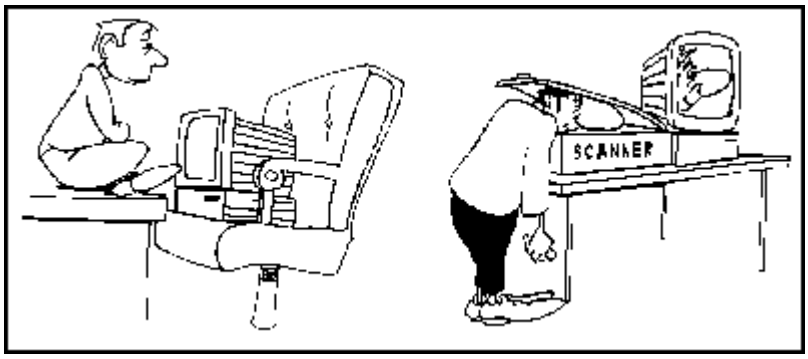
Е.А. Еремин

Учебный компьютер E97

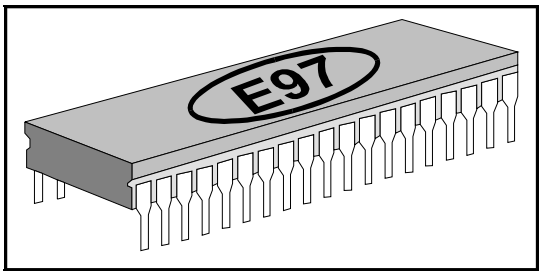
(данный PDF-файл сформирован автором из материалов своей книги "**Как работает современный компьютер**". Пермь: Изд-во ПРИПИТ, 1997. 176 с.)



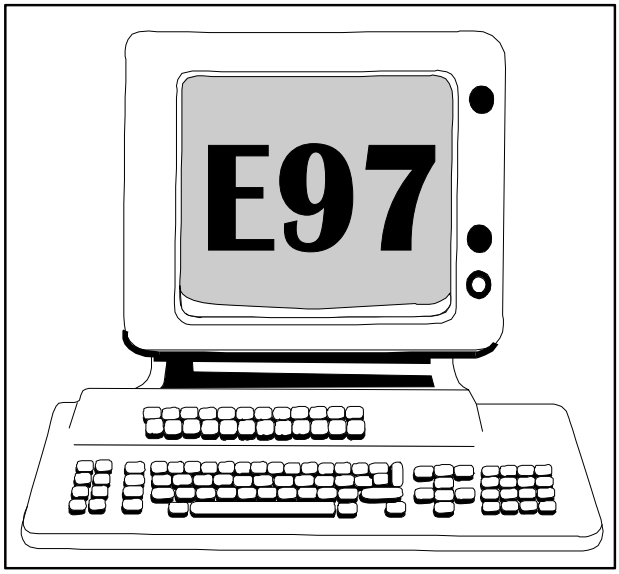
Учебный компьютер "E97"

По общепринятой методике, изучение основных принципов устройства ЭВМ ведется на базе упрощенного рассмотрения одной из реально существующих ЭВМ. Неудобством такого подхода является необходимость постоянно что-то недоговаривать и пропускать, причем порой эти пропуски заметны даже обучаемым.

В данном пособии автором предложен альтернативный способ изложения материала. Разработана специальная



учебная модель микропроцессорной ЭВМ четвертого поколения "E97", в которую включены наиболее характерные черты работы современных компьютеров. Модель не повторяет полностью ни один реальный компьютер, хотя дает достаточно полное представление о его устройстве. В "E97" нет излишних технических деталей, он достаточно прост и содержит только то, что необходимо для изучения.



В пособии последовательно и систематически описывается учебная модель "E97". Сначала подробно объясняется устройство этой модели, система ее команд и принципы работы с памятью и внешними устройствами. Обсуждение завершается методическими рекомендациями по применению "E97", содержащими разбор 16 нетривиальных задач и список из 50 примерных заданий для закрепления материала.

1. "E97" - первая учебная модель микрокомпьютера

Если рассмотреть использующиеся в отечественных учебниках модели ЭВМ ("Кроха", "Малютка", "Нейман" и др.), то окажется, что они во многом не соответствуют устройству современных компьютеров. В качестве наиболее существенных недостатков можно выделить следующие.

1) Учебные ЭВМ никак не учитывают основополагающие принципы устройства современного четвертого поколения ЭВМ. Наоборот, все они навивают ностальгические воспоминания о "старых добрых временах" машин 1-2 поколения.

2) Ни одна из моделей не работает с *нечисловыми* данными; нельзя продемонстрировать даже обработку такого важного и широко распространенного вида информации, как текст.

3) Во всех современных ЭВМ давно используется байтовая организация памяти, но в учебных ЭВМ это либо совсем не отражается, либо отражается лишь частично, как, например, в компьютере "Нейман". Во всяком случае, везде команды и данные для простоты модели предполагаются одинаковой длины, что не соответствует действительности уже начиная с ЕС ЭВМ.

4) Только в модели "Малютка" делается попытка (хотя и не очень успешная) показать, как процессор может обрабатывать массивы информации – а в этом состоит одно из главных предназначений ЭВМ! Данная ограниченность моделей является принципиальной: ни в одну из них не заложены методы адресации, позволяющие организовать доступ к последовательно расположенным однородным данным.

5) Ни одна из моделей не позволяет по-настоящему объяснить ученикам, как процессор работает с внешними устройствами. Имеющиеся табло и достаточно неопределенные устройства ввода слишком условны и к тому же не имеют ничего общего с периферийными устройствами современного компьютера.

6) Существующие модели игнорируют одно из важнейших достижений программирования – переход с возвратом, т.е. вызов подпрограммы. Иными словами, мы не показываем обучаемому способ реализации фундаментальной алгоритмической структуры, используемой практически в каждой программе. Как можно без этого говорить о методах обработки информации на ЭВМ! По-видимому, отсутствие возможности перехода к подпрограммам связано с нежеланием авторов моделей "раскрыть секрет" работы стека. По-моему, стек является одним из фундаментальных понятий в информатике и дать хотя бы начальное представление о нем совершенно необходимо. К тому же при внимательном рассмотрении принцип стека оказывается ненамного сложнее разборки и сборки детской пирамидки.

Пожалуй, достаточно критики. Подчеркнем, что ее главная цель состоит не в том, чтобы показать, как плохи существующие модели, а *указать пути* их необходимого улучшения. Перейдем теперь к конструктивной части, т.е. к устранению перечисленных недостатков.

В свете всех сформулированных здесь положений, автор поставил задачу **создать свободную от описанных выше ограничений и недостатков учебную модель современного компьютера.** Это будет первая модель, соответствующая компьютеру ***четвертого поколения***. Сразу спешу заметить, что не стоит бояться ставить широкие и далеко идущие цели при разработке модели – в конце концов учитель не обязан использовать абсолютно все ее возможности.

Следующие разделы как раз и посвящаются описанию принципиально новой учебной ЭВМ. Для удобства изложения требуется дать ей какое-то имя. После некоторых размышлений, я остановился на коротком и "скромном" названии "E97", где буква символизирует автора модели, а цифры обозначают год ее создания. Итак, переходим к изучению учебного компьютера "E97".

2. Внутренняя структура компьютера "E97" и его процессора

Наконец-то все предварительное обсуждение завершено, и мы приступаем к знакомству с моделью микрокомпьютера "E97". Начнем с его функциональной схемы, приведенной на рис. 1.

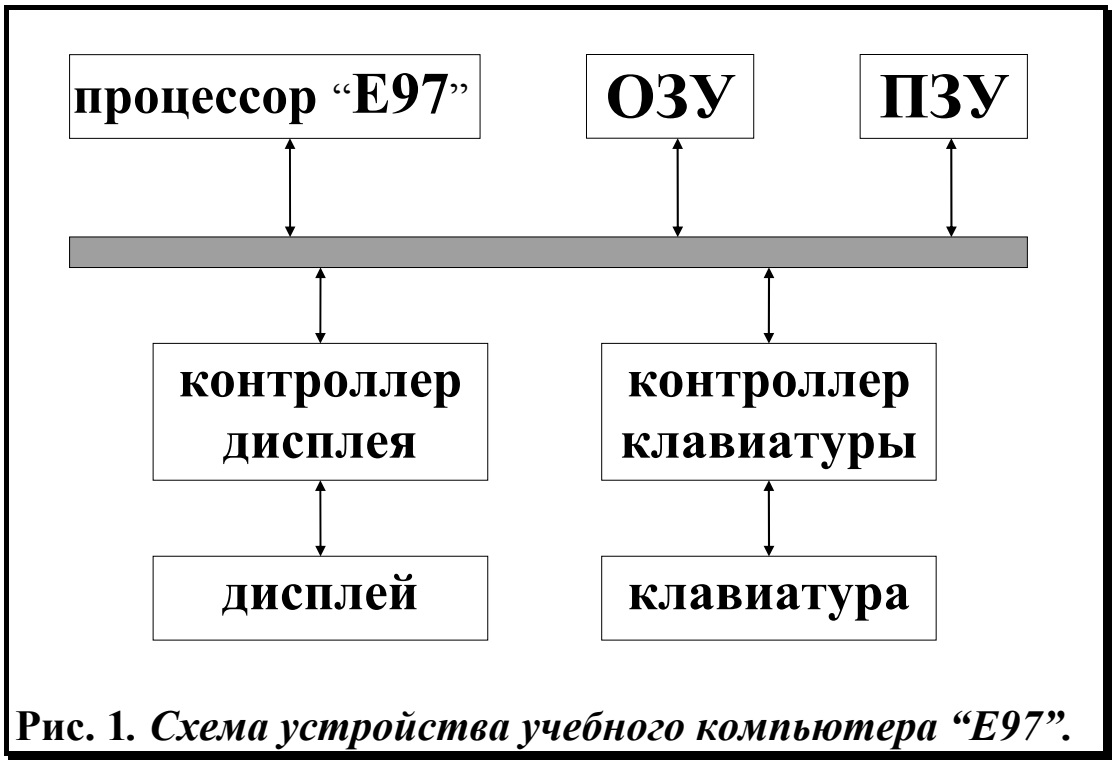


Рис. 1. ***Схема устройства учебного компьютера “E97”.***

Как следует из рисунка, в состав учебного микрокомпьютера входят следующие устройства: центральный процессор, память двух видов – ОЗУ и ПЗУ, а также два наиболее важных внешних устройства – клавиатура для ввода информации и дисплей для вывода; оба устройства, как принято в современных ЭВМ, подключены через обеспечивающие согласование контроллеры.

Главным блоком компьютера служит 16-разрядный процессор "E97", способный работать как с двухбайтовыми словами, так и с отдельными байтами. Впервые в рамках учебной модели реализована возможность оперировать с данными разной длины. Познакомившись с тем, как "E97" обрабатывает разные типы информации, читатель легко

сможет в будущем обобщить логику "один байт или много" на случай большей разрядности процессора.

В процессоре имеются внутренние регистры памяти, при помощи которых реализован метод косвенной адресации к ОЗУ. Очевидно, что полное 16-разрядное адресное пространство "E97" позволяет напрямую адресовать до 64 Кбайт памяти; для учебной ЭВМ это более чем достаточно. Поэтому реально существующей памяти будут соответствовать лишь некоторые диапазоны адресов (рис. 2). Для первой программной реализации модели "E97" приняты следующие значения констант:

HiRAM= 100,
LoROM=4000,
HiROM=4180.

Деление ПЗУ на две части обсудим позднее.

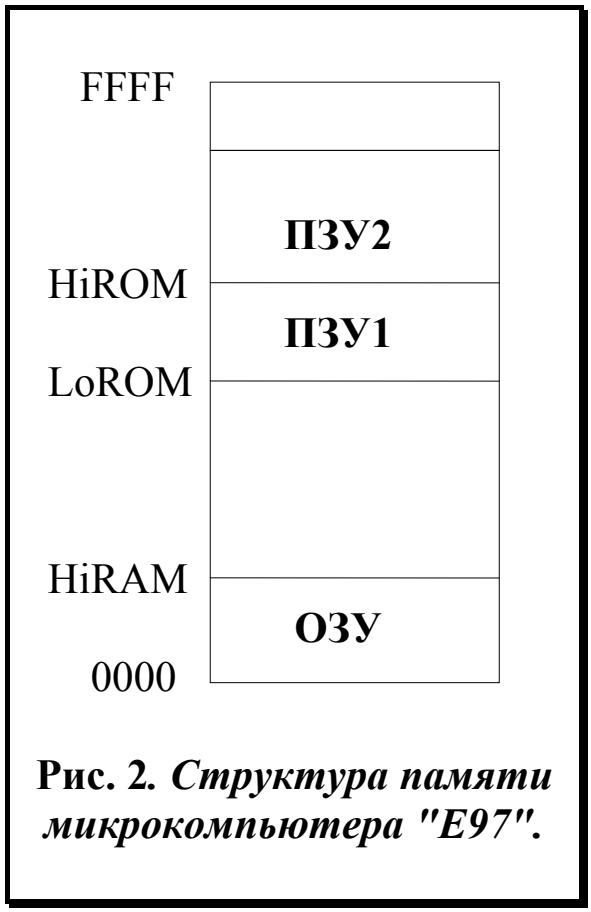


Рис. 2. ***Структура памяти микрокомпьютера "E97".***

Заметим, что ситуация, когда не все адресное пространство процессора физически заполнено ОЗУ отнюдь не так экзотична, как может показаться на первый взгляд, и может иметь место на практике.

Как-то плавно и незаметно мы перешли на обсуждение устройства памяти. Прежде всего отметим, что в "E97", как и в реальном компьютере, существует память двух видов – **оперативная** (ОЗУ) и **постоянная** (ПЗУ). В первой хранится текущая информация (т.е. программа и данные) по решаемой задаче, причем она может как считываться, так и записываться. Во второй, предназначенной только для считывания, содержатся разработанные при проектировании ЭВМ подпрограммы наиболее важных и часто используемых действий, среди

которых важное место занимают алгоритмы обмена с внешними устройствами.

Видеопамять в "E97" мы разместим в контроллере дисплея и для простоты первоначальной модели будем считать, что она не входит в адресное пространство процессора. Доступ к видеоОЗУ осуществляется путем обращения к внешнему устройству и более подробно будет рассматриваться позднее.

Минимальной адресуемой ячейкой памяти в современных ЭВМ является байт. Все байты в памяти "E97" пронумерованы и их 16-разрядные номера находятся в пределах от 0000 до FFFF. Реальные адреса должны попадать в выделенные на рис. 2 области. При обращении к любому другому адресу происходит аварийное прекращение выполнения программы.

Байты в памяти могут объединяться в слова, которые для нашего компьютера состоят из двух соседних байтов (у современных процессоров обычно из четырех, но для понимания основных принципов это не очень существенно). По традиции примем, что слово адресуется наименьшим из номеров байтов, причем этот адрес соответствует младшему байту слова. Для удобства изображения содержимого ОЗУ на экране дисплея будем считать, что адрес слова *всегда является четным* – строго говоря, это не для всех процессоров обязательно. При попытке обратиться к слову с нечетным адресом наш учебный процессор будет прекращать выполнение программы, фиксируя тем самым ошибку обучаемого.

Для наглядности проиллюстрируем все эти важные положения примером:

40	41	42	43
00	FF	34	12

представление в байтах

(41)	FF	00	40
(43)	12	34	42

представление в словах

Слово с адресом 40 равняется FF00, а 42 - 1234; адреса 41 и 43 для слов запрещены. Обратите внимание на то, что байт

40 имеет нулевое значение, а байт с адресом 41 – значение FF. Иначе говоря, байты слова хранятся в памяти "задом наперед" – *сначала младший, а затем старший*. Именно так хранится слово во многих реальных компьютерах, в том числе и в IBM PC.

ПЗУ есть существенная особенность модели "E97", так что имеет смысл рассмотреть его подробно. Наличие в модели ПЗУ дает целый ряд преимуществ, важнейшими из которых являются следующие.

Во-первых, можно показать обучаемым, что в современных ЭВМ не всякое элементарное действие реализуется в виде команды процессора – многие из них выполняются по программе. В "E97", например, это нахождение остатка от деления, вычисление модуля, перевод целого числа из двоичной системы в десятичную. Наличие таких подпрограмм освобождает модель от необходимости вводить в систему команд учебного процессора разные нестандартные и несуществующие в реальных процессорах команды типа "вывод числа на экран в десятичной форме" и ей подобных.

Во-вторых, наличие ПЗУ позволяет не конкретизировать алгоритм работы с внешними устройствами на уровне системы команд процессора, а перенести их в ПЗУ. Именно так делается и в настоящих компьютерах: ПЗУ такого рода носит специальное название **BIOS** (от английского Basic Input Output System - **базовая система ввода-вывода**) и служит для обслуживания конкретных типов внешних устройств, входящих в состав машины. BIOS имеет стандартные точки входа, к которым и обращается все программное обеспечение при необходимости произвести обмен информации с внешним устройством.

Кроме описанного выше, использование ПЗУ приводит к появлению в модели дополнительных преимуществ методического плана. Оформив, как это сделано в "E97", ПЗУ в виде текстового файла, который загружается при старте программы, мы получаем замечательную возможность легко модифицировать работу нашей модели без пе-

рекомпиляции программы. По мнению автора, стремление разработать полезную стандартную подпрограмму, которую учитель затем добавит к ПЗУ, может служить мощным стимулом для некоторых наиболее способных учеников.

Наконец, подробно прокомментированный текстовой файл с ПЗУ, используемый в "E97", может дополнительно служить хорошим наглядным примером того, как нужно составлять программы. Можно даже порекомендовать дать индивидуальные задания наиболее интересующимся ученикам разобрать самостоятельно те или иные подпрограммы в ПЗУ. Имеется определенная категория людей, которым деятельность такого рода очень нравится.

Завершая обсуждение ПЗУ, отметим еще одну деталь. В нашей учебной ЭВМ оно условно делится на две части, названные на рис. 2 ПЗУ1 и ПЗУ2. ПЗУ1 доступно для изучения и представляет собой настоящие программы в кодах процессора "E97". В его состав в первой версии программной реализации включены следующие полезные подпрограммы: определение остатка от деления, вычисление модуля, преобразование целого числа в строку с его десятичным представлением (подготовка к выводу на экран), вывод целого числа, вывод строки символов, вывод символа, перепись массива и несколько подпрограмм-заготовок для реализации в будущем компилятора Паскаля. Кроме того, в ПЗУ1 находятся коды перехода к функциям ввода в ПЗУ2. Некоторые из перечисленных подпрограмм будут позднее подробно анализироваться в качестве примеров. Полная распечатка текстового файла с содержимым ПЗУ1 приведена в приложении.

Вы, наверное, обратили внимание, что заполнен не весь объем ПЗУ1: чтобы убедится в этом, сравните приведенные на рис. 2 константы LoROM и HiROM с максимально возможным адресом в распечатке. Оставшееся "свободное" место предназначено для ваших упражнений. Порекомендую только не располагать свои собственные программы сразу после окончания имеющихся в ПЗУ программ:

лучше оставьте место под возможные авторские расширения модели.

Другая часть долговременной памяти - ПЗУ2, напротив, "скрыта" от просмотра. Определены лишь входные точки и их назначение; в основном это операции ввода. Подчеркнем, что деление на ПЗУ1 и ПЗУ2 является чисто "техническим" приемом и не имеет никаких аналогий в реальных ЭВМ. Наличие "закрытого" ПЗУ2 просто позволяет легче и быстрее осуществить программный имитатор учебного микрокомпьютера. Возможно, в последующих версиях ПЗУ2 будет конкретизировано и "рассекречено" путем перевода на язык "E97". Строго говоря, если вам или вашим ученикам по какой-то причине очень понравится программировать для нашей учебной модели, можно попытаться не ждать новых версий, а сделать это самостоятельно (хотя бы для нескольких подпрограмм).

Таким образом, введение в модель "E97" постоянного запоминающего устройства делает ее более "открытой" для пользователей и дает им потенциальную возможность расширения "встроенного" программного обеспечения.

У нас остались нерассмотренными лишь контроллеры внешних устройств. В учебном микрокомпьютере они фигурируют в виде портов ввода-вывода. Каждому из устройств соответствует два таких порта: **регистр данных** и **регистр состояния**. Через первый происходит обмен информацией, а второй позволяет этот обмен синхронизировать. Например, если старший бит регистра состояния установлен в единицу, то это означает, что устройство к обмену готово.

Отметим, что контроллеры современных периферийных устройств имеют обычно более двух портов. Например, в распространенных IBM-совместимых компьютерах для работы со стандартным печатающим устройством используется три порта – к уже названным добавляется регистр управления. В оправдание выбранной нами "двухпортовой" модели (которая, кстати, реально существовала на многих

компьютерах) скажем, что ничего принципиально нового наличие дополнительных портов не вносит.

Более подробно работа с внешними устройствами будет описано позднее.

Такова внутренняя структура учебного микрокомпьютера. Рассмотрим теперь устройство самого процессора, каким он видится программисту. "E97" состоит из семи 16-разрядных регистров: 4 регистров общего назначения **R0 - R3**, счетчика адреса команд **PC**, указателя стека **SP** и регистра состояния процессора **PS**, в котором мы будем использовать только два младших бита N и Z. "Поведение" этих управляющих битов согласно общепринятым закономерностям следующее:

N = 0 – результат >=0, N = 1 - результат <0,

Z = 0 – результат <>0, Z = 1 - результат =0.

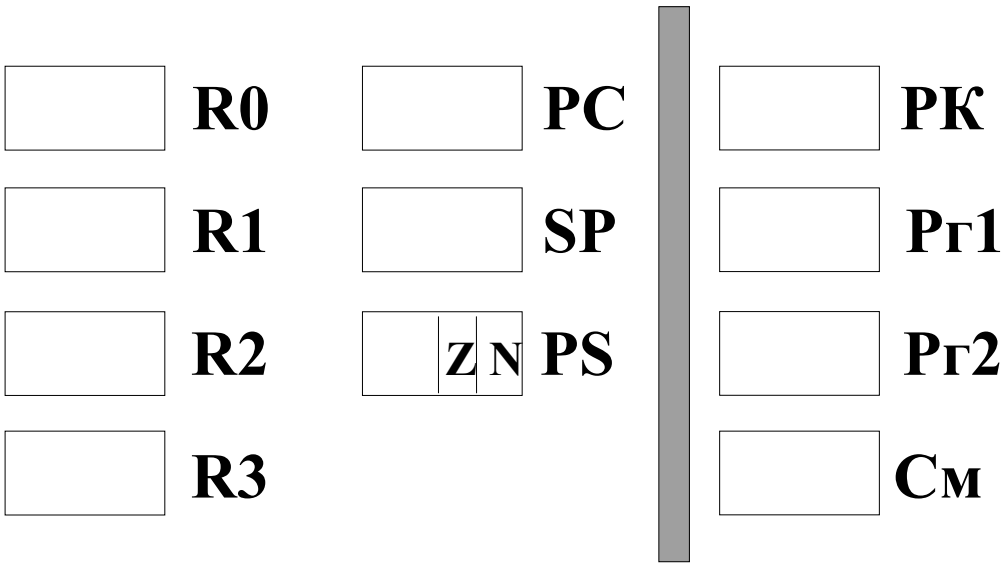


Рис. 3. ***Внутреннее устройство учебного процессора "E97"***

Все эти объекты изображены на рис. 3. После долгих колебаний автор все-таки сохранил за ними общепринятые "англоязыкие" сокращения и не стал вводить собственные русские. Кроме уже названных, в "E97" имеется ряд *внутренних* регистров, которые процессор использует в процессе исполнения операций. Это **регистр команд РК**, предназна-

ченный для хранения кода исполняемой в данный момент команды; **регистры операндов Рг1 и Рг2**, куда считываются исходные данные; **сумматор См**, в котором производится требуемое в команде действие и получается результат. Мы не можем непосредственно изменять содержимого этих "служебных" регистров, но в нашем учебном компьютере они доступны для нашего наблюдения.

Если вы внимательно прочитали материал данного раздела, то готовы к изучению системы команд процессора "E97".

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Назовите основные функциональные устройства учебного компьютера "E97" и охарактеризуйте их назначение.

2. Какие виды памяти существуют в "E97" и как они адресуются?

3. Сопоставьте адресацию к байтам и к словам. Какие вам удалось заметить отличия?

4. В ОЗУ начиная с адреса 300 хранятся следующие байты: 00, 01, 02, 03. Какое слово будет прочитано по адресу 302?

5. В ОЗУ начиная с адреса 300 хранятся слова 0102 и 5301. Совпадают ли находящиеся в адресах 301 и 302 байты?

6. Что такое BIOS и какова его роль в современном компьютере. Знаете ли вы название фирмы, написавшей BIOS для вашего компьютера?

7. Пользуясь рис. 2 и данными к нему, вычислите размеры ОЗУ и ПЗУ. Что оказалось больше?

8. Почему, по-вашему, удобнее иметь программу вывода текста на экран, чем встроенную в процессор команду вывода?

9. Какие значения получат управляющие признаки, если результат операции получится равным 1? -1? 0?

10. Какие внутренние регистры содержит учебный процессор "E97" и каково их назначение?

3. Система команд "E97"

Перейдем к самому важному – системе команд, которые умеет выполнять наш учебный процессор. Машинная команда состоит из **операционной** и **адресной** частей: первая указывает, что надо сделать с данными, а вторая - где их взять и куда поместить результат. В этом разделе мы будем говорить в основном об операционной части, лишь коротко упоминая об адресной: в последней нас пока главным образом будет интересовать количество операндов (адресов). Вопросам, связанным с подробностями адресации данных, будет посвящен следующий раздел.

Итак, рассмотрим структуру команды "E97". В зависимости от конкретной операции, ее формат может иметь некоторые особенности, но в наиболее полной форме он состоит из 4 частей по 4 бита каждая (см. рис. 4б): **модификатор МОД**, **код операции КОП** и два **операнда ОП1** и **ОП2**. Назначение трех последних для нас уже очевидно. Что же касается МОД, то он указывает варианты реализации команды, например, адресовать байт или слово, по каким управляющим битам переходить и т.п. Более подробно роль этой части команды будет обсуждаться позднее.

Наиболее простой формат команд из всех возможных, имеют две – нет операции (ее код 0) и останов (код F). Как видно из рис. 4а, в этих командах задействован только КОП, остальные 12 бит значения не имеют.

а (КОП=0, КОП=F)

б (КОП=1-B)

в (КОП=C)

г (КОП=D)

д (КОП=E)

Рис. 4. ***Форматы команд учебного процессора "E97".***

Основная масса команд, коды которых заключены в интервале от 1 до B, является двухадресной и соответствует уже упоминавшемуся ранее рис. 4б, к ним относятся:

1 – перепись,
2 – сложение,
3 – вычитание,
4 – сравнение,
5 – умножение,
6 – деление,
7 – логическое "И",
8 – ИЛИ,
9 – ИСКЛЮЧАЮЩЕЕ ИЛИ,

A – ввод из порта,
B – вывод из порта.

Операция переписи выполняется достаточно тривиально: считывается информация из ОП1 и копируется в ОП2. Совершенно аналогично работают ввод и вывод из порта, с той лишь разницей, что в качестве одного из операндов указывается номер порта. Все остальные двухадресные команды с кодами 2-9 представляет собой определенные действия над двумя данными, выполняемые по универсальной схеме

ОП2 операция ОП1 ==> ОП2.

Например, по команде деления процессор извлекает ОП2, делит его на ОП1 и результат помещает вместо первоначального значения ОП2.

Некоторую особенность имеет команда сравнения. При ее исполнении производится вычитание ОП2 – ОП1, но результат никуда не записывается. Тогда для чего же вычитать? Исключительно ради установки управляющих битов, которые в дальнейшем могут быть проанализированы командами условных переходов.

Для закрепления рассмотрим небольшой довольно абстрактный пример, который производит вычисление по формуле R0 = (R1+R2)*R3 и результат из R0 выводит в порт номер 3:

адрес	код	команда
0000	0110	R1 ==> R0
0002	0220	R0 + R2 ==> R0
0004	0530	R0 * R3 ==> R0
0006	0B03	R0 ==> порт 3
0008	0F00	останов

Думается, разобравшись с этой программой, вы поняли, как работают все двухадресные команды с кодами от 1 до B.

Перейдем теперь к рассмотрению команд переходов. Как мы уже знаем, они бывают **абсолютные**, когда значение адреса для перехода задается явно, и **относительные**, когда адрес следующей команды вычисляется путем значе-

ния текущего программного счетчика и указанного в теле команды смещения. В соответствии с этим в "E97" есть два типа переходов с кодами операций C и D; их форматы представлены на рис. 4в и г.

Начнем с абсолютного перехода, код которого равен C. Если процессор встретит в программе команду из двух слов (как возникают такие команды, пока не спрашивайте – об этом в следующем разделе!)

1C0D
0056

то следующей будет выполняться команда с адресом 56. Иными словами, адрес перехода берется из самой команды.

По другому обстоит дело с относительным переходом, код которого D. В качестве примера возьмем команду

1D06.

Для определенности будем считать, что эта команда находится в памяти по адресу 42. В строгом соответствии с основным алгоритмом работы процессора, после выборки рассматриваемой команды счетчик адреса команд PC автоматически увеличивается до 44. Затем, выполняя расшифрованную команду перехода, процессор прибавит к текущему содержимому PC смещение 06 и тем самым осуществит переход на адрес 44 + 6 = 4A. Обратите внимание, что итоговый адрес в случае относительного перехода зависит от расположения команды перехода в ОЗУ.

При отрицательном смещении возможно получение адреса меньше исходного.

Надеюсь, вы помните о нашей договоренности, что адрес слова, а значит и команды, всегда четный - в противном случае "E97" фиксирует ошибку и останавливается. Это означает, что младший бит обязан ***всегда*** иметь нулевое значение, а следовательно, фактически не используется. В процессоре PDP для "экономии места" этот бит даже не хранился: там код смещения в командах относительных переходов сдвинут на один разряд вправо. Такой прием позволяет увеличить максимально возможную "дальность" перехода вдвое, но зато вычислению адреса должен пред-

шествовать обратный сдвиг кода смещения на один разряд влево. В нашем учебном процессоре простота и наглядность важнее технического совершенства, поэтому смещение хранится без всяких преобразований. Но помните: его нельзя задавать нечетным!

Наиболее наблюдательные читатели, видимо, заметили, что при обсуждении команд перехода мы незаметно включили в работу модификатор команд. Для переходов его роль проста и наглядна: МОД показывает, по какому условию осуществляется переход. Таблица всех используемых в "E97" значений модификаторов выглядит так:

0 – возврат из подпрограммы
1 – безусловный переход
2 – N = 0 (>=0)
3 – N = 1 (<0)
4 – Z = 0 (<>0)
5 – Z = 1 (=0)
6 – N = 1 or Z=1 (<=0)
7 – N = 0 and Z = 0 (>0)
9 – вызов подпрограммы.

Становится очевидным, что рассмотренные в обоих предыдущих примерах команды с МОД = 1 являются наиболее простым вариантом перехода – безусловным.

Для работы с условными переходами следует твердо запомнить следующее правило: ***если анализируемое условие справедливо, т.е. состояние управляющих признаков совпадает с требуемым, то переход происходит. В противном случае никаких действий не производится, а значит переход просто игнорируется и процессор, как обычно, выбирает следующую команду***. Кстати, если вы знакомы хотя бы с одним из языков программирования, то все эти прописные истины вам давно известны.

А вот пример для закрепления. Пусть в R1 и R2 хранятся некоторые двоичные коды. Если они совпадают, требуется занести в R3 ноль, иначе – оставить без изменения. Один из возможных вариантов решения имеет вид:

адрес	код	команда, пояснения
0000	0110	R1 ==>R0
0002	0920	R0 xor R2 ==>R0
0004	4D02	если результат <>0, к адресу 8
0006	0103	R0 (т.е. 0) в R3
0008	0F00	останов

В программе используется свойство операции XOR - "исключающее или" - давать нулевой результат при побитном совпадении кодов.

И еще об одном виде перехода хочется поговорить особо – о **переходе с возвратом** или о **переходе к подпрограмме**, как его часто называют. Эта команда, как мы видели при анализе существующих моделей, в учебных ЭВМ никогда ранее не реализовывалась, хотя на практике подпрограммы (процедуры) и функции играют очень важную роль.

Подпрограммы полезны, когда в разных местах программы требуется выполнить одни и те же действия. В этом случае имеет смысл оформить повторяющиеся действия в виде подпрограммы, а затем просто вызывать ее в нужных местах. В каком-то смысле это похоже на публикацию текстов песен, когда припев пишется один раз, а в дальнейшем просто ставится ссылка на него в виде слова "ПРИПЕВ".

Наиболее важное отличие перехода к подпрограмме от безусловного перехода состоит в том, что требуется иметь возможность вернуться из подпрограммы в то же самое место, откуда она была вызвана. Это напоминает покупку билета с обратом по сравнению со сменой места жительства. Применительно к процессору возможность возврата означает запоминание где-нибудь значения программного счетчика PC: для возврата достаточно будет просто восстановить в PC сохраненное значение. Остается выяснить, где и как лучше всего запоминать содержимое программного счетчика в момент вызова подпрограммы. Для этой цели можно было бы использовать специальный регистр, но тогда после первого же обращения к подпрограмме этот ре-

гистр оказывался бы занятым и новую подпрограмму уже невозможно было бы вызвать до окончания работы данной. Иначе говоря, для обеспечения возможности вложенных друг в друга подпрограмм, необходимо уметь сохранять не одно значение PC, а несколько. К счастью, для реализации такого своеобразного механизма памяти давно разработана так называемая стековая память. Стек идеально подходит для реализации любой вложенности конструкций и при этом требует наличия всего одного выделенного регистра – указателя стека SP.

Но довольно общих рассуждений – пора рассмотреть конкретный пример. Пусть в некотором месте программы находится команда из двух слов

9C0D
0030.

Исполнять эту команду процессор будет так. Прежде всего, он уменьшит SP на 2 и запомнит по полученному адресу текущее содержимое PC (надеюсь, вам не стоит напоминать, что счетчик к этому времени уже будет показывать на следующую команду?) Затем последует переход по адресу 30, считанному ранее из второго слова команды. Таким способом мы попадем в подпрограмму, надежно спрятав в стеке адрес основной программы, куда нужно вернуться. Обсудим теперь, как произойдет возврат.

В конце любой подпрограммы должна стоять команда 0C00 или 0D00. Встретив ее, процессор извлечет из стека занесенное туда ранее значение и поместит его в PC. При этом он не забудет увеличить SP на 2, освободив ненужную более ячейку памяти в стеке. Таким образом, прерванное на время выполнение основной программы продолжится с нужного места.

ПРИМЕЧАНИЕ. Команды 0C и 0D тождественны и их младшие 8 бит не используются.

Для того, чтобы лучше разобраться в описанном механизме, попробуйте самостоятельно проанализировать ситуацию, когда одна подпрограмма вызывает другую. Полу-

чилось? Тогда займемся последней группой команд процессора, для которой КОП = E.

Прежде всего, почему такому КОП соответствует группа, а не одна команда? Все дело в том, что эти команды одноадресные и освобождающиеся от одного из операндов 4 бита можно использовать для задания номера операции в этой группе. Назовем полученные биты дополнением к КОП – дКОП. В итоге получим формат, приведенный на рис. 4д. Из рисунка видно, что код операций для всех одноадресных операций состоит из двух шестнадцатеричных цифр, причем первая из них всегда E. Теперь познакомимся с этими командами более подробно.

Команда E1 выполняет над единственным операндом ОП1 логическую операцию НЕ, т.е. заменяет нулевые биты единицами и наоборот.

Команды E2-E9 обеспечивают работу со стеком. Так, при коде операции E2 ОП1 заносится в стек, а при E3 – считывается оттуда. Например, вот как можно поменять местами содержимое регистров R1 и R2 с использованием стека:

адрес	код	команда
0000	0E21	записать R1 в стек
0002	0E22	записать R2 в стек
0004	0E31	считать значение из стека в R1
0006	0E32	считать значение из стека в R2
0008	0F00	останов

Команды E4 и E5 позволяют изменять значение SP на величину ОП1, что часто бывает полезно при работе со стеком, например, при "освобождении" в нем сразу нескольких "этажей". По кодам E6 и E7 можно задать новое значение SP и прочитать его текущее значение в ОП1. Наконец, наиболее экзотические из этой группы команды E8 и E9 сохраняют в стеке и восстанавливают для последующего анализа регистр состояния процессора PS. Эта парочка замечательна тем, что обрабатывает вполне определенный операнд, поэтому содержимое ОП1 в команде значения не имеет; *договоримся заполнять его нулем*.

Осталось рассмотреть последнюю группу команд - сдвиги. Их коды EA-EC. Все они осуществляют сдвиг кода в ОП1 на один разряд влево или вправо в зависимости от значения дКОП. Полезно помнить, что сдвиг влево эквивалентен умножению, а вправо – делению на два. Проделайте сдвиги над каким-нибудь числом и убедитесь в этом самостоятельно.

Команда EC, называемая **арифметическим** сдвигом, отличается от обычного сдвига EB тем, что старший знаковый разряд при арифметическом сдвиге сохраняет свое значение, например:

исходное значение ОП1: 1111 0000 1111 0000
результат команды EB: **0**111 1000 0111 1000
результат команды EC: **1**111 1000 0111 1000.

Арифметический сдвиг бывает полезен для деления отрицательных чисел, т.к. в этом случае автоматически сохраняется признак знака минус – единица в старшем разряде.

Завершая обсуждение системы команд, рассмотрим интересный, поучительный и практически полезный пример. Часто для оптимизации программ по быстродействию вместо умножения числа на 10 используется "хитрый" алгоритм сдвигов (особенно это актуально, если в системе команд процессора умножение не реализовано). В основе вычислений лежит математическое тождество

$$10z = 2z + 8z.$$

Программа для определения 10*R0 имеет вид:

адрес	код	команда	пояснения
0000	0EA0	сдвиг R0 влево	2*R0
0002	0101	R0 ==> R1	2*R0
0004	0EA0	сдвиг R0 влево	4*R0
0006	0EA0	сдвиг R0 влево	8*R0
0008	0210	R0 + R1 ==> R0	10*R0
000A	0F00	останов	

Конечно, по длине она явно больше, чем простое умножение. Но работать, как не странно, будет быстрее, т.к. умножение - очень "медленная" операция. Не верите? Про-

верим для конкретного процессора. К сожалению, для современных моделей процессоров сделать это трудно из-за сложности алгоритмов их работы, поэтому произведем оценку для более «старых» моделей. Например, для Intel 80286 команды сдвига регистра микропроцессора на один разряд, переписи регистр-регистр и сложение регистр-регистр выполняются очень быстро и требуют всего по 2 машинных такта. Несложно подсчитать, что все вычисления по нашей программе будут выполнены в течение 10 тактов работы процессора. В то же время только непосредственно на умножение регистр-регистр Intel 80286 тратит 21 такт, да еще в один из регистров потребуется занести значение 10! Таким образом, налицо явная экономия времени.

Можно считать, что рассмотренный пример также косвенно демонстрирует потенциальное преимущество в быстродействии RISC-процессора, оперирующего небольшим, но тщательно оптимизированным набором команд, по сравнению с обычным процессором.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Из каких частей состоит полная команда "E97"?

2. Какие форматы может иметь команда "E97"? Приведите примеры команд на каждый из них.

3. Постарайтесь найти все команды, в которых какие-либо биты не используются.

4. Напишите программы вычисления 2-3 арифметических выражений.

5. Чем отличаются переходы с кодами C и D и как они работают?

6. Если рассмотренную в одном из примеров команду 1D06 поместить в ячейку с адресом не 42, а 4E, то какая команда будет выполняться следующей?

7. Каково должно быть смещение в команде перехода к адресу 56, если сама команда имеет адрес 4E? 5E?

8. Какие значения может принимать модификатор в командах перехода и что они обозначают?

9. Напишите программу, которая переписывает в R3 наименьшее из чисел, хранящихся в регистрах R1 и R2.

10. Придумайте 3-4 примера задач, где может быть полезно использование процедур или функций.

11. Как процессор обращается к подпрограмме и выходит из нее обратно?

12. Какие можно делать операции с указателем стека?

13. Чем отличается сдвиги с кодами операций EB и EC? Приведите пример, когда эти различия незаметны.

4. Адресация данных в "E97"

Теперь, когда мы знаем практически все об операционной части команды, можно, наконец, поговорить и об адресной. Посмотрим, какими способами могут представляться операнды ОП1 и ОП2. Вы скоро обнаружите, что для "E97" отобраны наиболее простые из имеющихся в реальных процессорах методы адресации, но на первых порах это скорее должно казаться достоинством, чем недостатком учебной модели.

Начнем с того, что под кодирование каждого операнда всегда отводится 4 двоичных разряда. Из них старшие два будут всегда задавать тип адресации данных, а младшие - его конкретизировать. В большинстве случаев два младших бита будут просто представлять собой номер регистра, с участием которого осуществляется адресация.

Старшая "половинка" модификатора, соответствующая типу адресации, может содержать четыре неповторяющиеся двоичные комбинации:

00 – **регистровый** метод адресации: операнд является содержимым указанного регистра;
01 – метод **косвенной** адресации: операндом является содержимое ячейки ОЗУ, адрес которой задан в указанном регистре;
10 – резерв; возможно, в будущих версиях здесь будет реализован индексный метод адресации;
11 – адресация **по PC**: операнд извлекается с использованием информации, входящей в команду; более подробно существующие варианты этого метода поясняются ниже.

Рассмотрим перечисленные способы адресации подробнее.

Наиболее простым является регистровый метод, который мы фактически уже использовали в примерах предыдущего раздела. Учитывая, что этому методу соответствуют нулевые значения старших битов, полный код операнда совпадает с номером регистра: двоичная комбинация 0000 соответствует R0, 0001 - R1 и т.д. В качестве данных для операции используется информация, содержащаяся в указанном регистре. Например, если R1=3, а R2=5, то в результате выполнения команды

0212 : R2 + R1 ==> R2

получится R2=8. Схематически это можно представит так:

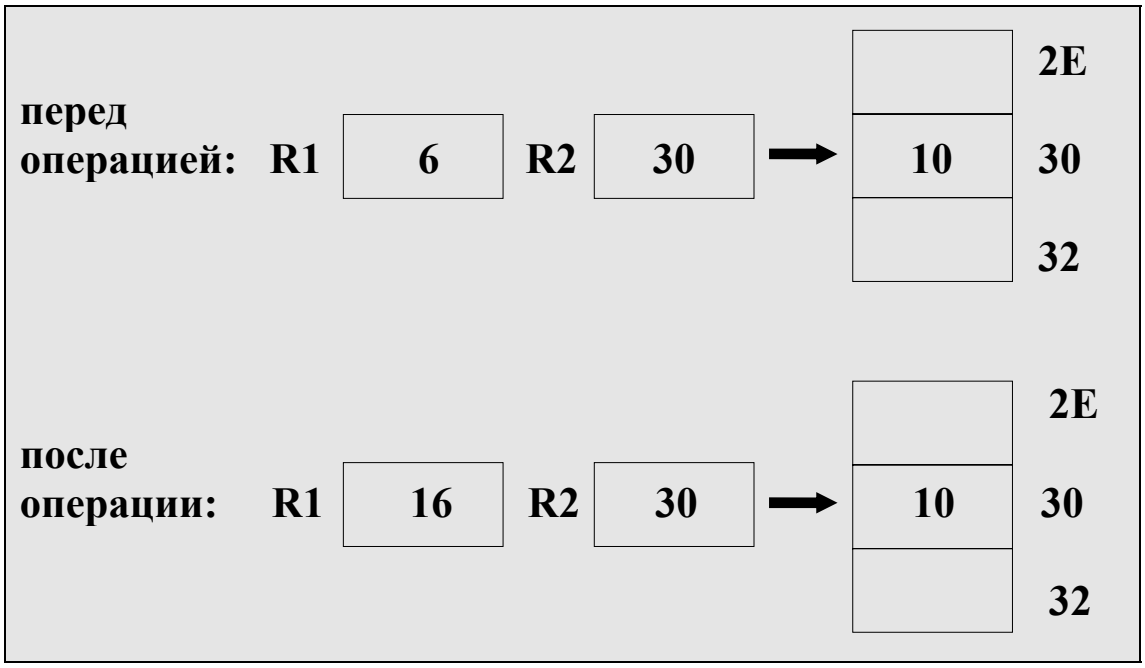
перед операцией:	**R1**	**3**	**R2**	**5**
после операции:	**R1**	**3**	**R2**	**8**

При косвенной адресации код операнда выглядит несколько сложнее: косвенное обращение по регистру R0 имеет вид 0100 (т.е. 4), по R1 - 0101 (5) и т.д. Содержимое

указанного регистра при этом служит не операндом, а его адресом в ОЗУ. Рассмотрим команду

0261 : R1 + (R2) ==> R1 ,

где скобки у R2 символизируют косвенную адресацию. Пусть содержимое R2 в данный момент равно 30, а R1=6. Примем также, что в ячейке памяти с адресом 30 хранится число 10. Тогда процессор "E97", выполняя команду, к имеющемуся в R1 значению 6 прибавит число из ячейки 30, на которую указывает R2, и результат операции 16 занесет в R1.



И, наконец, рассмотрим способы адресации по программному счетчику PC. Поскольку в этом случае регистр, по которому производится адресация, уже однозначно определен, освобождаются два младших бита операнда и их можно использовать для других целей. В связи с этим удается получить 4 различных способа адресации по счетчику:

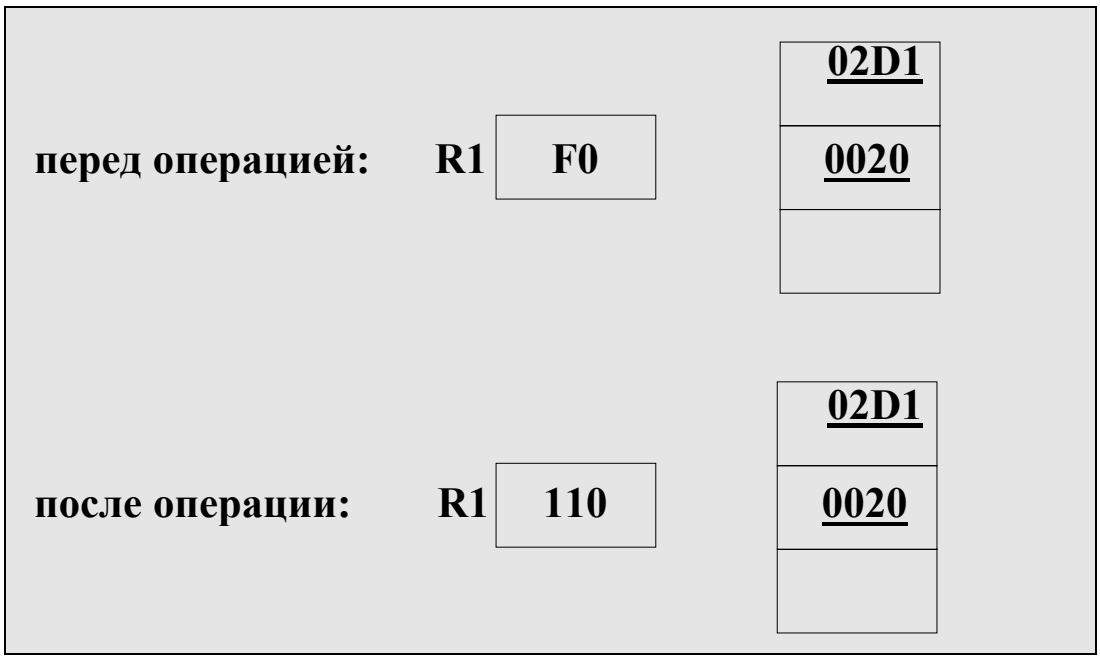
11 00 – резерв;
11 01 – операндом служит константа, входящая а команду;
11 10 – операнд извлекается из ячейки ОЗУ, адрес которой входит в команду;
11 11 – резерв.

Отметим, что использование любого зарезервированного метода адресации воспринимается как ошибка и приводит к аварийному прекращению выполнения программы.

Итак, в "Е97" существует два метода адресации по PC, соответствующие шестнадцатеричным кодам **D** и **E** в качестве операнда. Изучим их на примерах. Команда, состоящая из двух слов

02D1 : R1 + 20 ==> R1
0020

выполняется так: к текущему значению R1 прибавляется извлеченное из команды число 20 и результат помещается в R1. Если первоначальное значение R1 было, скажем, F0, то в результате операции в R1 запишется 110:



Договоримся, что случай использования этого типа адресации в ОП2 является ошибкой, т.к. по логике операции тогда требуется поместить результат в константу! Для предотвращения этих ошибок научим "Е97" их фиксировать.

Рассмотрим еще одну команду с адресацией по PC:

021E : (20) + R1 ==> (20)
0020

Она прибавляет к числу, хранящемуся в ячейке памяти 20, значение R1. В случае, когда R1=F0, а (20)=40, ответ будет (20)=130:

И еще одно замечание теоретического плана. Описанные выше методы адресации по программному счетчику и работа со стеком в нашем процессоре выглядят как *самостоятельные* методы доступа к данным. В то же время удается построить наглядную и достаточно общую теорию адресации, когда указанные выше методы являются просто *частным случаем*. Если вас заинтересовал этот вопрос, обратитесь к литературе по PDP-11.

Так выглядят методы адресации данных, которые вы можете использовать в "E97". При их описании я все время начинал с двоичной структуры записи ОП1 или ОП2, а затем переводил ее в шестнадцатеричную цифру. Такой вариант для понимания логики построения команд, без сомнения, предпочтительнее. В то же время существуют люди, которым манипуляции с нулями и единицами кажутся неприятным анахронизмом. Для них в приложении 3 приве-

дена готовая таблица кодировки методов адресации "E97", пользуясь которой можно сразу получить шестнадцатеричную цифру для ОП1 или ОП2.

На этом можно было бы и закончить, если бы не наличие в командах модификатора, хранящегося в первой цифре шестнадцатеричного представления команды. Его значение также может существенно влиять на извлечение данных и запись результата.

Во всех наших примерах МОД=0 и, следовательно, не оказывает влияния на расшифровку и выполнение операции. Если же он имеет ненулевое значение, необходимо дополнительно учитывать и этот фактор.

Модификатор состоит из 4 битов, причем два старших отвечают за "переключение" байт/слово, а два младших - за особый способ представления данных, называемый "**короткой константой**". С нее-то мы и начнем.

Многие команды оперируют с небольшими константами, значения которых не превышает по модулю 15. Такие константы вполне "уместятся" в четырех отведенных под ОП1 битах. Для реализации такой возможности в "E97" используются 2 младших бита модификатора – 0-й и 1-й. Признаком, по которой процессор распознает "короткую константу", служит наличие единицы в первом разряде модификатора. В этом случае содержимое ОП1 рассматривается как двоичное число, а 0-й бит модификатора служит его знаком, например:

25A3 : R3 * 10 ==> R3

или 3653 : R3 / (-5) ==> R3

Из приведенных примеров видно, что операции с "короткой константой" занимают всего одно слово. Данный способ оказывается, таким образом, экономичнее описанного чуть раньше метода адресации по PC: вспомните, что там константа хранится во втором слове команды.

Отметим, что 0-й и 1-й биты модификатора влияют только на ОП1 и при работе с ОП2 всегда игнорируются.

Наличие операций с "короткой константой" позволяет упростить систему команд процессора. Например, в про-

граммах часто требуются команды, увеличивающие или уменьшающие данные на единицу. Любой процессор имеет для этой важной ситуации специализированные команды, обозначаемые чаще всего INC и DEC. В "E97" эти команды получаются как частный случай операций с "короткой константой". Примеры наиболее часто встречающихся команд такого сорта приведены в приложении 2.

ПРИМЕЧАНИЕ. Если описанный механизм показался вам сложным для объяснения ученикам, то можно просто дописать к системе команд несколько дополнительных операций, не раскрывая способы их реализации и даже не упоминая термина "короткая константа".

Во всех предыдущих рассуждениях мы молчаливо полагали, что процессор работает только с 16-разрядными словами. Но это не всегда так – иногда требуется выполнить те или иные действия над байтами, например, при обработке текста. Старшие биты модификатора как раз и предусмотрены для указания режима байт/слово: 3-й бит соответствует ОП1, а 2-й - ОП2. Если соответствующий бит модификатора установлен в единицу, то процессор оперирует с байтом, иначе - со словом. Как всегда, наличие 2 битов порождает 4 варианта:

00 – ОП1 - слово, ОП2 - слово;
01 – ОП1 - слово, ОП2 - байт;
10 – ОП1 - байт, ОП2 - слово;
11 – ОП1 - байт, ОП2 - байт.

Наиболее простыми и очевидными являются комбинации слово-слово и байт-байт, когда оба операнда и результат имеют одинаковый тип. Смешанные комбинации слово-байт и байт-слово более экзотичны и нуждаются в подробном рассмотрении. Если вы уже устали от вариантов адресации и не намерены вникать в тонкости, то без большого ущерба сразу переходите к чтению вывода и ответам на контрольные вопросы. Но для более глубокого понимания все же попробуйте разобраться в приведенном ниже материале.

Строго говоря, операция слово-байт, например, нахождение суммы слова и байта не является чем-то абсурдным. В самом деле, почему нельзя к слову 528 прибавить байт 19? Некоторую неоднозначность вызывает только вопрос, чем заполнить недостающие старшие разряды слова, особенно если байт является отрицательным числом. Все эти вопросы в принципе можно было бы успешно разрешить, установив систему правил обработки данных, но мы поступим проще: запретим все "смешанные" операции слово-байт и байт-слово кроме одной - переписи байт-слово. Эта "исключительная" операция будет всегда иметь МОД=8 и КОП=1, например:

8101 : R0b ==> R1

(договоримся байтовый операнд помечать малой латинской буквой b).

При работе с байтовыми данными всегда будем в регистрах использовать младший байт. Если же речь идет о переписи байта в слово, то дополнительно сформулируем **правило "расширения знака"**:

все биты старшего байта слова при переписи в него байта заполняются значением, которое имеет самый старший бит переписываемого байта.

Эта слегка запутанная формулировка на самом деле довольно проста. Например, для обсуждаемой команды 8101 при R0=201 результатом переписи в R1 будет 1, а при R0=281 после выполнения этой же операции R1 получит значение FF81, т.к. старший (знаковый) бит кода 81 равен 1.

Таким образом, подробно рассмотренная выше команда переписи позволяет "арифметически корректно" преобразовывать байт в слово, сохраняя при этом знак числа.

Чтобы еще лучше представить себе влияние модификатора, сравним 3 похожие команды:

1) 0141 : (R0) ==> R1 (слово ==> слово)
2) C141 : (R0)b ==> R1b (байт ==> байт)
3) 8141 : (R0)b ==> R1 (байт ==> слово).

Пусть до операции во всех трех случаях R0=30, (30)=F1F2, а R1=1F2F. Тогда после выполнения соответствующих команд получим следующие результаты:

1) R1=F1F2

2) R1=1FF2 (старший байт остался без изменения)

3) R1=FFF2 (произошло расширение знака).

Скорее всего, вам не понравились "мудрствования" по поводу преобразования байта в слово. Поэтому, умалчивая о тонкостях, можно поступить проще: считать, что в двухадресных командах модификатор всегда равен либо **0 для операции со словами**, либо **C для байтов**. Естественно, что для одноадресных команд 2-й бит модификатора не используется, а значит, его содержимое не имеет значения.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Какие методы адресации реализованы в "E97"? Приведите по 2-3 примера на каждый из них.

2. Как кодируются следующие операнды: R2? (R3)? Константа 345? Ячейка 66?

3. Объясните, как будет выполняться команда 0572 и какие значения требуется знать, чтобы предсказать ее результат.

4. Чем отличается выполнение команд 0301 от 0345?

5. Изменится ли значение регистра R0 при выполнении команды 0E14?

6. Сколько слов занимает команда (40) - (50) ==> (40) и как она кодируется?

7. Как записывается команда, умножающая на 3 содержимое ячейки 50?

8. Какой командой можно присвоить регистру R2 некоторое значение?

9. Что такое "короткая константа" и когда она бывает полезна?

10. Для разбираемых в данном разделе команд 25A3 и 3653 напишите эквивалентные, используя метод адресации констант по счетчику команд.

11. Почему среди списка полезных команд с "короткой константой" нет умножения на 2?

12. Подробно объясните с точностью до бита кодировку команды C470.

5. Реализация работы с внешними устройствами

Как уже неоднократно говорилось ранее, одним из достоинств модели "E97" является отражение наиболее простых методов обмена с внешними устройствами, свойственных современным компьютерам.

Как и в реальных микропроцессорах, "общение" с периферийными устройствами осуществляется через порты ввода-вывода. Для учебной ЭВМ принята наиболее простая модель, когда каждому из имеющихся внешних устройств соответствует два порта – **порт данных** и **порт состояния**. Через порт данных происходит обмен информацией. Порты данных устройств ввода доступны только по чтению, а вывода только по записи. Порты состояния позволяют управлять процессом обмена данными, т.к. предоставляют процессору сведения о текущем состоянии внешнего устройства. Реальная синхронизация ввода-вывода является достаточно сложной проблемой и существенно зависит от типа периферийного устройства. Поэтому в "E97" регистр состояния устроен максимально просто: в нем имеется единственный используемый бит – **бит готовности**. Часто для удобства программного анализа бит готовности помещают в старший, т.е. знаковый разряд регистра состояния. В на-

шем учебном компьютере готовность устройств отображается единицей в старшем, 7-м бите младшего байта. Регистр состояния доступен только для чтения, так что пользователь не имеет возможности влиять на состояние этого регистра. Следует понимать, что "в реальной жизни" в регистре состояния могут использоваться и другие биты, в том числе и такие, на которые пользователь может программно воздействовать.

Все порты считаются 16-разрядными, но реальная информация всегда располагается в их младшем байте. Содержимое битов с 8 по 15 в выходные порты формально заносится, но никакого влияния на устройство вывода не оказывает.

Каждому порту соответствует свой номер (адрес). В "E97" из 16 возможных используются следующие порты:

0 – порт состояния клавиатуры (только чтение)
1 – порт данных клавиатуры (только чтение)
2 – порт состояния дисплея (только чтение)
3 – порт данных дисплея (только запись).

Стандартный алгоритм обмена с использованием портов ввода-вывода выглядит следующим образом. Считывается порт состояния и проверяется значение *знакового бита* его *младшего* байта. Эта операция повторяется до тех пор, пока бит готовности не будет установлен внешним устройством в единицу, что является сигналом процессору к началу обмена. Только после него процессор может записать информацию в порт данных, если речь идет об устройстве вывода, или считать их из порта, если это порт ввода.

Для того, чтобы учащиеся привыкали анализировать состояние бита готовности, в программной модели "E97" обычное значение этого бита равно 0 и только после 2-3 считываний регистра состояния он устанавливается программой имитатором учебной ЭВМ в 1. Такое поведение бита готовности делает невозможным "случайный" нормальный обмен без циклического опроса этого важного бита.

Вот как может выглядеть правильная программа вывода одиночного символа из R0 на дисплей (именно так она реализована в ПЗУ "E97"):

```
┌──→ 0A21    порт 2 ==> R1
│    E401    сравнить с 0 R1b
└─── 2DFA    если >=,  PC-6
     0B03    R0 ==> порт 3
```

Первые три команды считывают и проверяют бит готовности в порту 2, а последняя обеспечивает собственно вывод требуемого символа на дисплей.

Несколько "технических" замечаний по этой программе. Как уже отмечалось ранее, порты 2 и 3 являются 16-разрядными, поэтому команды имеют модификатор, соответствующий операции со словом, хотя символу достаточно только одного байта. Обратите также внимание на команду с "короткой константой" 0, которая ***обязательно*** (!) должна проверять знаковый разряд ***младшего байта***, а не всего слова в целом. Наконец, в приведенном примере мы впервые имеем команду перехода с отрицательным смещением FA. Проверим, что переход действительно такой, как показывает стрелка. Пусть для определенности программа начинается с адреса 60; тогда адрес ячейки, в которой находится команда перехода, обязан быть 64. После выборки перехода PC=0066, поэтому новое значение программного счетчика вычисляется как

```
 +0066
  FFFA
(1)0060
```

(единица переноса в старший разряд теряется: ее просто негде сохранить). Отметим, что при преобразовании смещения от байта к слову использовалось правило "расширения знака", сформулированное в предыдущем разделе: т.к. старший знаковый бит у FA равен 1, то старший байт слова также заполняется единицами и принимает значение FF. Кстати, в случае положительного смещения правило тоже дает верный результат, т.к. в этом случае знаковый разряд

0, а значит слева к байту просто дописывается два шестнадцатеричных нуля.

Мы подробно рассмотрели программирование процедуры вывода символа; ввод осуществляется совершенно аналогично.

Остается сказать несколько слов об идеологии обмена с внешними устройствами в "E97".

В порт ввода под номером 1 с клавиатуры заносится готовый 8-битный код клавиши, в котором уже учтено состояние регистра (верхний/нижний, т.е. заглавный/строчный) и текущий алфавит (русский/латинский). Новый символ поступает в порт только после считывания старого. При нажатии на клавишу дополнительной клавиатуры, в порт последовательно поступает так называемый "префиксный код" (для IBM PC он равен 0) и код, соответствующей нажатой клавише. Таким образом, "E97" позволяет изучать вполне реальные принципы работы клавиатуры.

Теперь по поводу работы с дисплеем. В первой программной реализации "E97" экран предполагается **символьным** и **черно-белым**. Хотя это и не обязательно для написания программ, все же желательно рассказать ученикам, что это означает. В текстовом режиме дисплей без изменения помещает в свою внутреннюю видеопамять тот самый код символа, который мы задаем при выводе в порт 3. Как же в таком случае увидеть его на экране? Дело в том, что контроллер дисплея автоматически с довольно большой частотой - 50-70 раз в секунду - "сканирует" содержимое видеопамяти и разворачивает его на экране. При этом он пользуется таблицей "начертания символов", называемой **знакогенератором**. В знакогенераторе последовательно располагается информация о построчных битовых образах всех символов, входящих в алфавит машины. Единичному биту соответствует светящаяся точка на экране, а нулевому – темная. Заметим, что таблицу знакогенератора чаще всего разрешается изменять, что приводит к изменению вида соответствующих символов на экране. Таким образом, при

работе в текстовом режиме минимальным элементом является не отдельная точка экрана, а так называемое **знакоместо**, т.е. область, на которой изображается один символ (знак).

Если дисплей цветной, то картина несколько усложняется. Здесь мы не будем ее рассматривать, и не только потому, что этот режим в "E97" не реализован: дело в том, что реализация цветных режимов для различных типов дисплеев отличается довольно сильно.

Текстовой режим некоторое время был единственным режимом работы дисплеев. Постепенно его конкурент - графический режим, когда минимальным элементом изображения служит не символ, а пиксел (точка), набрал силу и существенно потеснил своего предшественника. В нем по-прежнему можно выводить символы, которые на плоскости графического экрана сразу же преобразуются в определенную комбинацию светящихся точек, т.е. фактически "рисуются". Но гораздо важнее, конечно, то, что в графическом режиме можно реализовать графики, диаграммы, рисунки и т.д., существенно повышающие наглядность вывода. С переходом к графическим средам типа Windows, текстовой режим дисплея сохраняется лишь ради совместимости со "старыми" программами: все программное обеспечение, написанное для Windows, работает только в графическом режиме.

Возможно, в новой версии учебной ЭВМ будет реализован графический режим дисплея, но пока, осмелюсь еще раз напомнить, дисплей "E97" предполагается текстовым и черно-белым.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Что такое порт ввода-вывода?

2. Как называются порты, имеющиеся у каждого из внешних устройств "E97"?

3. Перечислите все известные вам порты ввода-вывода, реализованные в учебном компьютере "E97", и их номера.

4. Каков стандартный алгоритм обмена процессора с внешним устройством? Какую роль играет в нем бит готовности?

5. Может ли описанный вами алгоритм обмена "зациклиться" и при каких условиях? Что вы можете предложить для устранения этого недостатка?

6. По аналогии с приведенной в тексте программой вывода символа на дисплей, напишите самостоятельно программу ввода символа с клавиатуры. Не забудьте организовать при этом "эхо-печать", т.е. вывести принятый с клавиатуры символ.

7. Разработайте программу приема кодов клавиши с дополнительной клавиатуры.

8. Как работает текстовой режим дисплея и чем он отличается от графического?

9. Возможно ли в текстовом режиме нарисовать какую-либо картинку? Как это сделать?

6. Примеры программ

В данном разделе довольно подробно разбирается большое количество самых разнообразных задач. Автору хотелось, чтобы они были полезными для обучения и интересными. В связи с этим необходимые, но тривиальные упражнения типа "*сложить содержимое всех четырех регистров процессора*" сюда, естественно, не попали.

Все примеры тщательно отлаживались, и затем проверенные работающие программы сразу же записывались на диск в виде текстовых файлов с расширением .COD. Эти файлы непосредственно включались в текст, который вы

читаете, что исключало "ручной" набор большого числа цифр и неизбежные при этом ошибки. Одним словом, делалось все возможное, чтобы избежать ошибок. Тем не менее, веру в "печатное слово" не стоит абсолютизировать и логическая проверка программ не помешает.

Изложение всех задач ведется достаточно подробно, но при этом предполагается, что читатель уже знаком с системой команд учебного микропроцессора "E97".

ПРИМЕР 1. ОСТАТОК ОТ ДЕЛЕНИЯ.

Целые числа A и B хранятся в регистрах R1 и R2. Вычислить результат деления нацело A div B и остаток от деления A mod B, поместив результаты в регистры R1 и R2 соответственно.

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Линейная программа, работающая с регистрами микропроцессора. С небольшой модификацией содержится в ПЗУ с адреса 4000.

РЕШЕНИЕ. Для деления нацело в "E97" существует специальная команда. Что касается остатка от деления, то его можно вычислить по формуле

A **mod** B = A - B * (A **div** B)

При вычислениях для хранения промежуточных результатов используется регистр R0.

Адрес	Код	Операция	Комментарии
0000	0110	R1 ==> R0	A скопировать в R0
0002	0621	R1 div R2 ==>R1	A div B
0004	0512	R2 * R1 ==> R2	B * (A div B)
0006	0320	R0 - R2 ==> R0	A - B * (A div B)
0008	0102	R0 ==> R2	A mod B ==> R2
000A	0F00	останов	

ПРИМЕР 2. ПОВЕРХНОСТЬ ПАРАЛЛЕЛЕПИПЕДА.

Вычислить полную поверхность параллелепипеда со сторонами A, B и C. Считать, что исходные значения находятся в ячейках ОЗУ. Результат также поместить в ячейку.

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Линейная программа. Работает с ячейками ОЗУ, адреса непосредственно входят в команду.

РЕШЕНИЕ. Как известно, полная поверхность параллелепипеда вычисляется по формуле

$$S = 2 * (A * B + A * C + B * C)$$

Для упрощения программы, выражение удобно представить в виде

$$S = 2 * [A * (B + C) + B * C]$$

Запомним на будущее, что преобразование исходного выражения часто позволяет заметно сократить программу.

Адрес	Код	Операция	Комментарии
0000	01E0	(22) ==> R0	B ==> R0
0002	0022		
0004	0101	R0 ==> R1	B ==> R1
0006	02E0	R0 + (24) ==> R0	B + C
0008	0024		
000A	05E0	R0 * (20) ==> R0	A * (B + C)
000C	0020		
000E	05E1	R1 * (24) ==> R1	B * C
0010	0024		
0012	0210	R0 + R1 ==> R0	A * (B + C) + B * C
0014	0200	R0 + R0 ==> R0	2 * [A * (B + C) + B * C]
0016	010E	R0 ==> (26)	результат ==> S
0018	0026		
001A	0F00	останов	
.....			
0020	0002		A
0022	0003		B
0024	0004		C
0026	0034		S

ПРИМЕЧАНИЕ. Не забывайте, что ответ 34 в памяти ЭВМ представлен в шестнадцатеричной системе. В десятичной, как и положено, получится 52.

ПРИМЕР 3. РАЗМЕР МАССИВА.

Вычислить объем памяти, который требуется для размещения целочисленного двумерного массива L x M (L - в R2, M - в R3)

а) в байтах - результат Sb поместить в R0

б) в килобайтах - результат Sk поместить в R1.

УКАЗАНИЕ. Если размер массива в килобайтах не является целым числом, то "неполный" килобайт тоже должен учитываться, например, для массива 1 Кбайт 10 байт правильный ответ 2 Кбайта.

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Несмотря на наличие в указании слова "если", это линейная программа. Использует константы в командах. Дается два решения: одно традиционное, а другое - эстетически более красивое и интересное.

РЕШЕНИЕ. Размер массива в байтах Sb вычисляется по формуле

$$Sb = 2 * L * M,$$

где коэффициент связан с тем, что каждое целое число занимает два байта. Несколько сложнее обстоит дело с размером в килобайтах Sk, т.к. это число может не быть целым. Существует простой способ обойти эту трудность: для этого надо к Sb прибавить 1023 байта и тогда, если только число килобайт не являлось целым, оно возрастет на 1. Например: Sb = 1026 байт, т.е. 1 Кбайт 2 байта. После сложения получим Sb = 1026 + 1023 = 2049 байт, т.е. 2 Кбайта 1 байт. Остается лишь нацело поделить Sb на 1024 и мы обязательно получим правильный результат 2 - второй килобайт "неполный".

Адрес	Код	Операция	Комментарии
0000	0120	R2 ==>R0	L
0002	0530	R0 * R3 ==> R0	L * M

0004	0200	R0 + R0 ==> R0	Sb = 2 * L * M
0006	0101	R0 ==> R1	Sb ==> Sk
0008	02D1	R1 + 3FF ==> R1	+1023 к Sk (для округления)
000A	03FF		
000C	06D1	R1 div 400 ==> R1	Sk div 1024 (в Кбайтах)
000E	0400		
0010	0F00	останов	

Мы уже видели, что иногда более длинная программа работает гораздо быстрее, особенно когда малая длина достигается за счет очень "медленных" команд умножения или деления. Приведенное выше решение также можно оптимизировать по быстродействию. Для этого вспомним, что деление на 2 эквивалентно сдвигу числа вправо на один разряд. Таким образом, деление на 1024 можно заменить десятью сдвигами, т.к. это десятая степень двойки. Но можно поступить и еще проще: поместить старший байт на место младшего, что заменит сразу 8 сдвигов. Вариант такого алгоритма и приведен ниже. Добавим только, что для выделения старшего байта использован следующий прием: слово из регистра записывается в память по адресу 20, при этом его младший байт имеет адрес 20, а старший – 21; именно этот байт считывается затем в R1 и сдвигается дважды.

Адрес	Код	Операция	Комментарии
0000	0120	R2 ==> R0	L
0002	0530	R0 * R3 ==> R0	L * M
0004	0200	R0 + R0 ==> R0	Sb = 2 * L * M
0006	0101	R0 ==> R1	Sb ==> Sk
0008	02D1	R1 + 3FF ==> R1	+1023 к Sk (для округления)
000A	03FF		
000C	011E	R1 ==> (20)	Sk ==> память
000E	0020		
0010	81E1	(21)b ==> R1	старший байт ==> Sk
0012	0021		
0014	0EB1	сдвиг вправо R1	сдвиг Sk
0016	0EB1	сдвиг вправо R1	сдвиг Sk
0018	0F00	останов	

Разумеется, данный вариант решения несколько сложнее для понимания и при первом знакомстве без особого ущерба может быть пропущен.

ПРИМЕР 4. ВВОД ЛАТИНСКИХ БУКВ.

Программа принимает латинскую букву и обрабатывает ее так, чтобы она всегда была

а) заглавной

б) строчной.

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Линейная программа, демонстрирующая использование логических операций для обработки символов. Вызывает программы ввода и вывода символа из ПЗУ.

РЕШЕНИЕ. Главная "хитрость" решения состоит в том, чтобы понять, чем отличаются заглавные буквы от строчных. Для этого из таблицы ASCII выберем наугад одну из букв и выпишем в двоичном виде коды заглавного и строчного символов. Например, для буквы R получим:

R 0101 0010
r 0111 0010.

Повторив аналогичные действия еще для нескольких букв, сделаем вывод, что **коды заглавных и строчных букв отличаются единственным битом** - ***пятым***, если считать младший бит за нулевой. Запомним этот факт, поскольку он справедлив для любого современного компьютера. Итак, задача становится более простой и понятной: для преобразования символа в заглавный надо сбрасывать 5-й бит, а в строчный его устанавливать. Для этой цели прекрасно подходят логические операции **И** и **ИЛИ**.

Рассмотрим подробнее, как сбросить заданный бит. Для этого сначала вспомним таблицу истинности для логической операции И. Внимательно проанализировав упомянутую таблицу в свете решения нашей задачи, заметим следующее. Если в одном из операндов бит равен 0, то вне зависимости от содержимого другого операнда результат всегда 0. Напротив, если в одном из операндов бит установлен

в 1, то результат повторяет бит второго операнда. Получается, что если нам хочется сохранить разряд в исходном коде, надо произвести логическое умножение на 1, а если сбросить в 0 – на 0. Для нашей задачи требуется сохранить все биты, кроме пятого, значит, потребуется константа

1101 1111 = DF.

После выполнения логического И с данной константой любой код "потеряет" ненужный нам бит, сохраняя все остальные.

Аналогичным путем можно рассмотреть логическую операцию ИЛИ и сделать вывод о том, что для установки 5-го бита необходимо выполнить ее с константой

0010 0000 = 20.

Еще одной особенностью решения является активное использование подпрограмм из ПЗУ. Как вы, вероятно, помните, при вызове подпрограмм обязательно используется стековая память, поэтому должен быть корректно определен указатель стека SP. Скорее всего, ваш программный имитатор позаботился об этом (интересно, а сумеете вы узнать, куда показывает SP?), но для воспитания "хороших манер" приведенная в решении программа начинает работу с установки указателя стека.

Адрес	Код	Операция	Комментарии
0000	0E6D	26 ==> SP	установка указателя стека
0002	0026		
0004	9C0D	вызов п/п 40FE	ввод символа (без эхопечати!)
0006	40FE		
0008	0101	R0 ==> R1	сохранить введенный символ
000A	07D0	DF and R0 ==> R0	сделать букву заглавной
000C	00DF		
000E	9C0D	вызов п/п 4088	вывести результат
0010	4088		
0012	0F00	останов	

```
.........
 0022    0072
 0024    0012
 0026    0000
```

Для преобразования символов в строчные, достаточно с адреса 000A вписать команду логической операции **ИЛИ**

08D0

20

Данная задача очень поучительна и интересна с практической точки зрения.

При работе с программой полезно обратить внимание учеников на ячейки 22-24, которые "самостоятельно" заполняет компьютер. Это ни что иное, как стек: в ячейке 24 сохранился адрес выхода из последней подпрограммы, а в 22 - значение R0 при входе в нее (в нашем случае код 72 соответствует букве r).

ПРИМЕР 5. ГДЕ НАХОДИТСЯ КВАРТИРА?

Представьте себе 9-этажный дом с 3-мя подъездами, причем на каждом этаже находится по 6 квартир. Вы первый раз идете к своим новым знакомым в квартиру номер N. В какой подъезд вам заходить и на какой этаж подниматься?

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Это тоже линейная программа, хотя многие сначала пытаются применить развилку или даже цикл. Активно используются подпрограммы ввода-вывода из ПЗУ, что позволяет легко придать работающей программе вполне "реальный" вид. Для нахождения остатка от деления также вызывается подпрограмма из ПЗУ.

РЕШЕНИЕ. Обозначим **P** искомый номер подъезда, а **E** - этаж. Тогда для ответа на вопрос задачи достаточно проделать следующие вычисления:

N1 = N - 1

R1 = N1 div 54

R2 = N1 mod 54

$$P = R1 + 1$$
$$E = R2 \text{ div } 6 + 1$$

где 54 = 6*9 – количество квартир в подъезде. Наличие в нескольких местах прибавления и вычитания единицы связано с тем, что номера квартир, этажей и подъездов начинаются с 1, а не с 0, как гораздо удобнее для написания формул.

И еще одно замечание. Программе требуется рабочая область в ОЗУ для стека и для формирования текстового представления выводимых чисел. Ее адрес трудно предсказать заранее, поэтому для удобства исправлений адрес этот помещается в R0 самой первой командой и в дальнейшем везде используется значение этого регистра.

Адрес	Код	Операция	Комментарии
0000	01D0	70 ==> R0	адрес рабочей области ОЗУ
0002	0070		
0004	0E60	R0 ==> SP	установка указателя стека
0006	9C0D	вызов п/п 40DC	вывод текста
0008	40DC		
000A	4E02	(2)”N”	текст "N="
000C	003D	“=”	
000E	9C0D	вызов п/п 4108	ввод целого числа (N)
0010	4108		
0012	2311	R1 - 1 ==> R1	N - 1
0014	01E2	(50) ==> R2	54
0016	0050		
0018	9C0D	вызов п/п 4000	N1 div 54 ==> R1
001A	4000		N1 mod 54 ==> R2
001C	2211	R1 + 1 ==> R1	R1 + 1
001E	011E	R1 ==> (54)	сохранить P
0020	0054		
0022	06E2	R2 div (52) ==> R2	R2 div 6
0024	0052		
0026	2212	R2 + 1 ==> R2	R2 div 6 + 1

0028	012E	R2 ==> (56)	сохранить E
002A	0056		
002C	9C0D	вызов п/п 40DC	вывод текста
002E		40DC	
0030	5002	(2)”P”	текст "P="
0032	003D	“=”	
0034	0103	R0 ==> R3	адрес рабочей области для текста
0036	9C0D	вызов п/п 4068	вывод целого числа из R1 (P)
0038	4068		
003A	9C0D	вызов п/п 40DC	вывод текста
003C	40DC		
003E	2005	(5)	текст " E="
0040	2020	3 пробела	
0042	3D45	“E=”	
0044	01E1	(56) ==> R1	E
0046	0056		
0048	0103	R0 ==> R3	адрес рабочей области для текста
004A	9C0D	вызов п/п 4068	вывод целого числа из R1 (E)
004C	4068		
004E	0F00	останов	
0050	0036	54	число квартир в подъезде
0052	0006	6	число квартир на этаже
0054	0003		P
0056	0001		E

ПРИМЕР 6. ВЫЧИСЛЕНИЕ МОДУЛЯ.

Найти модуль числа, находящегося в R1.

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Разветвляющийся алгоритм с "неполной" (имеющей единственную ветвь) развилкой. Похожая программа, чуть-чуть менее рациональная, занесена в ПЗУ с адреса 4010.

РЕШЕНИЕ. Если R1 неотрицательно, то ответом является исходное число, поэтому сразу следует переход на команду останова. В противном случае модуль вычисляется по правилу получения дополнительного кода:

$$-A = \text{not}(A) + 1.$$

В программе активно используются команды с "короткими константами"; если вы не хотите объяснять их своим ученикам, то самостоятельно замените их на более длинные, содержащие константу во втором слове команды. Например, вместо 2401 придется записать 04D1 и далее саму константу 0000. Не забудьте после замены команды 2211 на более длинную исправить величину смещения в условном переходе.

Адрес	Код	Операция	Комментарии
0000	2401	сравнить R1 с 0	для неотрицательного R1 -
0002	2D04	если >=0, то pc=pc+4	- выход (к 0008)
0004	0E11	not R1 ==> R1	получение допол-
0006	2211	R1 + 1 ==> R1	нительного кода
0008	0F00	останов	

ПРИМЕР 7. МАКСИМУМ ИЗ ТРЕХ ЧИСЕЛ.

В регистрах R1, R2 и R3 находятся произвольные целые числа. Переписать наибольшее из них в R0.

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Разветвляющаяся программа, содержащая две развилки; вторая – "неполная".

РЕШЕНИЕ. Сначала максимальное из чисел в R1 и R2 занесем в R0. После этого, если R3 окажется больше R0, "исправим положение", переписав в качестве ответа R3.

Адрес	Код	Операция	Комментарии
0000	0412	сравнить R2 с R1	
0002	3D04	если <0, то pc=pc+4	к записи R1 (к 0008)
0004	0120	R2 ==> R0	запомнить R2
0006	1D02	pc=pc+2	обход второй ветви (к 000A)

0008	0110	R1 ==> R0	запомнить R1
000A	0403	сравнить R3 с R0	
000C	3D02	если <0, то pc=pc+2	к выходу (к 0010)
000E	0130	R3 ==> R0	запомнить R3
0010	0F00	останов	

Может быть, кому-то захочется видеть адреса переходов в явном виде, а не вычислять их через смещение. В этом случае программа действительно легче читается, но перестает быть перемещаемой по памяти и становится заметно длиннее. Судите сами.

Адрес	Код	Операция	Комментарии
0000	0412	сравнить R2 с R1	
0002	3C0D	если <0, то к 000C	к записи R1
0004	000C		
0006	0120	R2 ==> R0	запомнить R2
0008	1C0D	к 000E	обход второй ветви
000A	000E		
000C	0110	R1 ==> R0	запомнить R1
000E	0403	сравнить R3 с R0	
0010	3C0D	если <0, то к 0016	к выходу
0012	0016		
0014	0130	R3 ==> R0	запомнить R3
0016	0F00	останов	

Во всех дальнейших программах переходы будут относительными, так как это более рационально и, в то же время, более сложно. Перейти к более наглядным и простым для программирования абсолютным переходам читатели при необходимости смогут самостоятельно.

ПРИМЕР 8. ОПОЗНАВАНИЕ КЛАВИШИ F1.

Программа должна при нажатии клавиши помощи F1 выводить текст "HELP!" и игнорировать нажатие на все остальные клавиши.

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Разветвляющаяся программа, демонстрирующая некоторые приемы обработки служебных клавиш.

РЕШЕНИЕ. Напомним читателю, что при нажатии на любую служебную клавишу в компьютер поступает не один, а два кода: первый – "префиксный" – всегда одинаков, а второй определяет, какая именно клавиша дополнительной клавиатуры была нажата. Таким образом, для решения поставленной задачи можно предложить следующий алгоритм. Если считанный код не является "префиксом", то нажатая клавиша принадлежит к основной части клавиатуры; следовательно, она не может быть интересующей нас F1 и анализ можно закончить. В противном случае требуется проверить второй код и в случае совпадения с требуемым выдать соответствующий текст.

Адрес	Код	Операция	Комментарии
0000	0E6D	40 ==> SP	установка указателя стека
0002	0040		
0004	9C0D	вызов п/п 40FE	ввод символа (без эхопечати)
0006	40FE		
0008	04E0	сравнить R0 с (24)	сравнить с кодом "префикса"
000A	0024		
000C	4D14	если <>0, то pc=pc+14	не совпало – выход (к 0022)
000E	9C0D	вызов п/п 40FE	ввод символа (без эхопечати)
0010	40FE		
0012	04E0	сравнить R0 с (26)	сравнить с кодом клавиши F1
0014	0026		
0016	4D0A	если <>0, то pc=pc+A	не совпало – выход (к 0022)
0018	9C0D	вызов п/п 40DC	вывод текста
001A	40DC		
001C	4805	(5)”H”	текст "HELP!"
001E	4C45	“EL”	
0020	2150	“P!”	

0022	0F00	останов	
0024	0000		код "префикса"
0026	003B		код клавиши F1

Остается только заметить, что коды клавиш могут быть машинно-зависимыми.

ПРИМЕР 9. ПРИЕМ ЦИФРЫ.

Программа принимает код нажатой клавиши и анализирует его. Если введена цифра, то она выводится на экран, преобразуется в целое число от 0 до 9 и записывается в память, иначе на экран выводится символ "?" и в качестве ответа выдается число -1 (код FFFF).

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Разветвляющаяся программа, демонстрирующая приемы обработки символов с клавиатуры. Позволяет проиллюстрировать отличие представления в компьютере цифровых символов "0" - "9" и чисел 0 - 9.

РЕШЕНИЕ. Коды цифр от "0" до "9" образуют возрастающую последовательность от 30h до 39h. Поэтому, если введенный символ является цифрой, то его код обязательно попадает в указанный интервал.

Адрес	Код	Операция	Комментарии
0000	0E6D	50 ==> SP	установка указателя стека
0002	0050		
0004	9C0D	вызов п/п 40FE	ввод символа (без эхопечати)
0006	40FE		
0008	04D0	сравнить R0 с 30	(код <30 - не цифра)
000A	0030	("0")	
000C	3D14	если <0, то pc=pc+14	если не цифра - к 0022
000E	04D0	сравнить R0 с 3A	(код >= 3A - не цифра)
0010	003A	(":")	
0012	2D0E	если >=0, то pc=pc+E	если не цифра - к 0022

0014	9C0D	вызов п/п 4088	вывод символа
0016	4088		
0018	03D0	R0 - 30 ==> R0	преобразовать код символа "0"-"9" в число 0-9
001A	0030		
001C	010E	R0 ==> (38)	записать результат
001E	0038		
0020	0F00	останов	
0022	01D0	3F ==> R0	код "?" в R0
0024	003F	("?")	
0026	9C0D	вызов п/п 4088	вывод символа
0028	4088		
002A	01DE	FFFF ==> (38)	-1 в результат
002C	FFFF		
002E	0038		
0030	0F00	останов	

ПРИМЕР 10. ПРОВЕРКА АДРЕСА ПАМЯТИ.

Допустимые адреса байтов ОЗУ для "E97" находятся в интервале 0 - FF, а для ПЗУ1 4000 - 417F. Проверить, является ли число в R3 допустимым адресом байта и если да, то считать его содержимое в R2, в противном случае занести в R2 код FFFF.

ЗАМЕЧАНИЕ. Учесть, что адрес более 7FFF формально является отрицательным, т.е. <0.

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Достаточно сложная разветвляющаяся программа, содержащая целую систему проверок. Показывает еще один возможный способ программирования "полной" развилки. Содержит операции с байтами.

РЕШЕНИЕ. Главное ветвление программы состоит в том, что при правильном адресе в R2 помещается байт из ОЗУ или ПЗУ, а при неправильном – заданный код. Вторая ветвь программы настолько мала, что удобнее реализовать развилку следующим образом: сначала "на всякий случай" записать в R2 FFFF, а затем проверить правильность адреса.

При обнаружении ошибки программу можно немедленно завершать, т.к. R2 уже установлен; при отсутствии ошибок заменим содержимое R2 байтом из памяти (предварительно "почистив" R2, чтобы старший байт обязательно был нулевым!)

Адрес	Код	Операция	Комментарии
0000	01D2	FFFF ==> R2	
0002	FFFF		
0004	2403	сравнить R3 с 0	(адрес не бывает <0)
0006	3D16	если <0, то pc=pc+16	если да – то к 001E
0008	04D3	сравнить R3 с 4180	(адрес не бывает >=4180)
000A	4180		
000C	2D10	если >=0, то pc=pc+10	если да – то к 001E
000E	04D3	сравнить R3 с 100	(адрес ОЗУ <100)
0010	0100		
0012	3D06	если <0, то pc=pc+6	если да – то к 001A (читать)
0014	04D3	сравнить R3 с 4000	(адрес ПЗУ не бывает <4000)
0016	4000		
0018	3D04	если <0, то pc=pc+4	если да – то к 001E
001A	2102	0 ==> R2	очистить R2 (старший байт)
001C	C172	(R3)b ==> R2b	прочитать в младший байт R2
001E	0F00	останов	

ПРИМЕР 11. СУММА ПОСЛЕДОВАТЕЛЬНЫХ НАТУРАЛЬНЫХ ЧИСЕЛ.

Найти сумму первых 100 натуральных чисел.

Говорят, такую задачу некогда предложил учитель математики детям, чтобы организовать себе время для отдыха. На его беду в классе сидел юный Гаусс, который немедлен-

но выдал правильный ответ. Будущий гениальный математик заметил, что 1+100=101, 2+99=101, ... Определить количество таких пар и умножить его на 101 оказалось возможным даже в уме.

Мы будем решать задачу "в лоб", т.е. честно производить суммирование с помощью компьютера.

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Простейшая циклическая программа.

РЕШЕНИЕ. Поместим в R3 обрабатываемое в данный момент число N (меняется от 1 до 100), а в R0 - результирующую сумму S. Зададим им начальные значения и будем циклически добавлять к S текущее значение N. Признаком окончания цикла будет ситуация, когда N>100.

Адрес	Код	Операция	Комментарии
0000	2113	1 ==> R3	1 ==> N
0002	2100	0 ==> R0	0 ==> S
0004	0230	R0 + R3 ==> R0	S = S + N
0006	2213	R3 + 1 ==> R3	N = N + 1
0008	04D3	сравнить R3 с 100	сравнить N и 100
000A	0064		
000C	6DF6	если <=0, то pc=pc+F6	если N<=100, к повторению (000E+FFF6 = 0004)
000E	0F00	останов	

Обратите внимание на тот момент, что для организации цикла используется обычный условный переход: наш учебный процессор просто не имеет специальной команды организации цикла. В некоторых реально существующих моделях команды циклов все же имеются, но они используются далеко не всегда: достаточно сказать, что цикл с неопределенным числом повторений такими командами принципиально не может быть реализован.

ПРИМЕР 12. ПЕЧАТЬ ЛАТИНСКОГО АЛФАВИТА.

Вывести на экран весь латинский алфавит от "A" до "Z".

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Циклическая программа, параметром цикла в которой служит символ.

РЕШЕНИЕ. Разместим в R0 выводимый латинский символ, первоначальное значение которого будет "A" (код 65=41h). Вывод будем осуществлять обращением к соответствующей подпрограмме ПЗУ. Для перехода к следующему символу алфавита достаточно прибавить 1 к коду текущего символа (не правда ли очень похоже на переход к следующему числу в предыдущем примере?). Остается только проверить, не выходит ли вновь полученный символ за латинский алфавит, т.е. не превышает ли его код 5Ah ("Z") и если да, то закончить.

Адрес	Код	Операция	Комментарии
0000	0E6D	26 ==> SP	установка указателя стека
0002	0026		
0004	01D0	41 ==> R0	код первого символа
0006	0041	("A")	
0008	9C0D	вызов п/п 4088	вывод символа
000A	4088		
000C	2210	R0 + 1 ==> R0	следующий символ
000E	04D0	сравнить R0 с 5A	его код <= "Z"?
0010	005A	("Z")	
0012	6DF4	если <=0, то pc=pc+F4	к повторению (0008)
0014	0F00	останов	

ПРИМЕР 13. ПЕРЕПИСЬ МАССИВА ДАННЫХ.

Скопировать данные, расположенные в последовательных адресах памяти, в указанное место ОЗУ. Начало массива задано в R1, количество байт в нем - в R0, а адрес начала будущей копии информации – в R2.

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Циклическая программа, демонстрирующая использование метода косвенной адресации для организации доступа к массиву данных. Эта программа находится в ПЗУ с адреса 40AE. Советую также самостоятельно разобрать более общую подпрограмму переписи массива (адрес 4096), в которую пример 13 входит в качестве составной части.

РЕШЕНИЕ. Предполагая, что R0 - R2 уже заданы (не забывайте это делать перед каждым запуском программы!), напишем следующую коротенькую программу. Ее главной командой является самая первая - C156. При выполнении этой операции извлекается байт (обратите внимание, именно байт, а не слово!), адрес которого находится в R1, и затем прочитанный код записывается по адресу в R2. После этого модифицируются хранящиеся в R1 и R2 адреса, а из счетчика байтов вычитается 1; пока результат вычитания <>0, цикл повторяется.

Адрес	Код	Операция	Комментарии
0000	C156	(R1)b ==> (R2)b	перепись очередного байта
0002	2211	R1 + 1 ==> R1	новый адрес массива-источника
0004	2212	R2 + 1 ==> R2	новый адрес массива-приемника
0006	2310	R0 – 1 ==> R0	уменьшить счетчик байтов
0008	4DF6	если <>0, то pc=pc+F6	к повторению (к 0000)
000A	0F00	останов	

ПРИМЕР 14. СУММА МАССИВА.

Массив целых чисел начинается с адреса, указанного в R2, и имеет длину в байтах, заданную в R3. Вычислить сумму этого массива и поместить вместо его первого элемента.

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Циклическая программа, работающая с массивами. Демонстрирует организацию процесса суммирования.

РЕШЕНИЕ. Некоторый "подвох" в условии задачи состоит в том, что длина массива указана в байтах, хотя суммируются слова. Поэтому удобнее заданное количество байт преобразовать в количество слов путем сдвига вправо (деление на 2!).

Еще обязательно необходимо сохранить где-нибудь начальный адрес массива, т.к. после вычисления суммы ее надо занести вместо первого элемента. В программе для запоминания адреса использован регистр R1.

Остальная часть программы достаточно проста и в особых разъяснениях не нуждается. Отметим только, что "забавная" команда 2222 получает в R2 адрес следующего слова путем увеличения предыдущего адреса на 2.

Адрес	Код	Операция	Комментарии
0000	0EB3	сдвиг вправо R3	кол-во байт в кол-во слов
0002	2100	0 ==> R0	0 ==> сумму
0004	0121	R2 ==> R1	сохранить адрес 1-го элемента
0006	0260	R0 + (R2) ==> R0	добавить слагаемое к сумме
0008	2222	R2 + 2 ==> R2	адрес следующего слова
000A	2313	R3 - 1 ==> R3	уменьшить счетчик
000C	4DF8	если <>0, то pc=pc+F8	к повторению (к 0006)
000E	0105	R0 ==> (R1)	сумму в 1-й элемент массива
0010	0F00	останов	

ПРИМЕР 15. СКОЛЬКО РАЗ ВСТРЕЧАЕТСЯ В ТЕКСТЕ ЗАДАННЫЙ СИМВОЛ.

В памяти, начиная с адреса 0016, хранится некоторый текст, после которого следует нулевой байт. Определить, сколько раз в тексте встречается символ, код которого задан в R0.

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Комбинированная программа: внутри цикла содержится проверка условия. В отличие от предыдущих примеров, количество повторений цикла заранее неизвестно. Важную роль занимает работа с байтовыми данными.

РЕШЕНИЕ. Пусть R1 будет счетчиком количества совпадений с заданным символом (начальное значение 0), а в R2 хранится адрес рассматриваемого символа текста (начальное значение берем из условия задачи). Алгоритм состоит в последовательном сравнении каждого символа заданного текста с образцом в R0 и прибавлении 1 к R1 в случае совпадения. Цикл прекращается, когда после очередного увеличения адрес показывает на нулевой байт.

Адрес	Код	Операция	Комментарии
0000	2101	0 ==> R1	счетчик совпадений
0002	01D2	16 ==> R2	адрес начала текста
0004	0016		
0006	C460	сравнить R0b с (R2)b	сравнить символ с образцом
0008	4D02	если <>0, то pc=pc+2	обход при несовпадении (к 000C)
000A	2211	R1 + 1 ==> R1	увеличить счетчик совпадений
000C	2212	R2 + 1 ==> R2	вычислить следующий адрес
000E	E406	сравнить (R2)b с 0b	текст не закончился?
0010	4DF4	если <>0, то pc=pc+F4	нет - к повторению (к 0006)
0012	0F00	останов	
0014	0000		
0016	4854	“TH”	текст
0018	5349	“IS”	"THIS IS MY TEXT"+
001A	4920	“ I“	+код 0
001C	2053	“S “	
001E	594D	“MY”	

0020	5420	“ T”
0022	5845	“EX”
0024	0054	“T”(0)

ПРИМЕР 16. НАЙТИ В ТЕКСТЕ ЗАДАННЫЙ СИМВОЛ.

В памяти, начиная с адреса 001A, хранится некоторый текст, длина которого равна 15 (Fh) байтам. Определить номер первого совпадающего с образцом символа в тексте. При отсутствии требуемого символа результат должен быть равен 0. Те, кто программирует на Паскале, конечно, без труда узнали в описании функцию POS.

КРАТКАЯ ХАРАКТЕРИСТИКА РЕШЕНИЯ. Комбинированная программа с циклом и развилкой. Особенностью задачи является возможность досрочного окончания цикла при нахождении символа. Как и в большинстве программ работы с символами, существенным образом используются команды обработки байтов.

РЕШЕНИЕ. Поместим в R1 счетчик символов, в R2 – адрес текущего символа. Затем будем сравнивать каждый символ текста с образцом в R0 и в случае совпадения прервем выполнение цикла. При несовпадении будем продолжать цикл до тех пор, пока счетчик не превысит Fh, т.е. не станет равным 10h. Если цикл завершится по выполнению этого условия, то образцовый символ найти не удалось и в качестве ответа в R1 следует занести 0.

Адрес	Код	Операция	Комментарии
0000	2111	1 ==> R1	номер символа
0002	01D2	1A ==> R2	адрес начала текста
0004	001A		
0006	C460	сравнить R0b с (R2)b	сравнить символ с образцом
0008	5D0C	если = 0, то pc=pc+2	выход при совпадении (к 0016)
000A	2211	R1 + 1 ==> R1	увеличить номер символа

000C	2212	R2 + 1 ==> R2	вычислить следующий адрес
000E	04D1	сравнить R1 с 10	текст не закончился?
0010	0010		
0012	4DF2	если <>0, то pc=pc+F4	нет – к повторению (к 0006)
0014	2101	0 ==> R1	при отсутствии символа – ответ 0
0016	0F00	останов	
0018	0000		
001A	4854	“TH”	текст
001C	5349	“IS”	"THIS IS MY TEXT"
001E	4920	“ I”	
0020	2053	“S “	
0022	594D	“MY”	
0024	5420	“ T”	
0026	5845	“EX”	
0028	0054	“T”	

Давайте ограничимся этими примерами, тем более, что сейчас их число равно 16, т.е. является степенью двойки: компьютер любит такие числа.

Кроме того, не забывайте о том, что у вас есть подробный текстовой файл E97ROM.TXT (см. приложение 4), где также можно найти много интересных и поучительных примеров.

В заключение приведем **несколько практических советов**, как писать программы в кодах процессора.

1. Дело это кропотливое, не надо стесняться проделать предварительную подготовку на бумаге (без компьютера). Поверьте, в дальнейшем такая тщательная подготовка сэкономит массу времени.

2. Начните с того, что спланируйте и запишите, где какая информация будет храниться: аргументы, результаты, рабочие переменные.

3. Тщательно продумайте алгоритм, попробуйте его как-нибудь записать (в виде словесного предписания, блок-схемы, или даже программы на Паскале). Вспомните, не

решали ли вы раньше подобных задач, если да – то найдите решение.

4. Напишите программу в виде содержательных обозначений (см. столбик "Операция" в любом из примеров). Внимательно **проверьте**.

5. Закодируйте полученную программу. Снова внимательно **проверьте**.

6. Теперь можно сесть за компьютер. Наберите программу из тетради и опять **проверьте**. Исправьте все замеченные ошибки.

7. Желательно до первого запуска записать набранную программу на диск: ниоткуда не следует, что она не испортится из-за ошибок.

8. Если программа невелика, лучше сначала проверить ее работу по шагам и только потом запустить полностью.

9. Никогда не ограничивайтесь одной проверкой, даже если ее результаты выглядят очень обнадеживающе. Обязательно придумайте систему тестов, проверяющие все ветви программы. Помните: найти ошибки в программе порой гораздо сложнее, чем ее написать.

10. Сравните хотя бы один результат с полученным "вручную". Старайтесь анализировать правдоподобность получаемых результатов; никогда не ограничивайтесь тем, что программа не выдала ошибок и что-то вывела на экран.

11. Попробуйте выполнить программу для одних и тех же исходных параметров несколько раз: иногда при этом получаются разные результаты, что чаще всего свидетельствует о том, что какой-то регистр или ячейка ОЗУ не получили своего начального значения.

12. Старайтесь при проверке использовать простые значения, но не особенно доверяйте числам типа 0 или 1.

13. Если программа "творит" что-то непонятное, **проверьте** ее еще раз, прежде чем повторно запускать: вдруг одна из многочисленных цифр неправильная. Затем запустите по шагам.

14. Если задачу удается разбить на несколько частей, **обязательно** сделайте это. Наберите первую часть и добей-

тесь ее работы, затем добавьте вторую и т.д. Не забывайте как-нибудь фиксировать заработавший вариант на бумаге и на диске.

15. Старайтесь вести подробные записи, используйте комментарии. Помните: все, что вы не описали сейчас, возможно, потом снова придется долго и мучительно расшифровывать.

16. Подумайте, не может ли что-то из решенной задачи быть полезно в дальнейшем. Если это так, запишите универсальную часть решения более подробно, чем обычно.

7. Задачи для самостоятельного решения

Ниже приводится список из 50 задач, предназначенных для самостоятельного решения. Это достаточно разнообразные задачи самого различного уровня сложности. Они составлены, пользуясь терминологией известной книги Юлиана Семенова, как "информация к размышлению" – т.е. это некие образцы, глядя на которые легко придумать серию похожих задач необходимой для учителя сложности.

Следует также иметь в виду, что практически каждая задача допускает уточнения в формулировке, что позволяет получить несколько задач с одинаковым "сюжетом", но разного уровня сложности. Рассмотрим для примера самую первую задачу: **"Тело равномерно движется со скоростью V. Вычислить путь, проходимый им за время T"**. Ее можно потребовать решить для случаев, когда V и T хранятся в регистрах процессора, в ячейках памяти или вводятся с клавиатуры. И это далеко не полный перечень возможностей. Кроме того, ее можно сделать циклической, потребовав построить таблицу значений V(T), добавить контроль корректности данных (T>0!) и т.п.

Автор считает, что каждый учитель сумеет составить себе задачи необходимой для его конкретных учеников

сложности, а данный список будет неплохим помощником в этом.

Вот образцы задач, которые можно решать на учебном микрокомпьютере "E97".

1. Тело равномерно движется со скоростью V. Вычислить путь, проходимый им за время T.

2. Написать программу вычисления периметра и площади квадрата со стороной A.

3. Написать программу для вычисления количества точек на экране, если известны его горизонтальный и вертикальный размеры. Дополнительно рассмотреть случай, когда пиксел экрана не совпадает с точкой, например, 1 пиксел представляет собой квадрат 2 x 2.

4. Размер экрана M x N точек. Сколько байт занимает в видеопамяти черно-белый рисунок в 1/4 экрана?

5. Дискета IBM "старого" формата имела 40 дорожек по 9 секторов, причем запись ведется на обе стороны диска. Сколько всего секторов на дискете? Сколько это составляет килобайт, если объем каждого сектора равен 512 байт? Попробуйте найти в литературе данные о других форматах дискет и рассчитайте их объем.

6. Сколько секторов по 512 байт займет L текстовых экранов из 24 строк по 80 символов?

7. Вычислить подкоренное выражение для формулы Герона S = p(p-a)(p-b)(p-c), где a, b, c - стороны треугольника, а p - его полупериметр.

8. Написать программу, выводящую на экран код нажатой клавиши. Дополнительно реализовать возможность вывода обоих кодов клавиш расширенной клавиатуры.

9. Решить задачу, обратную предыдущей, т.е. выводящую символ по заданному коду.

10. Написать программу, которая вместо вводимого с клавиатуры символа выводит следующий за ним в алфавите ЭВМ (или предыдущий).

11. Ввести два символа и поместить их в ОЗУ так, чтобы они попали в одно слово. Посмотреть, какой символ попал в старший, а какой в младший байт.

12. Найти сумму кодов двух введенных символов. Убедиться, что сумма кодов "1" и "2" не равняется 3.

13. Определить, делится ли заданное число на K без остатка.

14. Вычислить $(-1)^N$, используя анализ четности N.

15. Определить, войдет ли содержимое N текстовых экранов (24x80) на диск, если на нем имеется свободное место из B байт.

16. Хватит ли ОЗУ "E97" для размещения массива KxN целых чисел, если под программу его обработки требуется 40h байт?

17. На диске хранится черно-белый рисунок M1xN1 точек. Поместится ли он на экран размером M2xN2 полностью? Возможное усложнение: рисунок 16-цветный.

18. Является ли данный символ шестнадцатеричной цифрой и если да, то перевести его в число. Дополнительно можно учесть наличие заглавных и строчных латинских букв.

19. Ввести адрес ОЗУ A и целое число K; занести K в адрес A (аналог оператора POKE в языке BASIC). Контролировать корректность адреса (дополнительно заблокировать ввод внутрь самой программы!)

20. Написать программу, которая выводит на экран содержимое адреса памяти A (аналог функции PEEK). Предусмотреть проверку адреса на корректность, учесть наличие ПЗУ.

21. Напишите программу, которая при нажатии на клавишу F1 выводит текст "LIST", а по F2 – "RUN" и переводит строку. Расширьте список функциональных клавиш по своему усмотрению.

22. Сравнить числа A и B и вывести результат сравнения в виде A>B, A<B или A=B.

23. Для трех введенных чисел a, b и c напечатать максимальное и минимальное.

24. Вывести заданные числа a, b и c в порядке возрастания (убывания).

25. Прокомментировать введенную оценку от 2 до 5. Предусмотреть случай неправильных исходных данных.

26. Написать программу, определяющую, к какой из групп (цифры, латинские буквы и т.д.) относится введенный символ?

27. Вычислить сумму числовой последовательности

$$1 + 3 + 5 + 7 + ... + (2n-1)$$

и убедиться, что она равна n^2.

28. Вычислить сумму числовой последовательности

$$1 + 8 + 16 + 32 + ... + 8(n-1)$$

и убедиться, что она равна $(2n-1)^2$.

29. Написать программу вычисления N! В качестве усложнения задачи добавить контроль N, позволяющий избегать переполнения.

30. Числа Фибоначчи определяются по формуле

$$U_{n+1}= U_{n-1} + U_n \quad (U_1 = U_2 = 1)$$

т.е. каждый последующий член равен сумме двух предыдущих. Составьте программу вычисления числа Фибоначчи с заданным номером. Найдите сумму чисел Фибоначчи, не превышающих заданного K.

31. Определить цифры, из которых состоит четырехзначное число (это можно сделать путем последовательного деления на 10 и нахождения остатка).

32. Написать программу нахождения приближенного значения квадратного корня из X методом подбора такого минимального целого Y, что Y * Y >= X.

33. Написать программу вывода целого числа в двоичной, восьмеричной, десятичной, шестнадцатеричной системах счисления.

34. Написать программу ввода двоичного числа и его преобразования к восьмеричному, шестнадцатеричному и десятичному виду.

35. Заполнить символом "@" область памяти с адресами 50-6F.

36. Заполнить область памяти с адресами 50-6F числом 0 (т.е. заполнять ***словами, а не байтами***).

37. Поменять местами элементы массива с заданными номерами.

38. Подсчитать количество положительных, отрицательных и нулевых элементов целочисленного массива.

39. Найти сумму всех отрицательных и всех положительных элементов массива.

40. В числовом массиве "перенести в конец" все нулевые элементы, например: 1 0 0 2 0 3 ==> 1 2 3 0 0 0.

41. Результаты соревнований по метанию мяча хранятся в памяти ЭВМ в виде массива. Упорядочить его по убыванию.

42. Ввести в память посимвольно некоторый текст, определить его длину и напечатать "задом наперед".

43. Определить, является ли данный текст палиндромом (примером является слово "*казак*", которое читается "задом наперед").

44. Во введенный текст, начиная с заданной позиции, вставить заданное слово.

45. Заменить в тексте один указанный символ на другой.

46. Удалить из текста все цифры.

47. Составить программу шифровки и дешифровки чисел. Для этого создать специальную таблицу кодирования, в которой хранятся сведения, какую цифру каким символом заменять.

48. В тексте самую первую букву "О" поменяйте местами с самой первой буквой "А". Например, слово "ПОРАХОД" должно быть преобразовано в "ПАРОХОД". Для поиска буквы попробуйте применить подпрограмму.

49. Используя подпрограмму нахождения факториала, вычислите выражение (K! + L!) (M! - N!).

50. Используя подпрограмму вывода числа в шестнадцатеричной системе счисления, напишите программу, которая выводит на экран саму себя в виде:

<адрес> <команда>

Убедитесь, что выводимые коды действительно представляют собой вашу программу.

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ 1. Представление чисел в десятичной, двоичной и шестнадцатеричной системах счисления.

10	2	16
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	0001 0000	10

ПРИЛОЖЕНИЕ 2. Система команд процессора “E97”.

МОД	КОП	ОП1	ОП2	ПОЯСНЕНИЯ
X	**0**	X	X	нет операции
X	**1**	X	X	оп1 ==> оп2
X	**2**	X	X	оп2 + оп1 ==> оп2
X	**3**	X	X	оп2 - оп1 ==> оп2
X	**4**	X	X	оп2 - оп1 (сравнить)
X	**5**	X	X	оп2 * оп1 ==> оп2
X	**6**	X	X	оп2 / оп1 ==> оп2
X	**7**	X	X	оп2 AND оп1 ==> оп2
X	**8**	X	X	оп2 OR оп1 ==> оп2
X	**9**	X	X	оп2 XOR оп1 ==> оп2
X	**A**	X	X	порт1 ==> оп2
X	**B**	X	X	оп1 ==> порт2
X	**C**	X	X	переход по адресу
X	**D**	X	X	переход по смещению
X	**E**	*	X	(одноадресные операции)
X	**F**	X	X	останов

МОД	КОП	ОП1	ПОЯСНЕНИЯ
X	**E 1**	X	NOT оп1
X	**E 2**	X	оп1 ==> стек
X	**E 3**	X	стек ==> оп1
X	**E 4**	X	SP + оп1 ==> SP
X	**E 5**	X	SP - оп1 ==> SP
X	**E 6**	X	оп1 ==> SP
X	**E 7**	X	SP ==> оп1
X	**E 8**	0	PS ==> стек
X	**E 9**	0	стек ==> PS
X	**E A**	X	сдвиг влево оп1
X	**E B**	X	сдвиг вправо оп1
X	**E C**	X	арифметический сдвиг вправо оп1

Приложение 2 (продолжение).

Двоичные коды модификатора МОД для переходов.

0000 - возврат из подпрограммы
0001 - безусловный переход
0010 - N = 0 (>=0)
0011 - N = 1 (<0)
0100 - Z = 0 (<>0)
0101 - Z = 1 (=0)
0110 - N=1 or Z=1 (<=0)
0111 - N = 0 and Z = 0 (>0)
1001 - вызов подпрограммы.

Важные команды с “короткой константой”.

XX10 0001 0000 XXXX - очистить оп1
XX10 0010 0001 XXXX - +1 в оп1
XX11 0010 0001 XXXX
или
XX10 0011 0001 XXXX - -1 из оп1
XX10 0100 0000 XXXX - сравнить 0 с оп1
XX11 0101 0001 XXXX - *(-1) оп1
XX10 0111 0001 XXXX - оп1 MOD 2 ==> оп1

ПРИЛОЖЕНИЕ 3. Кодирование операндов в “E97”.

0	**00 00**	R0	[8]	**10 00**	резерв
1	**01**	R1	[9]	**01**	резерв
2	**10**	R2	[A]	**10**	резерв
3	**11**	R3	[B]	**11**	резерв
4	**01 00**	(R0)	[C]	**11 00**	резерв
5	**01**	(R1)	D	**01**	константа
6	**10**	(R2)	E	**10**	адрес ОЗУ
7	**11**	(R3)	[F]	**11**	резерв

ПРИЛОЖЕНИЕ 4. ПЗУ1 учебного компьютера “E97”.

(C) ESC. E97 ROM. Автор E.A.Еремин
Последняя редакция кода - 21.05.97
LoROM=4000h!!!

п/п **DIV/MOD** (деление с нахождением остатка):
адрес LoRom+0000

Рабочая формула: A mod B = A - (A div B) * B

IN: R1,R2; **OUT**: R1=R1 DIV R2; R2=R1 **MOD** R2; R0,R3 сохраняются

```
0E20   R0 ==> стек: сохранить R0 в стеке
0110   R1 ==> R0
0621   R1 div R2 ==> R1
0512   R1 * R2 ==> R2
0320   R0 - R2 ==>R0
0102   R0 ==> R2
0E30   стек ==> R0: восстановить из стека R0
0D00   возврат из п/п
```

п/п **ABS** (вычисление модуля): LoRom+0010h

Рабочая формула: при A<0 abs(A)=not(A)+1; иначе abs(A)=A

IN: R1; **OUT**: R1=ABS(R1); R0,R2,R3 сохраняются

```
2401    сравнить R1 с 0 (0 - "короткая константа")
3D02    если <0, то pc = pc+2: к вычислению модуля
0D00    возврат из п/п
0E11    NOT R1 ==> R1
2211    +1 в R1 (1 - "короткая константа")
0D00    возврат из п/п
```

п/п **IntToStr** (преобразование числа в текстовой вид): LoROM+001Ch

Цифры в десятичном текстовом представлении числа получаются путем его циклического деления на 10. Незначащие нули перед числом заменяются пробелами, например: "-0070” ==> " -70"

IN: R1-integer; R3-адрес строки; **OUT**: нет;
R0-R3 сохраняются

```
0E20    R0 ==> стек
0E21    R1 ==> стек: сохраним регистры
0E22    R2 ==> стек
2401    сравнить R1 с 0 (0 - "короткая константа")
3D06    если <0, то pc = pc+6: к занесению в строку знака минус
C1D7    20b ==> (R3)b: занести в строку входящую в команду байтовую константу "пробел" (R1>=0)
0020    константа "пробел"
1D06    pc = pc+6: обход занесения знака минус и взятия модуля
C1D7    2Db ==> (R3)b: занести в строку входящую в команду байтовую константу "минус" (R1<0)
002D    константа "минус"
9DDE    вызов п/п ABS (взять модуль R1)
2150    5 ==> в R0 (5 - "короткая константа"): должно быть 5 цифр
0203    R0 + R3 ==> R3: +5 в R3, чтобы он показывал на конец текста!
21A2    10 ==> R2 (10 - "короткая константа"): для перевода в 10 с/с
9DC6    вызов п/п DIV/MOD (в R2 - остаток от деления на 10, т.е. десятичная цифра)
02D2    R2 + 30 ==> R2: число ==> код цифры, например, 3 ==> "3"
0030    константа 30
C127    R2 ==> (R3): байт с цифрой запомнить
2313    -1 из R3 (переход к очередной цифре)
```

2310 -1 из R0 (счетчик цифр)
4DF0 если <>0, то pc = pc-10h: к 10 ==> R2 и повторению цикла
2140 4 ==> R0 - подавить можно не более 4-х незначащих нулей спереди
0131 R3 ==> R1 (начало текста) - указатель на текущий символ
0132 R3 ==> R2
2212 +1 в R2 - указатель на следующий символ
C4D6 сравнить 30b с (R2)b: символ "0"?
0030 константа 30
4D0C если <>0, то pc = pc+0Ch: найдена первая значащая цифра - выход
C156 (R1)b ==> (R2)b - перенести знак числа ("-" или "пробел")
C1D5 20b ==> (R1)b: "пробел" в строку - стереть старый знак
0020 константа 20
2211 +1 в R1: подготовить адрес знака для проверки следующей цифры
2310 -1 из R0: счетчик
4DEC если <>0, то pc = pc-14h: к +1 в R2 и повторению цикла
0E32 стек ==> R2
0E31 стек ==> R1: восстановим регистры
0E30 стек ==> R0
0D00 возврат из п/п

п/п **WriteInteger** (вывод целого числа): LoROM+0068

IN: R1 - число; R3 - адрес строки (т.е. буфера); **OUT**: нет; R0-R3 сохраняются

9DB2 вызов п/п IntToStr [расчет адреса: 6A+B2=1C] (вернет строку по адресу в R3)
0E23 R3 ==> стек
0E22 R2 ==> стек
2162 6 ==> R2 (6 - "короткая константа"): длина строки

```
9D06    вызов п/п WriteString: вывод строки
0E32    стек ==> R2
0E33    стек ==> R3
0D00    возврат из п/п
```

п/п **WriteString** (вывод строки на дисплей): LoROM+0078

IN: R2 - количество символов; R3 - адрес строки; **OUT**: нет; **MOD:** R2 и R3 (R3 указывает на следующий за текстом байт!); R0 и R1 сохраняются

```
0E20    R0 ==> стек
C170    (R3)b ==> R0b: очередной символ в R0
9D0A    вызов п/п OUTSYM: вывод символа на дисплей
2213    +1 в R3: адрес следующего символа
2312    -1 из R2: счетчик символов
4DF6    если <>0, то pc = pc-0Ah: к (R3)b ==> R0b и повторению
0E30    стек ==> R0
0D00    возврат из п/п
```

п/п **OUTSYM** (вывод символа на дисплей): LoROM+0088

IN: R0 - символ; **OUT**: нет; R0-R3 сохраняются

```
0E21    R1 ==> стек
0A21    порт 2 ==> R1: чтение порта состояния
E401    сравнить 0b с R1b (0 - "короткая константа")
2DFA    если >=0, то pc = pc-6: снова читать порт состояния (готовность - старший бит МЛАДШЕГО БАЙТА = 1, т.е. байт<0!)
0B03    R0 ==> порт 3: собственно вывод символа в порт данных
0E31    стек ==> R1
0D00    возврат из п/п
```

п/п **MovMas** (перепись массива): LoROM+0096

IN: R1,R2 - начало исходного и результирующего массивов
R0 - количество байт в массиве; **OUT:** нет;
MOD: все регистры, кроме R3

```
0421  сравнить R1 с R2
2D14  если >=0,то pc = pc+14: к MovMas+, иначе подготов-
      ка к MovMas-
0201  R0+R1 ==> R1
2311  -1 из R1 (конец 1 массива)
0202  R0+R2 ==>R2
2312  -1 из R2 (конец 2 массива)
```

п/п **MovMas-** (перепись массива по убыванию адресов): LoROM+00A2

IN: R1, R2 - конец (последний байт) исходного и результирующего массивов, R0 - количество байт в массиве;
OUT: нет; **MOD:** все регистры, кроме R3

```
C156  (R1)b ==> (R2)b: перепись очередного байта
2311  -1 из R1: адрес первого массива
2312  -1 из R2: адрес второго массива
2310  -1 из R0: счетчик байтов
4DF6  если <>0, то pc = pc-10: к повторению цикла перепи-
      си
0D00  возврат из п/п
```

п/п **MovMas+** (перепись массива по возрастанию адресов): LoROM+00AE

IN: R1, R2 - начало исходного и результирующего массивов, R0 - количество байт в массиве; **OUT**: нет;
MOD: все регистры, кроме R3

```
C156   (R1)b ==> (R2)b: перепись очередного байта
2211   +1 в R1: адрес первого массива
2212   +1 в R2: адрес второго массива
```

```
2310    -1 из R0: счетчик байтов
4DF6    если <>0, то pc = pc-10: к повторению цикла пере-
        писи
0D00    возврат из п/п
```

Библиотека исполнения для Паскаль-программ

п/п **WriteChar** (вывод значение типа CHAR):
LoROM+00BA

(OUTSYM выводит символ из R0, а Паскалю удобнее из R1!)

IN: R1 - символ; **OUT:** нет; R0-R3 сохраняются

```
0E20    R0 ==> стек
0110    R1 ==> R0: символ
9DC8    вызов п/п OUTSYM: вывод символа [расчет адреса:
        C0+C8=88]
0E30    стек ==> R0
0D00    возврат из п/п
```

п/п **WriteBoolean** (выводит значение типа BOOLEAN):
LoROM+00C4

При R1=0 выводится TRUE, иначе - FALSE

IN: R1 - значение; **OUT:** нет; R0,
MOD: R2, R3; R1 сохраняется

```
E401    сравнить R1b с 0b (0 - "короткая константа")
5D0A    если =0, то pc = pc+0A: к выводу TRUE
9D12    вызов п/п WritePasString: вывести текст, идущий
        далее
5404    (4)T ┐
5552    RU   ├ текст TRUE (4 - его длина)
0045    E(0) ┘
0D00    возврат из п/п
9D08    вызов п/п WritePasString: вывести текст, идущий
        далее
```

4605 (5)F
4C41 AL — текст FALSE
4553 SE
0D00 возврат из п/п

п/п **WritePasString** (вывод текста, находящегося после вызова п/п): LoROM+00DC

IN: в стеке - адрес следующего за командой вызова п/п слова, т.е. адрес начала текста! **OUT**: R2=0; R3 - адрес следующего за текстом слова; **MOD:** R2, R3; R0, R1 сохраняются

0E33 стек ==> R3: прочитать из стека адрес текста в R3
8172 (R3)b ==> R2: длина текста из байта преобразуется в слово!
2213 +1 в R3: перейдем к адресу начала текста
9D94 вызов п/п WriteString: вывод строки [расчет перехода: E4+94=78]

(далее установим R3 на "ближайший следующий" четный адрес, т.к. это адрес следующей за текстом команды:)

2213 +1 к R3
07D3 R3 and FFFE ==> R3 (сбросить младший бит)
FFFE константа FFFE
1C03 R3 ==> PC: перейти к следующей за текстом команде

П/п **LN** (вывод CR/LF - перевод строки): LoROM+00EC

IN, **OUT**: нет; R0-R3 сохраняются

0E20 R0 ==> стек
21D0 13 ==> R0 (13 - "короткая константа"): "возврат каретки" CR
9D96 вызов п/п OUTSYM: вывод символа [расчет перехода: F2+96=88]
21A0 10 ==> R0 (10 - "короткая константа"): "перевод строки" LF

9D92 вызов п/п OUTSYM: вывод символа [расчет перехода: F6+92=88]
0E30 стек ==> R0
0D00 возврат из п/п

ПОДПРОГРАММЫ ВВОДА

П/п **INSYMe** (ввод символа с эхо-печатью): LoROM+00FA **IN**: нет; **OUT**: R0 - символ; R1-R3 сохраняются

9D02 вызов п/п INSYM: ввод символа с клавиатуры
1D8A переход к OUTSYM: к выводу [расчет перехода: FE+8A=88]

П/п **INSYM** (ввод символа без эхо-печати): LoROM+00FE **IN**: нет; **OUT**: R0 - символ; R1-R3 сохраняются

0A00 порт 0 ==> R0: чтение порта состояния
E400 сравнить R0b с 0b (0 - "короткая константа")
2DFA если >=0, то pc = pc-6: снова читать порт состояния (готовность - старший бит МЛАДШЕГО БАЙТА = 1, т.е. байт<0!)
0A10 порт 1 ==> R0: собственно ввод из порта данных клавиатуры
0D00 возврат из п/п

ОБРАЩЕНИЯ К ПЗУ2

П/п **Input_Integer** (ввод целого числа): LoROM+0108 **IN**: нет; **OUT**: R1 - введенное число; R0, R2, R3 - сохраняются

1D78переход на 10A+78=182

П/п **Input_Boolean** (ввод значения типа BOOLEAN):
LoROM+010A

IN: нет; **OUT**: R1 = 0 если введено TRUE и -1 если FALSE;
R0, R2, R3 - сохраняются

1D74 переход на 10C+74=180

РЕЗЕРВ

0000 ...

ПРИЛОЖЕНИЕ 5. ПЗУ2 учебного компьютера “E97”.

```
{@@@@@@@@ "содержимое" ПЗУ2 @@@@@@@@}
procedure showerror(d:integer);
var l:integer;
begin l:=d;
     while l>1 do
          begin write(' ');l:=l-1 end;
          writeln('^ошибка')
        end;

procedure input_integer;
{условный адрес входа - LoROM+182}
var d,l,s,x0:integer;
begin
     repeat{повторение, пока не введено правильное  число}
     x0:=WhereX-1;{запомнить позицию курсора}
     readln(comstr); if computer='UKNC' then writeln;
     l:=length(comstr);
     if comstr[1]='-'
       then begin s:=-1; delete(comstr,1,1); l:=l-1 end
       else s:=1;
     if (l>5) or ((l=5) and (comstr>'32767')) {переполнение}
       then begin d:=6; if s=-1 then d:=d+1 end
       else val(comstr,r[1],d) {перевод из string в integer};
```

```
    if d<>0 then showerror(x0+d){индикация ошибки}
    until d=0;
r[1]:=s*r[1] {учесть знак}
end;

procedure input_boolean;
{условный адрес входа - LoROM+180}
var d,c,l,x0:integer;
   obrazec:string[6];
begin
    repeat {повторение, пока не введен правильный текст}
    x0:=WhereX-1;{запомнить позицию курсора}
    readln(comstr); if computer='UKNC' then writeln;
    for c:=1 to length(comstr) do
       {привести к заглавным буквам}
       if (comstr[c]>='a') and (comstr[c]<='z')
         then comstr[c]:=chr(ord(comstr[c]) and $5F);
    if comstr[1]='T' then obrazec:='TRUE'
                  else obrazec:='FALSE';
    l:=length(obrazec); d:=1;
    while (comstr[d]=obrazec[d]) and (d<=length(comstr))
        do d:=d+1; {сравнение текста с образцом}
    if (d=l+1) and (length(comstr)=l) then d:=0
      {тексты совпали и их длины равны - Ok};
    if d<>0 then showerror(x0+d) {индикация ошибки}
    until d=0;
if obrazec='TRUE' then r[1]:=0 else r[1]:=-1;
end;
{конец ПЗУ2}
```