ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
Государственное образовательное учреждение высшего профессионального образования
“ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ”

Лабораторная работа №3

Система команд микропроцессора

Учебное пособие к выполнению лабораторных работ по дисциплине «Микропроцессорная техника» для студентов ФТФ специальности 140306.

Томск – 2004

УДК 681.322

Горюнов А.Г. Ливенцов С.Н. Система команд микропроцессора: Учеб. пособие.

Учебное пособие посвящено системе команд микроконтроллеров популярного семейства MCS51. Пособие содержит методику разработки прикладных программ на языке ассемблера для микроконтроллеров на примере MCS51, а также примеры программ для практического закрепления знаний основ микропроцессорной техники. Данное учебное пособие ориентировано на курс лабораторных работ с использованием учебно-лабораторных стендов SDK-1-1.

Пособие подготовлено на кафедре «Электроника и автоматика физических установок» ТПУ и предназначена для студентов очного обучения специальности 200600.

Учебное пособие рассмотрено и рекомендовано
к изданию методическим семинаром кафедры электроники и автоматики
физических установок "___"_____ 2004г.

Зав. кафедрой
доцент, к.т.н. _____ В. Ф. Дядик

Содержание

1 Цель работы

Целью работы является изучение системы команд микроконтроллеров популярного семейства Intel 8051, а также продолжение начатого в лабораторной работе №2 изучения интегрированной среды Keil uVision фирмы Keil Software Inc.

2 Содержание работы

1. Изучение системы команд микроконтроллера Intel 8051 [1].
2. Проработка примеров, иллюстрирующих особенности системы команд микроконтроллера Intel 8051.
3. Выполнение индивидуального задания.

3 Методика разработки прикладного программного обеспечения микроконтроллерных систем. Программирование микроконтроллера Intel 8051 на языке ассемблера

3.1 Общие сведения

Если задача на разработку прикладной программы для микроконтроллера поставлена, то для получения текста исходной программы необходимо выполнить ряд последовательных действий.

1. Подробно описать задачу.
2. Проанализировать задачу.
3. Выполнить инженерную интерпретацию задачи, желательно с привлечением того или иного аппарата формализации (граф автомата, сети Петри, матрицы состояний и связности и т.п.).
4. Разработать общую схему алгоритма работы контроллера.
5. Разработать детализированные схемы отдельных процедур, выделенных на основе модульного принципа составления программ.
6. Детально проработать интерфейс контроллера и внести исправления в общую и детализированные схемы алгоритмов.
7. Распределить рабочие регистры и память.
8. Сформировать текст исходной программы.

В результате работы по трем первым пунктам данного перечня получают так называемую функциональную спецификацию прикладной программы, в которой основное внимание уделяется детализации способов формирования входной и выходной информации.

На языке схем алгоритмов разработчик описывает метод, выбранный им для решения поставленной задачи. Довольно часто бывает, что одна и та же задача может быть решена различными методами. Способ решения задачи, выбранный на этапе её инженерной интерпретации, на основе которого формируется схема, определяет не только качество разрабатываемой прикладной программы, но и качественные показатели конечного изделия.

Алгоритм есть точно определенная процедура, предписывающая контроллеру однозначно определенные действия по преобразованию исходных данных в обработанные выходные данные. Поэтому разработка схемы требует предельной точности и однозначности используемой атрибутики: символических имен переменных, констант, подпрограмм (модулей), символических адресов таблиц, портов ввода вывода и т.п. Основное внимание при разработке следует уделить тому разделу функциональной спецификации прикладной программы, в котором приводится описание аппаратуры сопряжения с объектом управления. Это описание должно быть детализировано вплоть до электрических и временных характеристик каждого входного и выходного сигнала или устройства.

Успех разработки прикладной программы заключается в использовании метода декомпозиции, при котором вся задача последовательно разделяется на меньшие функциональные модули. Каждый из модулей можно анализировать, разрабатывать и отлаживать отдельно от других. При выполнении прикладной программы в микроконтроллере управление без всяких двусмысленностей передается от одного функционального модуля к другому. Схема связности этих функциональных модулей, каждый из которых реализует некоторую процедуру, образует общую схему алгоритма прикладной программы. Язык графических образов схемы алгоритма можно использовать на любом уровне детализации описания модулей вплоть до того, что каждому оператору схемы будет соответствовать единственная команда микроконтроллера.

Структурное программирование есть процесс построения прикладной программы из набора программных модулей, каждый из которых реализует определенную процедуру

обработки данных. Программные модули должны иметь только одну точку входа и одну точку выхода. Только в этом случае отдельные модули можно разрабатывать и отлаживать независимо, а затем объединять в законченную прикладную программу с минимальными проблемами их взаимосвязи.

Источником подавляющего большинства ошибок программирования является использование модулей, имеющих один вход и несколько выходов. При необходимости организации множественных ветвлений в программе декомпозицию задачи выполняют таким образом, чтобы каждый функциональный модуль имел только один вход и один выход. Для этого условные операторы (имеющие два выхода) или включают внутрь модуля, объединяя их с операторами обработки, или выносят в систему межмодульных связей, формируя тем самым схему алгоритма более высокого ранга.

Разработка схемы алгоритма функционального модуля программы имеет ярко выраженный итеративный характер, т.е. требует многократных проб, прежде чем возникает уверенность, что алгоритм реализации процедуры правильный и завершённый. Вне зависимости от функционального назначения процедуры при разработке её схемы необходимо придерживаться следующей очередности работы.

1. Определить, что должен делать модуль.
2. Определить способы получения модулем исходных данных (от датчиков через порты ввода, из таблиц в памяти или через рабочие регистры).
3. Определить необходимость какой-либо предварительной обработки введенных исходных данных (маскирование, сдвиг, масштабирование, перекодировка).
4. Определить метод преобразования входных данных в требуемые выходные. Используя операторы процедур и условные операторы принятия решения, отобразить на языке схемы алгоритма выбранный метод.
5. Определить способы выдачи из модуля обработанных данных (передать в память, в вызывавшую программу или в порты вывода).
6. Определить необходимость какой-либо вторичной обработки выводимых данных (изменение формата, перекодирование, масштабирование, маскирование).
7. Вернуться к пункту 1 настоящего перечня и проанализировать полученный результат. Выполнить итеративную корректировку схемы алгоритма с целью сделать её простой, логичной, стройной и обладающей чётким графическим образом.
8. Проверить работоспособность алгоритма на бумаге путем подстановки в него действительных данных. Убедиться в его сходимости и результативности.
9. Рассмотреть предельные случаи и попытаться определить граничные значения информационных объектов, с которыми оперирует алгоритм, за пределами которых он теряет свойства конечности, сходимости или результативности. Особое внимание при этом следует уделить анализу возможных ситуаций переполнения разрядной сетки, изменения знака результата операции, деления на переменную, которая может принять нулевое значение.
10. Провести мысленный эксперимент по определению работоспособности алгоритма в реальном масштабе времени, когда стохастические события, происходящие в объекте управления, могут оказать влияние на работу алгоритма. При этом самому тщательному анализу следует подвергнуть реакцию алгоритма на возможные прерывания с целью определения критических операторов, которые необходимо защитить от прерываний. Кроме того, в ходе этого мысленного эксперимента следует проанализировать логику алгоритма с целью определения таких последовательностей операторов, при выполнении которых микроконтроллер может “не заметить” кратковременных событий в объекте управления. При обнаружении таких ситуаций в логику следует внести коррективы.

Практика разработки программного обеспечения показала, что последовательное использование описанной поэтапной процедуры, составляющей основу метода структурного

программирования, позволяет уверенно получать работоспособные прикладные программы.

Преобразование разработанной схемы алгоритма в исходный текст программы дело несложное. Но прежде чем приступить к написанию программы необходимо специфицировать память и выбрать язык программирования.

Спецификация памяти и рабочих регистров заключается в определении адреса первой команды прикладной программы, действительных начальных адресов стека, таблиц данных, буферных зон передачи параметров между процедурами, подпрограмм обслуживания прерываний и т.д. При этом следует помнить, что в микроконтроллерах память программ и память данных физически и логически разделены.

Диапазон языков написания исходного текста прикладной программы простирается от машинного кода до почти естественного языка. В машинном коде или на языке ассемблера программировать труднее, чем на алгоритмическом языке высокого уровня, но зато получается более короткий код программы, требуется меньшая ёмкость памяти программы и выполняется такая программа быстрее.

Объектные коды, полученные путем трансляции исходных программ, написанных на алгоритмическом языке высокого уровня, занимают в памяти больше места и требуют большего времени на исполнение. Выбор языковых средств составления исходных программ в каждом конкретном случае зависит от характеристик прикладной задачи, опыта программиста и допустимых затрат на разработку.

По мнению [2], огромное большинство прикладных задач управления объектами вследствие того, что они должны решаться в реальном времени, предъявляет столь высокие требования по быстродействию, что для их решения основным языковым средством написания прикладных программ еще долгие годы будет оставаться язык ассемблера. Это положение о преимущественном использовании языка ассемблера подкрепляется и тем обстоятельством, что однокристальные микроконтроллеры имеют ограниченный объем резидентной памяти программ и, следовательно, критичны к длине прикладных программ.

3.2 Процедуры и подпрограммы

При разработке микроконтроллерных систем могут быть использованы два способа организации прикладных программ: монолитный и модульный. При первом способе вся прикладная программа разрабатывается как единое целое. При втором она строится из отдельных программных блоков, каждый из которых реализует некоторую процедуру обработки данных или управления. Взаимосвязь блоков определяется разработчиком при монтаже из этих блоков законченной прикладной программы.

Отдельные фрагменты прикладной программы могут быть получены в виде линейной последовательности блоков, другие (многократно используемые) обычно оформляются в виде подпрограмм, к которым прикладная программа, называемая основной, имеет возможность обратиться по мере необходимости. Подпрограмма должна обладать следующими свойствами: выполнять законченную процедуру обработки данных, иметь только один вход и один выход и не обладать эффектом последействия, при котором текущее выполнение подпрограммы оказывало бы влияние на её последующие выполнения.

3.2.1 Вызов подпрограммы

Обращение к подпрограмме осуществляется по команде вызова CALL MARK, где MARK – символическое имя процедуры. Имя процедуры используется в качестве метки, отмечающей одну из команд (чаще всего первую) подпрограммы. Для Intel 8051 мнемоническое значение CALL является обобщенным и транслируется в одну из команд ACALL или LCALL в зависимости от адресного расстояния вызываемой подпрограммы.

По команде CALL в стеке сохраняется значение счётчика команд, и возврат из подпрограммы осуществляется в то место основной программы, откуда был осуществлен вызов (к команде основной программы, следующей за командой CALL). Для этого любая под-

программа должна заканчиваться командой возврата RET, осуществляющей восстановление содержимого программного счётчика из стека.

Достаточно часто возникает необходимость такой организации вычислительного процесса, при которой подпрограмма вызывает другую подпрограмму, та в свою очередь вызывает следующую и т.д. Этот процесс называется вложением подпрограмм. Число подпрограмм, которые могут быть вызваны таким образом (глубина вложенности подпрограмм), ограничивается только ёмкостью стека.

3.2.2 Сохранение параметров основной программы

Иногда при обращении к подпрограмме возникает необходимость сохранить не только адрес возврата в основную программу, но и содержимое отдельных рабочих регистров. Удобным способом для этого является переключение банка регистров. Например, если основная программа использует банк регистров 0, то подпрограмма может использовать банк регистров 1. Однако переключение банка регистров не обеспечивает сохранение содержимого аккумулятора, что приводит к необходимости создавать в одном из рабочих регистров или в памяти копию аккумулятора.

3.2.3 Передача параметров

Для успешной работы любой подпрограммы необходимо однозначно определить способ передачи в неё исходных данных и способ вывода результата её работы. Подпрограмма, которой требуется дополнительная информация в виде параметров её настройки или операндов, называется параметризуемой.

Получили распространение четыре способа передачи параметров: через память, через регистры общего назначения, через регистр признаков, и через стек.

При передаче входных параметров через память основная программа обязательно содержит команды загрузки некоторых ячеек памяти, а подпрограмма – команды считывания из этих ячеек. При передаче входных параметров подпрограмма должна загрузить некоторые ячейки памяти, а основная программа – считать.

Передача параметров через регистры осуществляется аналогичным образом.

Третий способ передачи параметров – через регистр признаков – удобно использовать при передаче выходных параметров (например, в подпрограммах сравнения чисел). В этом случае подпрограмма должна установить (или сбросить) соответствующие признаки, а основная программа – проанализировать их значение. Intel 8051 обладает большими возможностями для передачи параметров через признаки. В нём имеется 128 флагов пользователя, доступных для модификации и анализа.

Способ передачи через стек позволяет использовать в качестве параметра содержимое счётчика команд.

Использование процедур, оформленных в виде подпрограмм, при разработке программного обеспечения имеет ряд достоинств. Прежде всего относительно простые модули, выделенные из сложной программы, могут программироваться несколькими разработчиками с целью сокращения времени проектирования. Еще более важным является то, что любая подпрограмма допускает автономную отладку. Это, как правило, многократно сокращает время отладки всего прикладного программного обеспечения. И, наконец, механизм использования подпрограмм уменьшает длину прикладной программы, что имеет своим следствием уменьшение требующейся ёмкости памяти программ.

Существенным является и то обстоятельство, что отлаженные процедуры организуются разработчиками в библиотеки параметризуемых подпрограмм и могут быть многократно использованы в проектной работе. Библиотека подпрограмм должна строиться на основе соглашения о едином способе обмена параметрами.

3.3 Правила записи программ на языке ассемблера

Исходный текст программы на языке ассемблера имеет определенный формат. Каждая команда и директива представляет собой строку:

```
МЕТКА          ОПЕРАЦИЯ  ОПЕРАНД(Ы)     КОММЕНТАРИИ
```

Поля могут отделяться друг от друга произвольным числом пробелов и табуляцией.

3.3.1 Метка

В поле метки размещается символическое имя ячейки памяти, в которой хранится отмеченная команда или операнд. Метка представляет собой буквенно-цифровую комбинацию, начинающуюся с буквы. Используются только буквы латинского алфавита. Ассемблер A51 допускает использование в метках символа подчеркивания (_). Метка всегда завершается двоеточием (:).

Директивы ассемблера не преобразуются в двоичные коды, а потому не могут иметь меток. Исключение составляют директивы резервирования памяти и определения данных (DS, DB, DW). У директив, определяющих символические имена, в поле метки записывается определяемое символическое имя, после которого двоеточие не ставится.

В качестве символических имен и меток не могут быть использованы мнемокоды команд, директив и операторов ассемблера, зарезервированные имена, а также мнемонические обозначения регистров и других внутренних блоков микроконтроллера.

3.3.2 Операция

В поле операции записывается мнемоническое обозначение команды или директивы ассемблера, которое является сокращением (аббревиатурой) полного английского наименования выполняемого действия. Например: MOV – move – переместить, JMP – jump – перейти, DB – define byte – определить байт.

Для микроконтроллера Intel 8051 используется строго определенный и ограниченный набор мнемонических кодов. Любой другой набор символов, размещенный в поле операции, воспринимается ассемблером как ошибочный.

3.3.3 Операнды

В этом поле определяются операнды (или операнд), участвующие в операции. Команды ассемблера могут быть без-, одно- или двухоперандными. Операнды разделяются запятой (,).

Операнд может быть задан непосредственно или в виде его адреса (прямого или косвенного). Непосредственный операнд представляется числом (MOV A, #15) или символическим именем (ADDC A, #OPER2) с обязательным указателем префикса непосредственного операнда (#). Прямой адрес операнда может быть задан мнемоническим обозначением (IN A, P1), числом (INC 40), символическим именем (MOV A, MEMORY). Указанием на косвенную адресацию служит префикс @. В командах передачи управления операндом может являться число (LCALL 0135H), метка (JMP LABEL), косвенный адрес (JMP @A) или выражение (JMP $ - 2, где $ - текущее содержимое счётчика команд).

Используемые в качестве операндов символические имена и метки должны быть определены, а числа представлены с указанием системы счисления, для чего используется суффикс (буква, стоящая после числа): B – для двоичной, Q – для восьмеричной, D – для десятичной и H – для шестнадцатиричной. Число без суффикса по умолчанию считается десятичным.

Ассемблер A51 допускает использование выражений в поле операндов, значения которых вычисляются в процессе трансляции.

Выражение представляет собой совокупность символических имен и чисел, связанных операторами ассемблера. Операторы ассемблера обеспечивают выполнение арифметических (“+” – сложение, “-” – вычитание, “*” – умножение, “/” – целое деление, MOD – деление по модулю) и логических (OR – ИЛИ, AND – И, XOR – исключающее ИЛИ, NOT

– отрицание) операций в формате 2-байтных слов. Например, запись ADD A, #((NOT 13)+1) эквивалентна записи ADD A, #0F3H и обеспечивает сложение содержимого аккумулятора с числом -13, представленным в дополнительном коде.

Широко используются также операторы LOW и HIGH, позволяющие вычислить младший и старший байты 2-байтного операнда.

3.3.4 Комментарий

Поле комментария может быть использовано программистом для текстового или символьного пояснения логической организации прикладной программы. Поле комментария полностью игнорируется ассемблером, а потому в нём допустимо использовать любые символы. По правилам языка ассемблера поле комментария начинается с точки с запятой (;).

3.4 Директивы ассемблера

Ассемблер транслирует исходную программу в объектные коды. Хотя он берет на себя многие из рутинных задач программиста, такие как присвоение действительных адресов, преобразование чисел, присвоение действительных значений символьным переменным и т.п., программист всё же должен указать ей некоторые параметры: начальный адрес прикладной программы, конец ассемблируемой программы, форматы данных и т.п. Всю эту информацию программист вставляет в исходный текст прикладной программы в виде директив, которые только управляют процессом трансляции и не преобразуются в коды объектной программы.

Ассемблер поддерживает ряд директив, которые позволяют дать символическое определение переменным, резервируют и инициализируют пространство памяти, определяют расположение сгенерированного объектного кода в памяти. За исключением DB и DW директивы не производят объектный код. Директивы используются, чтобы изменить состояние ассемблера, определить объекты и добавить информацию к объектному файлу.

Директивы ассемблера могут быть разделены на ряд категорий:

- символические определения,
- резервирование пространства памяти,
- инициализация данных,
- управление состоянием ассемблера,
- выбор сегментов,
- определение макрокоманд.

Далее перечисляются все директивы по категориям и кратко описываются результаты их действия. Основные часто используемые директивы перечислены в алфавитном порядке в приложении 1. Информацию по остальным можно найти в справочной системе Keil uVision [3].

3.4.1 Директивы символических определений

Директивы символических определений могут быть использованы для того, чтобы резервировать пространство памяти, поставить в соответствие символическим именам определённые числовые значения, регистры процессора и сегменты. Эти директивы требуют, чтобы имя символа было определено наряду с адресом, числовым значением, регистром или типом сегмента.

Директива	Описание
BIT	Определяет символическое имя, ссылающееся на адрес бита.
CODE	Определяет символическое имя, ссылающееся на адрес кода.
DATA	Определяет символическое имя, ссылающееся на адрес резидентной памяти данных.
EQU	Назначает символическому имени числовое значение или имя регистра.

IDATA Определяет символическое имя, ссылающееся на косвенно адресуемый адрес резидентной памяти данных.

SEGMENT Объявляет имя перемещаемого сегмента, его тип и расположение.

SET Назначает символическое имя числовому значению или регистру. Имя может быть впоследствии изменено с помощью директивы SET.

XDATA Определяет символическое имя, ссылающееся на адрес внешней памяти данных.

3.4.2 Директивы резервирования и инициализации памяти

Эти директивы используются для резервирования и инициализации слов, байтов или битов. В абсолютном сегменте зарезервированное пространство начинается с текущего адреса. В перемещаемом сегменте зарезервированное пространство начинается с текущего смещения. Указатель расположения поддерживается отдельно для каждого сегмента, к нему можно обращаться, используя символ ($).

Директива Описание

DB Заносит в память программ байтовую константу.

DBIT Резервирует пространство в битовом сегменте.

DS Резервирует пространство памяти в текущем сегменте.

DW Инициализирует память значением слова.

3.4.3 Директивы компоновки программы

Вы можете использовать директивы компоновки программы для того, чтобы дать объектному модулю имя и определить общие и внешние символы. Эти директивы используются в BL51 для объединения отдельных объектных модулей в единый абсолютный объектный модуль.

Директива Описание

EXTRN Определяет символические имена, которые объявлены в других объектных модулях.

NAME Определяет имя объектного модуля.

PUBLIC Определяет символические имена, которые могут использоваться в других объектных модулях.

3.4.4 Директивы управления состоянием ассемблера

Эти директивы используются для того, чтобы сообщить о конце трансляции программы, выбрать начальный адрес или смещение для сегмента, определить используемый банк регистров.

Директива Описание

END Сообщает о конце транслируемого модуля.

ORG Изменяет значение ассемблерного счётчика адреса текущего сегмента программы.

USING Выбирает номер банка регистров общего назначения.

3.4.5 Директивы выбора сегмента

Следующие директивы определяют сегменты данных и кода.

Директива Описание

BSEG Выбирает абсолютный битовый сегмент.

CSEG Выбирает сегмент программы в машинном коде.

DSEG Выбирает абсолютный сегмент резидентной памяти данных.

ISEG Выбирает абсолютный косвенно адресуемый сегмент резидентной памяти данных.

RSEG Выбирает предварительно определенный перемещаемый сегмент.

XSEG Выбирает абсолютный сегмент внешней памяти данных.

3.4.6 Директивы макроопределений

Следующие директивы используются для определения макрокоманд.

Директива	Описание
ENDM	Заканчивает макроопределение.
EXITM	Заставляет макрорасширение немедленно завершиться.
IRP	Определяет список аргументов.
IRPC	Определяет аргумент.
LOCAL	Определяет до 16 локальных символов, используемых внутри макрокоманды.
MACRO	Начало макроопределения, определяет имя макрокоманды и параметров, которые могут быть переданы макрокоманде.
REPT	Определяет количество повторений последующих строк.

3.5 Отладка прикладного программного обеспечения микроконтроллеров

После получения объектного кода прикладной программы наступает этап отладки, т.е. установления факта её работоспособности, а также выявления, локализации и устранения ошибок. Без этого этапа никакое программное обеспечение вообще не имеет права на существование. Отладка программного обеспечения представляет собой отдельную сложную задачу, которая почти не поддается формализации и требует для своего выполнения высокого профессионализма и глубоких знаний разработчика.

Обычно отладка прикладного программного обеспечения осуществляется в несколько этапов. Простые синтаксические ошибки выявляются уже на этапе трансляции.

Далее необходимо выполнить:

- автономную отладку каждой процедуры в статическом режиме, позволяющую проверить правильность проводимых вычислений, правильность последовательности переходов внутри процедуры (отсутствие “зацикливания”) и т.п.;
- комплексную отладку программного обеспечения в статическом режиме, позволяющую проверить правильность алгоритма управления (по последовательности формирования управляющих воздействий);
- комплексную отладку в динамическом режиме без подключения объекта для определения реального времени выполнения программы и её отдельных фрагментов.

Следует иметь в виду, что автономная отладка отдельных модулей настолько проще и эффективнее отладки всей прикладной программы, что переходить к этапу комплексной отладки целесообразно только после исчерпания всех средств автономной отладки.

Вышеперечисленные этапы осуществляются обычно с использованием кросс-систем, к которым и относится изучаемый пакет Keil Software.

В состав кросс-систем входят программы-отладчики, моделирующие выполнение программ, написанных для микроконтроллеров. Такие программные имитаторы позволяют эффективно отлаживать вычислительные процедуры, а также сам алгоритм функционирования контроллера.

Разработчику предоставлен доступ к любому ресурсу микроконтроллера, имеется возможность покомандного и пофрагментного исполнения программ и останова по условию, а также подсчёта числа тактов выполнения тех или иных фрагментов программы, инициирования прерывания, дизассемблирования содержимого памяти программ и т.п.

Кросс-отладчики позволяют промоделировать практически все возможные варианты работы программы и тем самым убедиться в её работоспособности. На этом этапе возможна проверка работоспособности программы в нештатных ситуациях, в условиях поступления некорректных входных воздействий.

Мощные имитаторы должны позволять моделировать объекты и датчики, подключаемые к микроконтроллеру. При этом появляется возможность выполнять комплексную отладку программного обеспечения, не опасаясь, что возможные ошибки в программе, ал-

горитме или некорректные действия оператора приведут к выходу из строя технических средств разрабатываемой системы.

Главным недостатком кросс-систем является невозможность прогона программы в реальном масштабе времени.

Наиболее полная и комплексная отладка прикладного программного обеспечения совместно с аппаратными средствами контроллера может быть произведена на инструментальном компьютере с так называемым внутрисхемным эмулятором.

4 Система команд микроконтроллера Intel 8051

4.1 Выполнение примеров программ

В пошаговом режиме работы Keil uVision выполните исследование приведённых ниже примеров программ. Первый пример содержит подробное описание последовательности действий. Остальные примеры исследуются самостоятельно.

4.1.1 Пример использования команд передачи данных

Пример 1. Запись данных. Записать в резидентную память данных по адресам 41 и 42 число 1C3FH:

```
LOAD:     MOV  R0,#41H   ;загрузка в R0 указателя данных
          MOV  @R0,#1CH  ;запись в память числа 1CH
          INC  R0        ;инкремент указателя
          MOV  @R0,#3FH  ;запись в память числа 3FH
```

Дополним программу директивами и командами ассемблера, обеспечивающими отладку и тестирование:

```
START:    JMP LOAD       ;переход к программе
          ORG  2100H;директива размещения программы с адреса 2100
LOAD:     MOV  R0,#41H   ;загрузка в R0 указателя данных
          MOV  @R0,#1CH  ;запись в память числа 1CH
          INC  R0        ;инкремент указателя
          MOV  @R0,#3FH  ;запись в память числа 3FH
          JMP LOAD       ;зацикливание программы
          END            ;директива окончания трансляции
```

Переходим к созданию проекта и тестированию программы в среде Keil uVision. Большинство шагов Вам уже знакомо по лабораторной работе №2 [4]. Специфика состоит только в том, что текст программы написан на языке ассемблера.

Шаг 1. На своем диске U в папке MPT необходимо создать следующие папки:

LAB3\Prim1

Загрузить программу Keil uVision. Если после загрузки программы открывается предыдущий проект, с которым производилась работа, то необходимо закрыть его.

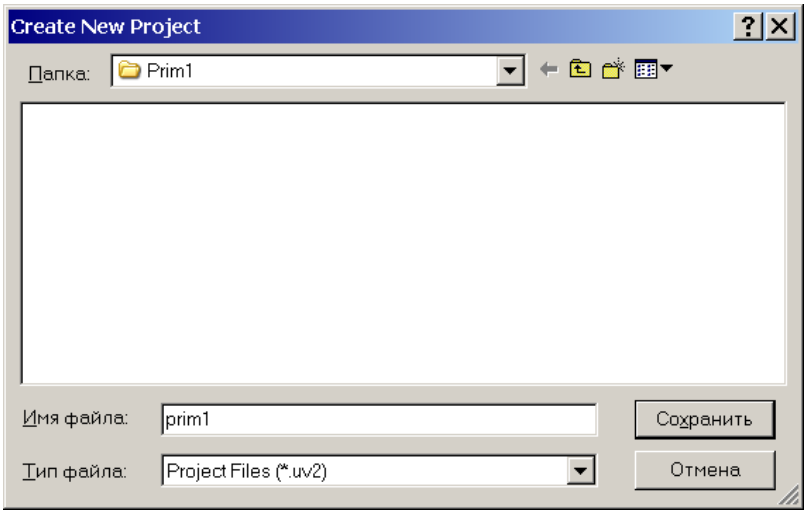

Рис. 1.

Шаг 2. Создать новый проект, используя Project -> New Project и сохранить его в папку U:\MPT\LAB3\Prim1 (рис. 1).

Произвести настройку параметров проекта аналогичным образом как в лабораторной работе №2 [4].

Шаг 3. Создать новый файл для последующего ввода текста программы (рис. 2) и сохранить его в папку проекта с расширением «.a», например «prim1.a» (рис. 3).

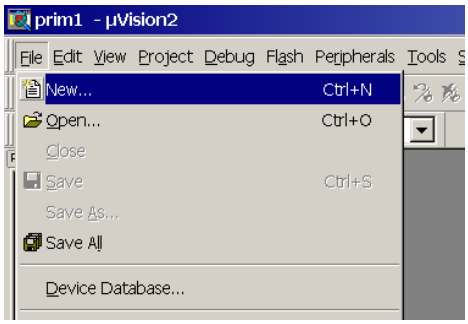

Рис. 2.

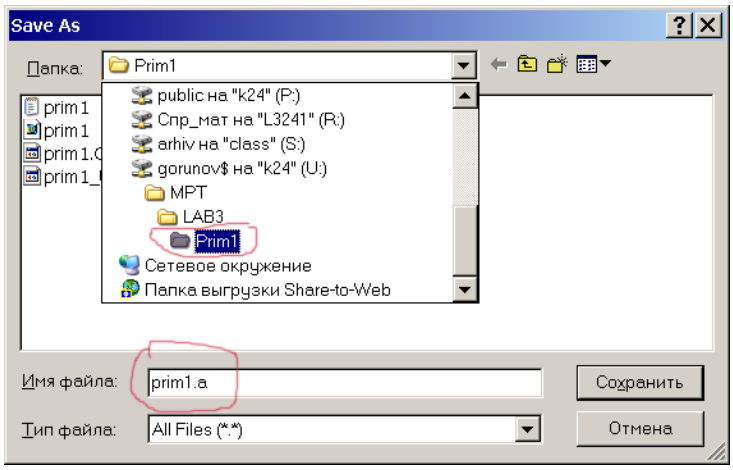

Рис. 3.

Шаг 4. Ввести текст программы примера 1 (см. рис. 4):

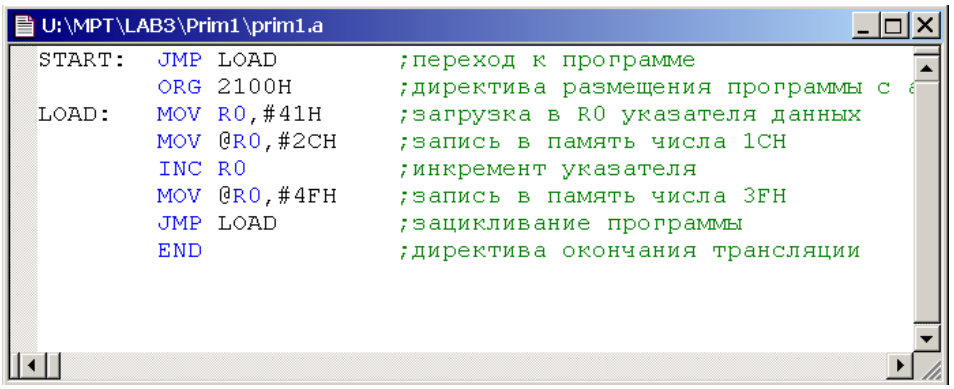

Рис. 4. Текст программы

Шаг 5. Добавить файл с исходным текстом программы в проект аналогичным образом как в лабораторной работе №2 [4]. Далее необходимо выполнить компиляцию проекта, используя Project -> Built Target.

Шаг 6. Запустить отладку программы с помощью команды Debug -> Start/Stop Debug Session.

Шаг 7. Открыть окно памяти с помощью View -> Memory Window (рис. 5). Затем, в открытом окне в поле “Address” ввести начальный адрес «дампа» резидентной памяти данных, например I: 40h (рис. 7).

Дальнейшие шаги. Выполните программу в пошаговом режиме. Наблюдая изменение содержимого окна резидентной памяти данных и окна Project Workspace, где отображаются регистры (рис. 6), убедитесь в правильности работы программы. Определите время выполнения программы.

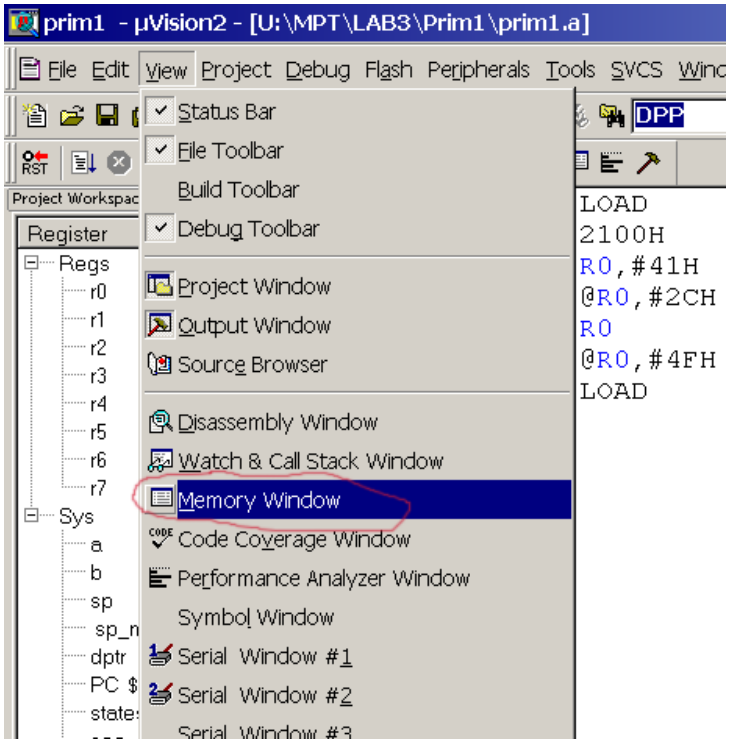

Рис. 5.

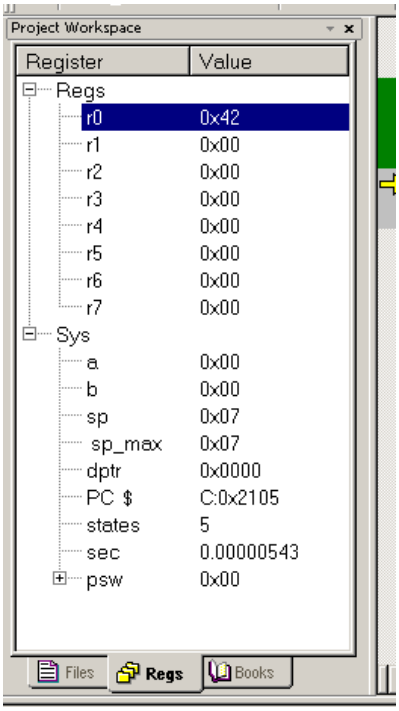

Рис. 6.

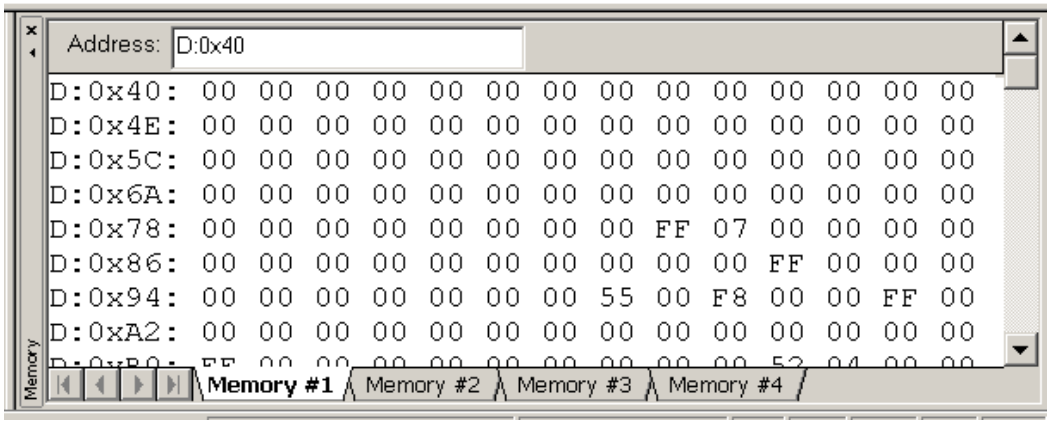

Рис. 7.

4.1.2 Примеры использования команд арифметических операций

Пример 2. Сложение. Сложить два двоичных многобайтных числа. Слагаемые располагаются в резидентной памяти данных, начиная с младшего байта. Начальные адреса слагаемых заданы в R0 и R1, формат слагаемых в байтах - в R2:

```
          CLR  C          ;сброс переноса
LOOP:     MOV  A, @R0     ;загрузка в A текущего байта первого
                          ;слагаемого
          ADDC A, @R1     ;сложение байтов с учетом переноса
          MOV  @R0, A     ;размещение байта результата
          INC  R0         ;продвижение указателей
          INC  R1
          DJNZ R2, LOOP   ;цикл, если не все байты просуммированы
```

При сложении чисел без знака на переполнение укажет флаг C, а в случае сложения чисел со знаком - флаг OV.

Дополните программу сложения командами, обеспечивающими её тестирование, составьте контрольный пример и выполните отладку в Keil uVision. Определите время вычисления в зависимости от формата исходных чисел.

Пример 3. Умножение. Команда MUL вычисляет произведение двух целых безнзковых чисел, хранящихся в регистрах A и B. Младшая часть произведения размещается в A, а старшая - в регистре-расширителе B. Если содержимое B оказывается равным нулю, то флаг OV сбрасывается, иначе - устанавливается. Флаг переноса всегда сбрасывается. Например, если аккумулятор содержал число 200 (0C8H), а расширитель 160 (0A0H), то в результате выполнения команды MUL AB получится произведение 32000 (7D00H). Аккумулятор будет содержать нуль, а расширитель – 7DH, флаг OV будет установлен, а флаг C - сброшен.

Пусть требуется умножить целое двоичное число произвольного формата на константу 73. Исходное число размешается в резидентной памяти данных, адрес младшего байта находится в регистре R0. Формат числа в байтах хранится в R1:

```
          MOV  A,#0       ;сброс аккумулятора
LOOP:     ADD  A,@R0      ;загрузка множимого
          MOV  B,#73      ;загрузка множителя
          MUL  AB         ;умножение
          MOV  @R0,A      ;запись младшего байта частичного
                          ;произведения
          INC  R0         ;приращение адреса
          MOV  A,B        ;пересылка старшего байта частичного
                          ;произведения в аккумулятор
          XCH  A,@R0      ;предварительное формирование
                          ;очередного байта произведения
          DJNZ R1,LOOP    ;цикл, если не все байты исходного
                          ;числа умножены на константу
```

Полученное произведение размещается на месте исходного числа и занимает в памяти на один байт больше.

Разберитесь в алгоритме умножения. Дополните программу командами, обеспечивающими её тестирование, составьте контрольный пример и выполните отладку в Keil uVision. Определите время вычисления в зависимости от формата исходного числа.

Пример 4. Деление. Команда DIV производит деление содержимого аккумулятора на содержимое регистра-расширителя. После деления аккумулятор содержит целую часть частного, а расширитель - остаток. Флаги C и OV сбрасываются. При делении на нуль устанавливается флаг переполнения, а частное остается неопределенным. Команда деления может быть использована для быстрого преобразования двоичных чисел в десятичные двоично-кодированные (BCD-числа).

В качестве примера рассмотрим программу, которая переводит двоичное число, содержащееся в аккумуляторе, в BCD-код. При таком преобразовании может получиться трёхразрядное BCD-число. Старшая цифра (число сотен) будет размещена в регистре R0, а две младшие в аккумуляторе:

```
MOV  B,#100    ;загрузка 100 для вычисления количества сотен
DIV  AB        ;аккумулятор содержит число сотен (старшую цифру)
MOV  R0,A      ;пересылка в R0 старшей цифры
XCH  A,B       ;пересылка остатка исходного числа в аккумулятор
MOV  B,#10     ;загрузка 10 для вычисления количества десятков
DIV  AB        ;A содержит число десятков, B - число единиц
SWAP A         ;размещение числа десятков в старшей тетраде A
ADD  A,B       ;подсуммирование остатка (числа единиц),
               ;теперь аккумулятор содержит две младшие цифры
```

Разберитесь в алгоритме перевода. Дополните программу командами, обеспечивающими её тестирование, составьте контрольный пример и выполните отладку в Keil uVision. Определите время вычисления.

4.1.3 Пример использования команд логических операций

Пример 5. Логические операции. Составьте программу поразрядной обработки данных. Программа должна установить нулевой разряд числа в регистре R5 в 1, сбросить четвёртый разряд в 0 и инвертировать шестой разряд. Используйте команды логических операций числа с масками. Составьте контрольный пример для отладки программы.

Выполните отладку программы. Убедитесь в её работоспособности на контрольных примерах. Определите время вычисления.

4.1.4 Пример использования команд передачи управления и работы со стеком

Пример 6. Операции со стеком. Перед загрузкой в стек содержимое регистра-указателя стека SP инкрементируется, а после извлечения из стека декрементируется.

По сигналу системного сброса в SP заносится начальное значение 07H. Для переопределения SP можно воспользоваться командой MOV SP, #d.

Таким образом, стек может располагаться в любом месте резидентной памяти данных. Стек используется для организации обращений к подпрограммам и при обработке прерываний, может быть использован для передачи параметров подпрограммам и для временного хранения содержимого регистров специальных функций.

Подпрограмма должна сохранить в стеке содержимое тех регистров, которые она сама будет использовать, а перед возвратом в прерванную программу должна восстановить их значения.

Подпрограмма с дополнениями для её тестирования может, например, иметь следующую структуру:

```
START:    MOV  R1,#02H   ;загрузка регистров
          MOV  A,#30H
          MOV  R2,#00H
          LCALL SUB      ;переход на подпрограмму
          SJMP START

SUB:      PUSH PSW  ;сохранение в стеке PSW
          PUSH ACC  ;сохранение аккумулятора
          PUSH B    ;сохранение расширителя аккумулятора B
          ADD  A,R1 ;собственно обработка данных
          MOV  R2,A
          POP  B    ;восстановление B
          POP  ACC  ;восстановление аккумулятора
          POP  PSW  ;восстановление PSW
          RET       ;возврат
          END
```

Если предположить, что SP перед возникновением прерывания содержал значение 1FH, то размещение регистров в стеке после входа в подпрограмму обработки будет таким, как на рис. 8.

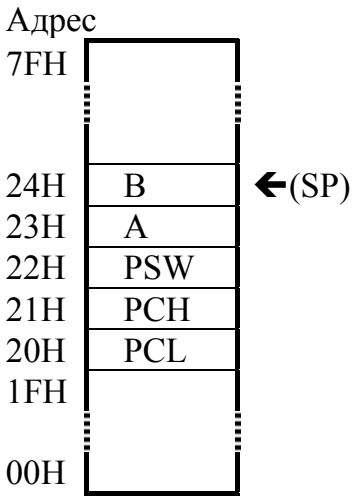

Рис. 8. Содержимое стека после выполнения команды CALL и серии команд PUSH.

Исследуйте процесс выполнения команд вызова и возврата из подпрограммы, а также команд работы со стеком. Для этого запустите программу в пошаговом режиме.

Замените команду POP и PSW на NOP и проследите, как будет выполняться программа. Объясните происшедшие изменения, сформулируйте выводы для отчёта.

4.2 Пример программы для учебно-лабораторного стенда SDK-1-1

Пример 7. Ниже приведён пример программы на языке ассемблера для учебно-лабораторного стенда SDK-1-1. Данная программа формирует эффект бегущих огней на светодиодах стенда. Число одновременно горящих светодиодов задаётся при помощи симулятора порта дискретного ввода/вывода (1, 2, 3, 4 - линии).

```
#include <ADuC812.inc>
NAME Primer_7

SV   EQU  7h   ; адрес порта светодиодов в ПЛИС

PROG_WM   SEGMENT CODE
PROG_T0   SEGMENT CODE
VAR1 SEGMENT   DATA
BITVAR    SEGMENT   BIT
STACK     SEGMENT   IDATA

; Макрос настройки таймера 0
Timer     macro V_TMOD,V_TCON
     mov TMOD,V_TMOD
     mov TCON,V_TCON
     endm

     RSEG  STACK
     DS    10H  ; 16 Bytes Stack
; Обработчик прерывания при перезагрузки МК
     CSEG AT 0h
     jmp START
; Вектор прерывания от таймера 0
     CSEG AT 0Bh
     jmp Timer0
; Вектор прерывания от таймера 0 в пользовательской таблице SDK
     CSEG AT 200Bh
     jmp Timer0

; Подпрограмма обработчика прерывания от таймера 0
     RSEG PROG_T0
Timer0: push PSW
     push ACC
     mov A,SVET
     RR  A
     mov R1,#SV
     mov R2,A
     mov SVET,A
     call Wr_Max
     SETB IRQ_T0    ; установка флага прерывания от таймера 0
     pop ACC
     pop PSW
     RETI

; Основная программа
     CSEG AT 2100h
```

```
     USING 0
START:    mov SP,#STACK-1 ; инициализация стека
; Настройка таймера 0 как 16-разрядного в режиме автоперезагруз-
ки
     Timer #1h,#10h
     SETB TR0  ; запуск таймера 0
     SETB EA        ; разрешение прерываний
     SETB ET0  ; разрешение прерывания от таймера 0

     mov SVETI,#1h  ; начальная загрузка
     mov SVET,SVETI ; начальная загрузка
     SETB IRQ_T0    ; установка флага прерывания от таймера 0
cycle:    mov A,SVET
     CJNE A,SVETI,wait
     mov A,P3
     ANL A,#00111100b
     mov SVET,A
     mov SVETI,A    ; сохранение переменной SVET
     CLR IRQ_T0     ; сброс флага прерывания от таймера 0
wait:     JB IRQ_T0,cycle
     jmp wait

; Подпрограмма работы с регистрами ПЛИС (Запись в рег. ПЛИС)
     RSEG PROG_WM
; R1 - адресс регистра ПЛИС, R2 - записываемое значение
Wr_Max:   mov R3,DPP
     mov DPP,#8h
     mov A,R2
     mov DPH,#0
     movx @R1,A
     mov DPP,R3
     RET

; Переменные в DATA
     RSEG VAR1
SVET:     DS 1
SVETi:    DS 1
; Переменные в битовой области
     RSEG BITVAR
IRQ_T0: DBIT 1

END
```

Создайте проект в Keil Software для примера 7, аналогичным образом как в лабораторной работе №2 [4]. Разберитесь с работой программы и составьте её блок-схему.

4.3 Индивидуальное задание

Вариант индивидуального задания указывается преподавателем.

5 Содержание отчёта

1. Цель работы.
2. Пример машинного кода для инструкции микроконтроллера Intel 8051 указанный преподавателем.
3. Результаты выполнения заданий.

3.1. Листинг примеров с комментариями и результатами экспериментов.

Листинг программы для SDK-1-1 с комментариями. Блок-схема программы.

Листинг программы индивидуального задания с комментариями. Блок-схема программы.

4. Выводы по проделанной лабораторной работе.

6 Контрольные вопросы

1. По каким функциональным группам можно классифицировать команды микроконтроллера?
2. Какой формат может иметь команда?
3. Как длительность машинного цикла микроконтроллера соотносится с его тактовой частотой?
4. Как определить время выполнения команды?
5. С какими типами данных может оперировать микроконтроллер?
6. Для чего используются четырёхбитные операнды?
7. Какие команды работают с четырёхбитными операндами?
8. Для чего используются двухбайтные операнды?
9. Как косвенно адресуются байты памяти?
10. Укажите назначение флагов слова состояния программы PSW.
11. Сформулируйте условия установки флага OV.
12. Каково назначение регистров указателей?
13. Может ли порт одновременно являться источником операнда и приемником результата операции?
14. Какие способы адресации используются в микроконтроллере?
15. Можно ли адресовать порты и регистры специальных функций косвенно?
16. Приведите примеры команд передачи данных с различными способами адресации.
17. Расшифруйте команду MOVC A, @A+DPTR.
18. Приведите примеры команд доступа к 256 байтам резидентной памяти данных, к внешней памяти данных.
19. Приведите примеры логических и арифметических команд.
20. Как выполнить вычитание многобайтных операндов?
21. Перечислите команды операций с битами.
22. Как инвертировать отдельные биты портов?
23. Можно ли адресовать биты косвенно?
24. Какие переходы возможны в командах управления?
25. Для чего используются косвенные переходы в программах?
26. Поясните отличия длинного, абсолютного и относительного переходов в программах.
27. Как организовать процедуру ожидания с помощью одной команды?
28. Какие команды используются при организации подпрограмм?
29. Какие команды модифицируют флаги результата?
30. Укажите, какие из регистров специальных функций допускают битовую адресацию.
31. Какие флаги используются командами условных переходов?
32. Чем отличаются команды RET и RETI?
33. Перечислите и охарактеризуйте этапы разработки прикладной программы.
34. Какова структура строки программы, написанной на языке ассемблера, какие поля строки являются обязательными?
35. Укажите правила выбора имени метки.
36. Каким образом выделяется комментарий?
37. Перечислите основные директивы языка ассемблера.
38. Какие директивы используются для указания начала и конца программного модуля?
39. Перечислите директивы, используемые для резервирования памяти.
40. Какие директивы используются для определения программных сегментов?
41. Укажите функции следующих директив: ORG, EQU, SET, BIT. В чём состоит отличие директив EQU и SET?

42. Укажите назначение директив DATA, XDATA.
43. Укажите назначение директив DB, DS, DW, END.
44. Укажите назначение директив RSEG, SEGMENT.
45. Каким образом определяется макрокоманда?
46. Укажите достоинства и недостатки подпрограмм по сравнению с макрокомандами. В каких случаях целесообразно использовать подпрограммы, а в каких - макрокоманды?
47. Какие команды можно использовать для создания циклической программы?
48. Какими обязательными свойствами должна обладать подпрограмма?
49. Каким образом используется стек при выполнении подпрограмм?
50. От чего зависит глубина вложенности подпрограмм?
51. Для чего может быть использован приём переключения регистровых банков?
52. Поясните механизм передачи параметров подпрограммы через память.
53. Поясните механизм передачи параметров подпрограммы через регистры общего назначения.
54. Поясните механизм передачи параметров подпрограммы через регистр признаков.
55. Поясните механизм передачи параметров подпрограммы через стек.
56. Укажите основные этапы работы с ассемблерной программой.
57. Как рассчитать время выполнения ассемблерной программы?
58. Укажите основные приемы, используемые при отладке программы.

Перечень источников

1. Горюнов А.Г. Ливенцов С.Н. Архитектура микроконтроллера i8051 L:\Study\МПТ\Методички\MCS51.pdf
2. Сташин В.В., Урусов А.В., Мологонцева О.Ф. Проектирование цифровых устройств на однокристальных микроконтроллерах. М.: Энергоатомиздат, 1990. 224 с.
3. Руководство пользователя компилятора ассемблера Ax51 L:\Keil\C51\HLP\A51.pdf
4. Лабораторная работа №2. Разработка прикладного программного обеспечения для микропроцессорных систем на основе микроконтроллера. (Быстрый старт) L:\Study\МПТ\Лаб_раб\LAB2\LAB2.pdf
5. Однокристальные микроЭВМ/ А.В.Боборыкин, Г.П.Липовецкий, Г.В.Литвин¬ский и др. М.: МИКАП, 1994. 400 с.
6. Микропроцессоры. В 3-х кн. Кн. 1. Архитектура и проектирование микро-ЭВМ. Организация вычислительных процессов: Учебник для втузов/ П.В.Нестеров, В.Ф.Шаньгин, В.Л.Горбунов и др.; Под ред. Л.Н.Преснухина. М.: Высшая шко-ла, 1986. 495 с.

Приложения

Приложение 1
Директивы ассемблера в алфавитном порядке

Ниже дано описание основных, часто используемых директив, доступных в ассемблере A51 пакета ProView. Директивы могут включать ряд факультативных полей или аргументов:

Имя	**Описание**
Address	Допустимый код или адрес данных.
Argument	Значение или выражение, заменяющее формальное имя пара-метра.
Bit-address	Допустимый адрес бита.
Expression	Допустимое выражение.
Label	Допустимый код программы или метка данных.
Number	Числовая константа, составленная только из цифр.
Parameter	Символическое имя формального параметра.
Register	Имя регистра: A, R0, R1, R2, R3, R4, R5, R6, или R7.
String	Строка символов и цифр.
Symbol	Допустимое символическое имя.

BIT

Описание Директива BIT назначает символическое имя адресу бита. Формат директивы:

symbol BIT *bit-address*

где **symbol** - символическое имя,
bit-address - адрес бита в резидентной памяти данных.

Символические имена, определенные директивой BIT, не могут быть изменены или переопределены.

Пример

```
RSEG      DATA_SEG       ;выбор сегмента
CTRL:          DS   1    ;однобайтовая переменная (CTRL)
ALARM          BIT  CTRL.0    ;бит в перемещаемом байте
SHUT      BIT  ALARM+1   ;следующий бит
ENABLE_FLAG    BIT  60H       ;абсолютный бит
DONE_FLAG BIT  24H.2     ;абсолютный бит
```

DATA

Описание Директива DATA назначает символическое имя адресу резидентной памяти данных. Формат директивы:

symbol DATA *address*

где **symbol** - символическое имя, которое может использоваться во всей программе,
address - адрес резидентной памяти данных, должен находится в диапазоне от 0 до 255.

Символические имена, определенные этой директивой, не могут быть изменены или переопределены.

Пример

```
SERBUF    DATA SBUF
RESULT    DATA 40H
RESULT2   DATA RESULT + 2
PORT1     DATA 90H
```

DB

Описание Директива DB заносит в память программ 8-разрядное значение байта. Директива имеет следующий формат:

label: **DB** *expression , expression ...*

где **label:** - метка, адрес инициализированной памяти,

expression - значение байта, которое может быть символом, символьной строкой или выражением.

Директива DB может быть определена только внутри сегмента кода. Если директива используется в другом сегменте, ассемблер A51 генерирует сообщение об ошибке.

Пример

```
REQUEST:       DB   'PRESS ANY KEY TO CONTINUE', 0
TABLE:         DB   0, 1, 8, 'A', '0', Low(TABLE), ';'
ZERO:          DB   0, ''''
CASE_TAB: DB   Low(REQUEST), Low(TABLE), Low(ZERO)
```

DS

Описание Директива DS резервирует определенное число байтов в резидентной памяти данных, внешней памяти данных или адресном пространстве кода программы. Директива имеет следующий формат:

label: **DS** *expression*

где **label:** - метка, присвоенная адресу зарезервированной памяти,

expression - количество зарезервированных байтов, не может содержать форвардные ссылки, перемещаемые символы или внешние символы.

Директива резервирует пространство в текущем сегменте по текущему адресу. Затем текущий адрес увеличивается на значение выражения. Сумма счётчика адреса и значения выражения не может превышать границу текущего адресного пространства.

Примечание. A51 - ассемблер с двумя проходами по исходному тексту программы. В первом проходе обрабатываются символы и определяется длина каждой команды. Во втором проходе обрабатываются форвардные ссылки и генерируется объектный код. По этим причинам выражение, используемое в директиве, не может содержать форвардные ссылки.

Пример

```
GAP: DS   (($ + 16) AND 0FFF0H) - $
     DS   20
TIME:     DS   8
```

DW

Описание Директива DW инициализирует память программ 16-разрядными значениями слова. Директива имеет следующий формат:

label: **DW** *expression , expression ...*

где **label:** - метка, присвоенная адресу зарезервированной памяти,

expression - выражения - данные, которые могут содержать символ, символьную строку или выражение.

Директива может быть определена только внутри сегмента кода. Если директива используется в другом сегменте, ассемблер A51 генерирует сообщение об ошибке.

Пример

```
TABLE:         DW   TABLE, TABLE + 10, ZERO
ZERO:          DW   0
CASE_TAB: DW   CASE0, CASE1, CASE2, CASE3, CASE4
               DW   $
```

END

Описание Директива END сообщает о конце ассемблерного модуля. Любой текст в ассемблерном файле, который появляется после этой директивы, игнорируется. Директива требуется в каждом исходном ассемблерном файле. Если директива отсутствует, ассемблер генерирует сообщение о фатальной ошибке.

Пример

```
END
```

EQU

Описание Директива EQU назначает числовому значению или регистру символическое имя. Формат директивы:

symbol **EQU** *expression*
symbol **EQU** *register*

где **symbol** - символическое имя, которое заменяется на выражение или регистр во всей ассемблерной программе,
expression - числовое выражение, не содержащее форвардных ссылок,
register - одно из следующих имен регистра: A, R0-R7.

Символические имена, определенные директивой, могут использоваться в операндах, выражениях или адресах. Символы, которые определены как имя регистра, могут использоваться во всех командах, работающих с регистрами. Символические имена, определенные директивой, не могут быть изменены или переопределены.

Пример

```
LIMIT   EQU  1200
VALUE   EQU  LIMIT - 200 + 'A'
SERIAL  EQU  SBUF
ACCU    EQU  A
COUNT   EQU  R5
```

ORG

Описание Директива ORG определяет адрес начала последующего кода программы или данных. Формат директивы:

ORG *expression*

где **expression** - должно быть абсолютным или простым перемещаемым выражением и не может иметь форвардных ссылок; символы, используемые в выражении, могут ссылаться только на текущий или абсолютный сегмент.

Когда ассемблер сталкивается с этой директивой, он вычисляет значение выражения и изменяет счётчик адресов (внутренняя переменная ассемблера) для текущего сегмента. Если директива находится в абсолютном сегменте, счётчику назначается значение истинного адреса. Если директива находится в перемещаемом сегменте, счётчику назначается смещение, определяемое выражением.

Директива изменяет счётчик адресов, но не производит новый сегмент. Неиспользованный диапазон адресов можно включить в текущий сегмент. Обратите внимание, что при использовании абсолютных сегментов счётчик не может ссылаться на адрес до начала смещения.

Примечание. A51 - ассемблер с двумя проходами по исходному тексту программы. В первом проходе обрабатываются символы и определяется длина каждой команды. Во втором проходе обрабатываются форвардные ссылки и генерируется объектный код. По этим причинам выражение, используемое в директиве, не может содержать форвардные ссылки.

Пример

```
ORG     100H
ORG     RESTART
ORG     EXTI1
ORG     ($ + 16) AND 0FFF0H
```

RSEG

Описание Директива RSEG выбирает перемещаемый сегмент, который был предварительно объявлен директивой SEGMENT. Формат директивы:

RSEG *segment*

где **segment** - имя сегмента, предварительно определенное директивой SEGMENT.

Для получения дополнительной информации относительно использования сегментов обратитесь к разделу справочной системы Keil Software “A51” [3].

Пример

```
MYPROG  SEGMENT  CODE       ;объявление сегмента
    RSEG        MYPROG     ;выбор сегмента
                MOV        A, #0
MOV     P0, A
```

SEGMENT

Описание Директива SEGMENT используется для того, чтобы объявить перемещаемый сегмент. Тип сегмента может быть определен в объявлении сегмента. Директива имеет следующий формат:

segment **SEGMENT** *segtype reloctype*

где **segment** - символическое имя, назначенное сегменту,

segtype - тип сегмента, определяющий адресное пространство сегмента;

reloctype - тип перемещения для сегмента, который определяет параметры перемещения для компоновщика L51; обратитесь к таблице ниже для получения дополнительной информации.

Имя каждого сегмента внутри модуля должно быть уникально. Однако L51 объединяет сегменты одинакового типа. Это также применимо к сегментам, объявленным в других исходных модулях.

Переменная segtype определяет адресное пространство для сегмента и может быть любой из следующих:

Segtype	**Описание**
BIT	Битовое адресное пространство (пространство резидентной памяти данных с адреса 20H по 2FH).
CODE	Пространство кода программы.
DATA	Пространство резидентной памяти данных (адреса с 0 по 127).
IDATA	Косвенно адресуемое пространство резидентной памяти данных (с 0 по 127 или с 0 по 255).
XDATA	Пространство внешней памяти данных.

Факультативный параметр reloctype - тип перемещения определяет действия компоновщика BL51. Следующая таблица содержит список допустимых типов настройки:

Reloctype	**Описание**
BITADDRESSABLE	Определяет сегмент, который будет перемещен L51 внутрь битовой адресуемой области памяти (адреса с 20H по 2FH в резидентной памяти дан-

ных). Разрешён только для сегментов DATA, которые по длине не превышают 16 байтов.

INBLOCK	Определяет сегмент, который должен содержаться в 2048-байтовом блоке. Этот тип допустим только для сегментов CODE.
INPAGE	Определяет сегмент, который должен содержаться в 256-байтовом блоке. Этот тип настройки допустим только для сегментов CODE и XDATA.
OVERLAYABLE	Определяет, что сегмент может использовать память совместно с другими сегментами этого же типа. При использовании этого типа настройки имя сегмента должно быть объявлено согласно правилам C51 или PL/M-51.
PAGE	Определяет сегмент, чей начальный адрес должен быть в 256-байтовой границе. Размещение сегмента выполняется компоновщиком BL51. Этот тип настройки допустим только для сегментов CODE и XDATA.
UNIT	Этот тип размещения задан по умолчанию как стандартный тип. Он определяет сегмент, который начинается в границе модуля. Модуль - байт для сегментов CODE, DATA, IDATA и XDATA и бит - для сегмента BIT.

Примечание. Сегментные символы, используемые в выражениях, представляют собой базовый адрес объединенного сегмента, вычисляемый компоновщиком BL51.

Для получения дополнительной информации относительно использования сегментов обратитесь к разделу справочной системы Keil Software “A51” [3].

Пример

```
STACK   SEGMENT   IDATA
RSEG        STACK           ;выбор сегмента
            DS      10H             ;резервирование 16 байтов
            MOV     SP, #STACK – 1  ;инициализация SP
```

SET

Описание Директива SET назначает символическое имя числовому значению или регистру. Формат директивы:

symbol **SET** *expression*
symbol **SET** *register*

где **symbol** - символическое имя, которое заменяется на выражение или регистр во всей ассемблерной программе,

expression - числовое выражение, не содержащее форвардных ссылок,

register - одно из следующих имен регистра: A, R0-R7.

Символические имена, определенные директивой, могут использоваться в операндах, выражениях или адресах. Символы, которые определены как имя регистра, могут использоваться во всех командах, работающих с регистрами. Имена, определенные директивой, могут быть изменены последующими директивами SET.

Пример

```
VALUE   SET  100
```

```
VALUE    SET  VALUE / 2
COUNTER  SET  R1
TEMP     SET  COUNTER
TEMP     SET  VALUE * VALUE
```

XDATA

Описание Директива XDATA назначает символическое имя адресу внешней па-мяти данных. Формат директивы:

symbol **XDATA** *address*

где **symbol** - символическое имя, которое может использоваться во всей программе,

address - адрес внешней памяти данных, должен находиться в диапазоне от 0 до 65535.

Символические имена, определенные этой директивой, не могут быть изменены или переопределены.

Пример

```
RSEG     XSEG1
ORG      100H
DTIM:    DS        6    ;резервирует 6 байтов для DTIM
TIME     XDATA     DTIM + 0
DATE     XDATA     DTIM + 3
```