатомиздат, 113114, Москва, М-114, Шлюзовая наб., 10

Владимирская типография Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли 600000 г. Владимир, Октябрьский проспект, д. 7