Замечание
Здесь и далее, если не указано иное, под словом ассемблер подразумевается ассемблер для AVR – архитектуры, используемой в контроллерах ATMEL.

0. Вводные замечания
Язык программирования "ассемблер" это запись непосредственно инструкций для процессора.

В принципе, инструкции процессора можно было бы записывать прямо в шестнадцатеричных кодах, например **$2001** – это инструкция, совершающая логическую операцию AND, причем в коде этой инструкции уже зашифровано, что операция должна быть осуществлена над первыми двумя регистрами (внутренними ячейками памяти) – а результат будет помещен в первую из этих двух ячеек.

По общепринятому мнению программировать прямо в кодах неудобно – не только из-за необходимости запоминать как шифруется каждая команда, но и из-за того, что приходится заниматься кучей нудных расчетов – считать на сколько шагов по программе должна перескочить инструкция перехода, например и т.п.

В связи с этим умные люди придумали программу, называемую также "ассемблер" (лучше называть ее "компилятор ассемблера" или хотя бы "ассемблировщик", хотя это не очень правильно) которая читает из входного файла названия инструкций и их операндов, по своим таблицам производит нужные расчеты и шифрования и записывает в выходной файл готовый код требуемой инструкции.

Очевидно, при таком подходе программист все равно будет **точно уверен** в том, сколько и каких инструкций он записал, но просто записывать их сможет в более удобном виде. Например, вышеупомянутая инструкция запишется так: **and r0,r1**, но на выходе ассемблера из нее получится все тот же код **$2001**. Впоследствии цепочку получившихся кодов мы можем записать в программную память процессора и он, после включения, радостно начнет их выполнять.

1. Начнем с примера
Переведем программу с Паскаля на ассемблер. Эта программа очень проста, она ничего не спрашивает у пользователя и ничего не сообщает ему. Если ее запустить, мы даже не заметим, что что-то произошло. Тем не менее она очень полезна для изучения.

текст на Паскале:	*текст на ассемблере:*	*коды инструкций:*

`{это комментарий}`
`var`
`  a,b,c:byte;`
`begin`
`  a:=1;`
`  b:=2;`
`  c:=a+b;`
`end.`
	`;это комментарий`
	`ldi r16,1`
	`ldi r17,2`
	`add r16,r17`
	`mov r18,r16`
		
		`$E001`
		`$E012`
		`$0F01`
		`$2F20`

Что именно здесь записано – будет объяснено ниже. Сейчас же попробуем преобразовать нашу программу в коды процессора.

Чтобы скомпилировать программу в Паскале мы обычно запускаем интегрированную среду разработки, например, Турбо-Паскаля, открываем в нем файл с исходным текстом, нажимаем **Compile** в меню и восторгаемся результатам. Если мы не пользуемся интегрированной средой обработки, мы просто запускаем компилятор Паскаля, указав ему в качестве параметра файл с программой:
`c:\tp70\tpc.exe prg1.pas`

из командной строки. В результате, если повезет, получаются несколько файлов, среди них и **prg1.exe – исполнимый модуль готовой программы**.

С ассемблером дело обстоит примерно так же. Существуют, вообще говоря, и интегрированные среды разработки в которых можно "щелкнуть" где-нибудь **Compile** и все получится. Однако чтобы разобраться с этими средами хорошо бы знать собственно ассемблер. Поэтому пока обойдемся без них.

Воспользуемся компилятором **avrasm32.exe**, который я высылал. Для этого:
- наберем нашу программу в каком-нибудь простом текстовом редакторе;
- сохраним ее в текстовый файл с именем **prg1.asm**;
- скомпилируем ее компилятором введя указанную ниже команду.

```
c:\atmel\avrasm32.exe -fG prg1.asm
```

Как видим, кроме имени файла с исходным текстом нам пришлось указать ключ "-fG" – не следует беспокоиться по этому поводу, для компилятора Паскаля количество возможных ключей гораздо больше, чем для компилятора ассемблера, нам просто повезло, что не пришлось их задавать. Этот конкретный ключ сообщает, что выходной файл мы хотим получить в самом примитивном и простом для нашего понимания формате (generic). Заметим, что программатор-прошивальщик **avreal.exe** работает с файлами, записанными в формате "Intel", поэтому для создания таких файлов нужно будет указать ключ "-fI" вместо только что упомянутого.

Итак, компилятор скомпилировал нашу программу и, вероятно, сообщил, после слов о копирайтах и о том, какой он хороший, что-то вроде:

```
Creating   'prg1.hex'

Assembling 'prg1.asm'

Program memory usage:
Code            :    4 words
Constants (dw/db):    0 words
Unused          :    0 words
Total           :    4 words

Assembly complete with no errors.
```

Здесь он пишет что создал файл **prg1.hex**, который является собственно выходным файлом и стал ассемблировать входной файл, записывая в выходной результаты. По окончании он сообщает, что всего скомпилировано **4 слова** кода и **компиляция завершена без ошибок**.

Слово – это историческое название для ячейки памяти, не равной байту. Слово может содержать несколько байт 2 или, скажем, 4 (и даже нецелое число, например, у разных контроллеров PIC слова бывают по 12 и 14 бит) – но вполне возможно, что собственно на байты оно не разбивается, как байт, например (в Паскале), не разбивается на кусочки по 4 бита – вы можете работать только с байтом целиком.

Заметим, что во входном файле у нас было 4 инструкции и они соответственно преобразовались в 4 слова. Каждая в одно. Например в микроконтроллере ATtiny15 флеш-память для программ имеет объем всего 512 слов. Если вы чувствуете, что написали программу, состоящую более чем из 512 инструкций, вы можете быть уверены, что она не влезет.

Теперь рассмотрим выходной файл **prg1.hex**. Он тоже текстовый, так что его можно без боязни открыть, например, в "блокноте". Сделав это мы увидим:

```
000000:e001
000001:e012
000002:0f01
000003:2f20
```
Догадаться несложно. Слева записан номер ячейки памяти программ, куда должна быть записана инструкция (то есть, ее **адрес**), а дальше через двоеточие шестнадцатеричное число, представляющее собственно код инструкции.

Теперь **перекомпилируйте** программу так, чтобы получить hex-файл в формате **Intel**. В дальнейшем компилируйте только в этот формат.

2. Разберемся с программой

В третьем параграфе мы попытаемся запустить нашу программу в симуляторе, однако прежде чем делать это, необходимо осознать смысл записанных нами инструкций.

В первую обратим внимание, что программа на Паскале состоит из двух важных частей:
- раздел распределения памяти **данных**;
- описание инструкций **кода**.

Первая часть начисто отсутствует в нашей ассемблерной программе. Программист на Паскале не заботится о том, где его переменные будут размещены в памяти компьютера – он только придумывает для них имена и определяет ее тип. Все остальное отдается на решение компилятора и операционной системы. В ассемблере все более "деревянно". Программист заранее решает, в каких ячейках памяти он собирается хранить свои данные. В нашем случае мы решили, что для хранения переменных мы используем регистры процессора:
- A хранится в 16-м регистре;
- B хранится в 17-м регистре;
- C хранится в 18-м регистре.

Регистрами называются ячейки оперативной памяти встроенные прямо в вычислительное устройство, их обычно немного, зато работа с ними происходит быстрее всего. Даже если у микроконтроллера есть дополнительная оперативная память, обычно его процессор не может выполнять с ней всех действий, какие он может выполнять с регистрами. Поэтому для обработки данных, находящихся в оперативной памяти их сначала нужно загрузить в регистры, потом обработать и выгрузить обратно.

Вторая часть собственно и записана всеми 4-мя инструкциями программы. Как мы видим, запись инструкции состоит из имени ("мнемоники") данной инструкции и указания того, с чем эта инструкция должна работать – то есть "операндов". Некоторые инструкции требуют в качестве операндов названий регистров, некоторые – просто чисел, некоторые требуют и того и другого, а некоторые не используют операндов вообще. Я настоятельно рекомендую открыть файл справки по ассемблеру **avrasm.chm** и найти в нем описания трех использованных нами инструкций. Вкратце их смысл изложен ниже.

Первая инструкция **LDI** (сокращение от Load Immediate – загрузить непосредственное значение) загружает в указанный регистр указанное число – мы указали регистр №16 и число 1. Как видим, обычно в инструкциях, обрабатывающих два операнда результат записывается в первый из операндов – это выглядит немножко "задом наперед" но в целом к этому быстро привыкаешь. Заметим также что:
- регистр обозначается буквой R (большой или маленькой) с номером регистра;
- десятичные числа обозначаются просто цифровой записью, начинающейся с цифры от 1 до 9.

Кроме этого можно записывать числа в двоичной, восьмеричной и шестнадцатеричной системе счисления. В ассемблере это бывает полезно, поэтому в отличие от большинства языков он позволяет такое разнообразие:
- двоичные состоят из цифр 0 и 1, в начале числа пишется префикс "0b";
- восьмеричные состоят из цифр от 0 до 7, в начале числа пишется префикс "0" (просто 0);
- шестнадцатеричные состоят из цифр от 0 до 9 и букв от A до F (можно маленькими), в начале числа пишется префикс "0x" или ставится знак "$".
Таким образом, например, 25=0b11001=031=0x19=$19.

Вторая инструкция, как несложно догадаться, осуществляет подобную же загрузку, но с регистром №17 и числом 2.

Что делает третья инструкция, можно догадаться из ее названия. **ADD** – по-английски "добавлять". Эта инструкция имеет в качестве операндов названия двух регистров и добавляет к числу, находящемуся в первом, число, находящееся во втором. Результат, как обычно, записывается обратно в первый регистр (в принципе, это соответствует смыслу слова "добавить").

Четвертая инструкция **MOV** (от английского слова **Move** – "задвинуть", в данном случае, – считается, что это не очень удачное название) также принимает два регистра в качестве операндов и "задвигает" в первый значение из второго. То есть она, как и **LDI** осуществляет загрузку в регистр, только значение берется не непосредственно "зашитое" в код инструкции, а то, которое находится в другом регистре. Как видим, в Паскале обе эти инструкции обозначаются знаком присваивания.

Обратим на то, как в ассемблере принято записывать комментарии. Комментарием считается все, что указано после знака "точка-с-запятой" и до конца строки.

Вообще инструкции могут выполнять еще и некоторые побочные действия, однако об этом мы пока говорить не будем, а перейдем к тестированию нашей программы в симуляторе.

3. Симулятор и отладка программы
Первым делом нам потребуется симулятор. Неплохой симулятор встроен в интегрированную среду разработки **AVR Studio** – ее можно скачать на сайте www.atmel.ru, или www.atmel.com например (http://www.atmel.ru/Software/files/astudio3.exe) – ссылку я даю на старенькую третью версию, хотя существуют уже и четвертые разные. Это не очень принципиально, просто старая версия достаточна, а по размеру значительно меньше.

Установив и запустив этот продукт (в дальнейшем будем называть его симулятором) откроем в нем созданный нами файл с кодами инструкций для микроконтроллера (в формате Intel). Сделайте это пользуясь меню или еще как-нибудь, если вы не можете этого сделать, то прекратите сейчас же читать этот текст и вообще больше не включайте никогда компьютер. Объяснений типа "Щелкните левой кнопкой мыши по слову **File** в строке меню в верхней части окна, потом выберите **Open** и в открывшемся диалоге найдите нужный каталог и выберите нужный файл" я давать разумеется **не буду**. В крайнем случае я буду записывать через слеш путь по системе меню и подменю (например, File/Open).

Если вы открываете файл в первый раз, то симулятор выдаст запрос (диалоговое окно с заголовком **Simulator Options**) по поводу того, какую микросхему мы собрались "симулировать". Можно выбрать из списка **Device** нужную или даже придумать что-то свое. Я выбрал **ATtiny15**, при этом симулятор сразу прописал сколько кода, данных и так далее есть в этой микросхеме. (кста-

ти, симулятор написал что у этой микросхемы 32 байта оперативной памяти еще есть, кроме регистров – по-моему это ошибка).

После нажатия кнопки OK симулятор немного подумает, и откроет файл, однако уже не в текстовом виде. Симулятор расшифровал коды инструкций и записал эти инструкции снова в "человеческом" виде, то есть на ассемблере. Это называется "дизассемблированием". Однако для отладки нам нужно открыть еще одно полезное окно, показывающее текущее состояние регистров процессора, которое можно достать из меню View/Registers. В этом окне перечислены все 32 регистра и указано, какие значения в них сейчас находятся.

Разместите оба открытых окна как-нибудь рядышком, чтоб получить примерно такой вид:

```
+00000000: E001      LDI     R16,0x01  | R0  = 0x00    R16 = 0x00
+00000001: E012      LDI     R17,0x02  | R1  = 0x00    R17 = 0x00
+00000002: 0F01      ADD     R16,R17   | ......................
+00000003: 2F20      MOV     R18,R16   | R15 = 0x00    R31 = 0x00
```

Теперь мы готовы наслаждаться видом выполняющейся программы. Обратите внимание, что в окне с инструкциями есть стрелка, показывающая, какая инструкция сейчас должна выполняться. Команда меню Debug/TraceInto (она привязана к клавише F11) позволяет выполнять программу по шагам. Сделайте это четыре раза и после выполнения каждой инструкции смотрите во втором окне, как изменяется содержимое регистров №№ 16, 17 и 18.

После того, как вы дойдете до конца (точнее когда счетчик инструкций станет равен 4 при том, что инструкции №4 в программе нет – они начинаются с 0) симулятор не даст больше трассировать программу. Чтобы повторить все сначала воспользуйтесь Debug/Reset. Вообще в этом меню есть много полезных команд.

Откройте окно состояния процессора (View/Processor). В нем можно видеть кое-какую полезную информацию о процессоре, которую мы изучим в дальнейшем. Пока же обратите внимание на:
- счетчик адресов инструкций (Program Counter) – это служебный регистр процессора, который хранит номер следующей выполняемой инструкции;
- счетчик циклов (Cycle Counter) – импульсов тактовой частоты процессора – обычно одна команда выполняется за один цикл (такт);
- указатель времени выполнения и секундомер – при тактовой частоте 1МГц один такт выполняется за 1 микросекунду (миллионная доля) и таково же время выполнения одной инструкции.

Попробуйте пошаговую трассировку, наблюдая за окном состояния процессора. Когда до вас дойдет смысл происходящих изменений, запустите Debug/AutoStep. Чтобы выйти из этого режима, используйте Debug/Break. Скорость выполнения в режиме AutoStep настраивается в DebugOptions в том же меню.

4. Ошибки в программе

Ошибки в программах могут быть двух типов: **синтаксические** и **алгоритмические**. В первом случае в программе просто написано что-то такое, чего компилятор не может разобрать и, соответственно, не знает, как превратить это в коды инструкций. Во втором случае программа компилируется нормально, но делает что-то не то, чего хотел программист, потому что он или сам не знал, чего хотел, или же не смог логически правильно выразить это на языке программирования. Ошибки второго типа мы не будем даже обсуждать – это абсолютно бесполезно, хотя это самые тяжелые и неприятные ошибки. О некоторых ошибках первого типа мы постараемся получить представление сейчас.

Запишем программу такого вида:

```
ldi r16,1
ldi r15,5    ; этого нельзя делать с регистрами с 0-го по 15-й
ldi r17,2
ldi r19,1000 ;такое число не влезет в однобайтовый регистр
add r16,r17
add r16,4    ;нельзя прибавить непосредственное значение
mov r18,r16
blaha muha   ;это вообще не инструкция.
```

Сохраним ее в файл (например, prg2.asm) и попробуем скомпилировать. в результате мы увидим следующие замечания от компилятора:

```
Creating   'prg2.hex'

Assembling 'prg2.asm'
prg2.asm(8) : error : Unknown instruction opcode
prg2.asm(2) : error : Illegal argument type or count
prg2.asm(6) : error : Illegal argument type or count

Assembly complete with 3 errors

Deleting   'prg2.hex'
```

То есть компилятор нашел ошибки во 2-й и 6-й строке, связанные с использованием неправильных аргументов (операндов) для правильных инструкций, а также в 8-й строке неправильную инструкцию вообще. После этого он заявляет "ассемблирование закончено с 3 ошибками" и пишет о том, что выходной файл prg2.hex был удален.

Относительно ошибки во 2-й строке нужны пояснения. Команда LDI позволяет загружать числа только в регистры с номерами с 16-го по 31-й. Чтобы загрузить непосредственное число в один из младших 15 регистров, необходимо загрузить его сначала в какой-нибудь из старших, а оттуда командой MOV скопировать уже куда надо.

Ошибка в 6-й строке связана с тем, что прибавлять к регистру можно только значение из регистра. Непосредственное значение прибавлять нельзя (в некоторых процессорах можно, там для этого есть специальная инструкция ADI).

Закомментируем эти три строки с ошибками, поставив в начале "точку-с-запятой", чтобы компилятор их не видел, и повторим компиляцию. Она завершится без ошибок, хотя мы-то знаем, что по крайней мере еще одна ошибка есть в четвертой строке. Откроем файл prg2.hex в отладчике, чтобы посмотреть, во что превратилась ошибочная команда. Вместо "**ldi r19,1000**" мы увидим "**ldi r19,0xE8**". Компилятор понял нашу инструкцию так, что от числа 1000 надо взять 8 младших разрядов (остаток от деления на 256) и записать их в регистр. Что важно – нас он о такой ошибке не предупредил!

5. Переходы в программе

Переведем с Паскаля еще одну программу.

текст на Паскале:	*текст на ассемблере:*	*коды инструкций:*
label **do_it_again;** **var** **i:byte;** **begin**	**clr r16** **inc r16** **rjmp 1**	**$2700** **$9503** **$CFFE**

i=0; **do_it_again:** **i=i+1;** **goto do_it_again;** **end.**		

Что это за команды?

CLR (Clear – очистка) записывает в регистр ноль – у этой команды только один аргумент.

INC (Increment – увеличение) прибавляет к регистру единицу (тоже один операнд).

RJMP (Relative Jump – относительный прыжок). С этой командой мы и будем разбираться подробнее. Как видим, аргументом команде задано число 1 – это номер ячейки памяти программ (адрес инструкции) куда мы хотим перейти. То есть мы желаем вернуться к 1-ой инструкции, содержащей команду "inc r16" (вспомним, что адреса инструкций начинаются с нуля).

Слово "относительный" и буква "r" в названии инструкции нас могут особенно не волновать, однако поясним: они означают что в кодовом представлении этой инструкции зашифровывается не абсолютный адрес перехода, а смещение относительно текущего – положительное или отрицательное. В данном случае в коде $CFFE этой инструкции записано число –2, то есть после выполнения инструкции из нового значения счетчика инструкций процессор вычтет 2, так что следующей инструкцией окажется первая. С этим связано ограничение: инструкция rjmp может прыгнуть не дальше, чем на 2048 слов вперед или назад по коду – дело в том, что величина смещения зашифровывается в команде 12 битами, которыми можно представить только такие числа.

Было бы неудобно постоянно запоминать адреса нужных инструкций. Хуже всего, что при вставке или удалении нескольких инструкций адреса бы, конечно, менялись. В связи с этим создатели компилятора ассемблера придумали метки. Рассмотрим ту же программу с участием метки:

текст на ассемблере:	*коды инструкций:*
clr r16	**$2700**
do_it_again:	
inc r16	**$9503**
rjmp do_it_again	**$CFFE**

Это уже больше похоже на Паскальный оригинал, хотя программа на самом деле не изменилась ни на один байт!

Когда компилятор ассемблируя исходный текст доходит до метки **do_it_again**, он запоминает в специальной таблице меток ее имя и текущее значение счетчика инструкций (в нашей программе получится do_it_again=1). **Сама метка**, как можно видеть, **никакой инструкции не порождает.** В дальнейшем, когда он встретит в тексте употребление метки **do_it_again**, он просто подставит вместо нее значение соответствующего адреса, запомненное в таблице, и скомпилирует то, что получилось.

Поэтому при дизассемблировании hex-файла в эмуляторе вы никаких меток, конечно, там не найдете. **Скомпилируйте программу, откройте ее в симуляторе и потрассируйте ее, чтобы увидеть как работает переход, организуя цикл с увеличением значения регистра №16 на единицу каждые два такта.**

Таким образом метка является как бы **подстановочным термином**, определяемым пользователем. Можно считать, что ассемблер перед компиляцией просматривает входной файл, составляет **словарь подстановочных терминов**, и заменяет все вхождения таких терминов нужными значениями или выражениями. Собственно на процесс преобразования мнемоник инструкций в коды подстановочные термины **не влияют**. Прежде чем двинуться дальше, рассмотрим другие виды подстановочных терминов.

6. Макроподстановки в ассемблере

Часто для удобства программиста полезно бывает заменить, например, название регистра или какое-нибудь число на более понятное слово. Иногда это просто улучшает читаемость программы, иногда это бывает необходимо для облегчения модификации программ. Посмотрим наш первый пример, переписанный с такими "удобствами".

`var`	`.def VAR_A=r16`	
`  a,b,c:byte;`	`.def VAR_B=r17`	
`begin`	`.def VAR_C=r18`	
`  a:=1;`	`.equ VALUE_A=1`	
`  b:=2;`	`.equ VALUE_B=2`	
`  c:=a+b;`		
`end.`	`ldi VAR_A,VALUE_A`	`$E001`
	`ldi VAR_B,VALUE_B`	`$E012`
	`add VAR_A,VAR_B`	`$0F01`
	`mov VAR_C,VAR_A`	`$2F20`

Мы видим, что для определения **подстановочных терминов** используются **директивы компилятора** – выражения, не влияющие на код самой программы, но управляющие работой компилятора. В данном случае директивы **DEF** и **EQU** заставляют компилятор записать кое-что в **таблицу подстановочных терминов**. Названия происходят от слов Define (определить) и Equivalent (равно).

Обратим внимание, что подстановочные термины для имен регистров определяются директивой DEF, а для чисел – директивой EQU.

Важно понять что такие макроподстановки не являются определением переменных или констант, как в Паскале. Для них не выделяется память и вообще они не изменяют конечного кода программы. Они все будут заменены в тексте программы еще до компиляции (однако текст во входном файле, конечно, не будет перезаписан).

Основных преимуществ от использования подстановочных терминов два:
- во-первых мы можем, если будем думать головой, назначить регистрам или каким-нибудь значениям "говорящие" имена, например, если вы назовете число 314 именем PI_100, то в дальнейшем вы легко догадаетесь, что имелось в виду число "пи" умноженное на 100;
- во-вторых, если, например, значение числа "пи" умноженного на 100 было употреблено в программе в 10 разных местах, а на следующий день вышел декрет правительства, что умножать нужно было не на 100, а на 0100 (то есть в восьмеричной системе, всего 64) нам легко будет изменить это значение переписав лишь саму макроподстановку; так же и с регистрами – если однажды нам покажется, что регистр № 16 неудобен для использования в качестве хранилища значения A (потому что мы использовали его еще для чего-нибудь), мы можем просто вписать в макроподстановке имя другого регистра, а не лазить по всей программе "поиском и заменой".

Отметим еще забавную особенность. Теперь исходный текст нашей программы составляет 154 байта, а компилируется он в 4 слова, то есть всего в 8 байт. Вообще с программами на ассембле-

ре такое бывает часто – пишешь полдня, а результат малюсенький.

6. Слово состояния (флаги процессора) и условные переходы

Вернемся к рассмотрению программы с циклом. Сам по себе такой цикл бесполезен, потому что он безвыходный (хотя в программах для контроллеров это встречается часто) – попав в него контроллер уже никогда не выберется обратно. Чтобы избежать этого в Паскале мы обычно каждый раз в конце цикла проверяем какое-нибудь условие и осуществляем переход только если оно выполняется (или не выполняется). Посмотрим, как это происходит в ассемблере.

текст на Паскале:	*текст на ассемблере:*	*коды инструкций:*
label ** do_it_again;** **var** ** i:byte;** **begin** ** i=0;** ** do_it_again:** ** i=i+1;** ** if i<10 then** ** goto do_it_again;** **end.**	**.def REG_I=r16** **.equ MAX_I=10** **clr REG_I** **do_it_again:** **inc REG_I** **cpi REG_I,MAX_I** **brlo do_it_again**	 **$2700** **$9503** **$300A** **$F3E8**

Первые две инструкции – очистка регистра и прибавление единицы – остались такими же, как и прежде. У нас они вопросов не вызывают, хотя мы и ввели теперь два подстановочных термина – но поскольку мы разобрались, что это только лишь текстовые замены, мы совершенно не переживаем по этому поводу. Можно лишь отметить что некоторые программисты предпочитают писать подстановочные термины большими буквами, чтобы легче отличать их от всего остального. Ну это не так важно.

Однако что дальше? Мы видим, что инструкция проверки условия и перехода "if...then goto" превратилась в две отдельные инструкции. Много напрягая мозги можно догадаться, что по-видимому инструкция **CPI** каким-то образом сравнивает регистр r16 с числом 10, а инструкция **BRLO** осуществляет переход на команду по адресу 1. Однако очевидно, этот переход выполняется не всегда, а только если результат сравнения был удовлетворительным.

Как же обмениваются информацией эти две команды?

Откроем в симуляторе hex-файл с нашей программой. Сразу обратим внимание, что при дизассемблировании команда BRLO получила немного другое название – просто у комадны, выполняющей данное действие (мы скоро узнаем, какое) есть несколько разных мнемоник, которые, в общем, правильно описывают ее назначение. Дизассемблер попросту не знает, как мы записали эту команду в исходном тексте, однако поскольку никакой разницы нет, его это не волнует. И нас волновать не должно.

Откроем окно состояния процессора и станем трассировать программу. Мы увидим, что при выполнении некоторых команд в восьми маленьких клеточках с общей надписью **Flags** появляются или исчезают галочки. Путем мучительных раздумий можно догадаться, что это восемь бит одного специального регистра процессора – регистра флагов или, как его еще называют, слова состояния процессора (Status Register, SREG и т.п.). Дело в том, что записью нулей и единичек в биты этого регистра процессор отмечает для себя кое-какие важные вещи и впоследствии, при выполнении некоторых инструкций он может проверять состояние этих "флажков" и выполнять эти инструкции по-разному в зависимости от того, что он там увидел.

Теперь все просто. Инструкция **CPI** (Compare with Immediate – сравнить с непосредственным значением) сравнивает значение в указанном регистре с указанным значением и в зависимости от результата устанавливает различные флажки в слове состояния. Фактически эта инструкция вычитает из регистра заданное число, только результат **никуда не записывает**, а лишь устанавливает в соответствии с характером этого результата нужные флаги (см. ниже).

Инструкция **BRCS** (Branch if Carry-flag is Set – перейти если флаг переноса установлен, то есть равен единице) проверяет флажок переноса (он отмечен буквой C в окне симулятора, это самый младший бит) и осуществляет переход на указанный адрес, если этот флажок установлен. Флаг переноса означает что при предыдущей команде вычитания (или сравнения) первое число оказалось меньше второго и нам пришлось "занять" как мы это делаем, вычитая в столбик. Поскольку предыдущей командой была команда CPI, это в данном случае означает, что значение в регистре №16 оказалось меньше 10.

И вот, поскольку **BRCS** осуществляет переход если предыдущая команда сравнения показала, что первый операнд меньше второго, эта команда также называется **BRLO**, что расшифровывается как "Branch if Lower" – перейти, если меньше. Поэтому-то у этой инструкции две мнемоники. В одних случаях мы хотим пояснить что нас интересует собственно флаг **Carry**, в других случаях мы имеем в виду, что нам интересен результат сравнения чисел. По сути же выполняется одно и то же действие.

Поясним еще значение некоторых флагов. Флаг переноса кроме вычитания участвует еще в операциях сложения. Как нетрудно предвидеть, он устанавливается если в результате сложения двух чисел (размером по одному байту) они оказываются слишком большими и необходимо перенести единичку в следующий (восьмой, если считать с нуля) разряд. Например если складывать $85+$B9=$13E – единица "вылезает" за пределы байта. Она-то и помещается в флаг переноса. Специальная команда ADC может складывать два регистра и прибавлять к результату эту самую единичку из флага переноса (если она там есть) – это позволяет удобно суммировать многобайтные числа в несколько инструкций. Есть также инструкция перехода, если флаг переноса сброшен (равен 0) – BRCC (Branch if Carry-flag is Cleared) – она также известна как BRSH (Branch if Same or Higher – перейти, если числа равны или первое больше) по тому действию, которое она выполняет после инструкции сравнения чисел.

Следующий флаг, первый бит (считая с 0) в служебном регистре SREG – флаг нуля, поэтому он и обозначен буквой Z. Он устанавливается если в результате предыдущей операции получился 0, поэтому, например, он устанавливается в самом начале программы при выполнении инструкции CLR. С его участием тоже можно выполнять условные переходы – дело в том, что после инструкции сравнения он устанавливается в 1 если числа были равны (тогда в результате их "виртуального" вычитания получается именно ноль) – поэтому инструкция записывается мнемоникой BREQ (Branch if Equal – перейти, если числа равны). Противоположная по смыслу инструкция, совершающая переход если флаг нуля сброшен (числа не равны) называется BRNE (Branch if Not Equal).

Следующие флаги – отрицательности (N), переполнения (V), знака (S) и полупереноса (H) также устанавливаются большей частью в арифметических и некоторых других операциях. Шестой бит (T) это пользовательский флаг – специальной инструкцией в него можно скопировать любой бит из любого регистра а потом записать его в какой-нибудь другой бит другого регистра. Кроме того есть инструкции (BRTS и BRTC), совершающие переход в зависимости от состояния этого флага.

Последний флаг (I) специальный – он разрешает или запрещает выполнение процессором про-

цедур прерываний по сигналам.

Все флаги можно установить или сбросить специальными инструкциями SE* и CL*, где вместо звездочки подставляется буква, обозначающая нужный флаг.

7. Служебные регистры

Кроме 32 регистров общего назначения процессору микроконтроллера доступны еще 64 служебных регистра. Запись информации в них, или чтение информации оттуда позволяет общаться с различными устройствами микроконтроллера. Например в служебных регистрах 4 и 5 хранится значение, показывающее какое напряжение было измерено аналого-цифровым преобразователем, регистр номер $17 содержит 8 бит, показывающих, включены ли соответствующие 8 ножек микросхемы на ввод логических сигналов или на вывод (когда они подключены на вывод логических сигналов, к ним можно с огромным интересом присоединять светодиоды, например), а самый старший служебный регистр номер $3F – это как раз наш знакомый регистр состояния процессора, содержащий флаги.

Для записи информации в служебные регистры используются специальные команды **IN** и **OUT**. Их названия связаны с тем что такие регистры вообще говоря не являются регистрами процессора, а внешние по отношению к нему – в старинных компьютерах они были в виде отдельных микросхем, припаиваемых к общей шине данных и адресов.

Команда IN принимает в качестве операндов имя общего регистра (от r0 до r31) и номер служебного регистра. Например "in r16,$2F" запишет в 16-й регистр число из служебного регистра номер $2F – этот регистр хранит текущее значение встроенного таймера.

Команда OUT принимает такие же операнды, но в обратном порядке. Например команда "out $1D,r5" запишет число из регистра №5 в служебный регистр номер $1D – этот регистр используется для записи значений в перепрограммируемое ПЗУ данных. Например если мы сделали на основе контроллера лабораторный стенд, нам будет полезно записывать в ПЗУ пользовательские настройки перед выключением этого стенда, или, например, хранить там таблицу синусов.

Однако, каждый раз вспоминать и писать номера служебных регистров неудобно – кроме того, они могут быть разными у контроллеров различных типов. Очевидно, было бы удобно использовать для "обзывания" этих регистров подстановочные термины. К счастью, существуют готовые файлы со списками таких макроподстановок. Например в пакет AVR Studio они входят – это файлы с расширением "**inc**". Чтобы "подключить" такой файл к своей программе, нужно просто написать в ней строчку вроде:

```
.include "tn15def.inc"
```

Когда компилятор дойдет до такой строки, он откроет указанный файл, прочитает также его, как будто бы это была часть нашего файла, запомнит макроподстановки, если они там есть, и потом продолжит компилирование нашей программы. Разумеется, чтобы этот файл нормально открылся, он должен лежать в том же каталоге – в противном случае нужно прописать полный путь к нему.

Например в файле с указанным названием записаны имена для служебных регистров контроллера **Tiny15** – причем эти имена написаны здесь так же, как в руководстве **ATtiny15.pdf**. Что чрезвычайно удобно.

Мы будем использовать служебные регистры в первую очередь, чтоб изменять напряжения на ножках микроконтроллера. Даже у самых маленьких 8-ногих микроконтроллеров под землю,

питание и сброс заняты только три ноги. Остальные 5 контактов могут быть использованы в произвольных целях. Например у ATtiny15 эти пять ножек отвечают 5 младшим битам "порта B". Порт B (бэ-латинское) обслуживается целыми тремя служебными регистрами, из которых нас интересуют два.

Служебный регистр $17 (называемый также DDRB) определяет своими битами, включены ли ножки на вход или на выход. Если они включены на выход, то в них можно будет создать напряжение либо равное потенциалу "земли" (0 Вольт), либо равное потенциалу питания (5 Вольт обычно). Если же они включены на вход, то на них не будет никакого напряжения, как будто они вообще от всего оторваны, однако контроллер будет сам измерять какое напряжение создается на них (внешней схемой, например, подсоединенным тумблером) и в одном из служебных регистров в соответствующие биты будет записываться 0 или 1.

Если порт включен на выход, то напряжения на ножках контролируются служебным регистром номер $18 (он называется PORTB). Если в какой-то бит записана единица, на соответсвующей ноге появляется 5 Вольт. Если ноль – появляется ноль.

Вот пример программы, включающей 5 вольт на ножке PB0 и 0 Вольт на ножке PB4.

текст на ассемблере:	*коды инструкций:*
.include "c:\atmel\tn15def.inc"	
ldi r16,0b00010001	**$e101**
out DDRB,r16	**$bb07**
ldi r16,0b00000001	**$e001**
out PORTB,r16	**$bb08**

Можно скомпилировать эту программу и залить ее в контроллер, чтобы проверить (вольтметром или светодиодом) что она действительно работает.

Другой вариант – использовать симулятор VMLAB, который позволяет конструировать несложные виртуальные схемы с использованием светодиодов, резисторов, выключателей и микроконтроллера. О нем мы поговорим позже.

Сейчас же заметим, что собственно изучение ассемблера – это только половина изучения программирования для микроконтроллеров. Как видим, еще нужно знать как служебные регистры позволяют управлять ресурсами микроконтроллера. Исчерпывающая информация по этому поводу представлена в официальных руководствах фирмы ATMEL, которые в виде PDF-файлов удобно скачиваются, например, с www.atmel.ru и которые требовалось попробовать переводить в качестве первого задания.

Что же касается самого ассемблера, эти 7 параграфов охватили его еще отнюдь не полностью, однако этого достаточно чтобы выполнить задание по написанию программы, мигающей светодиодом, подключенным к одной из ножек. Что от вас и требуется в скором времени.