Внимание!программы на ассемблере не

длжны запускаться на трансляцию с рабочего стола т.к. не допускаются русские буквы.

Принципиальная схема и макет

Для понимания того что из себя представляет наша конструкция, для которой мы будем писать программу, приведу ниже принципиальную схему устройства.

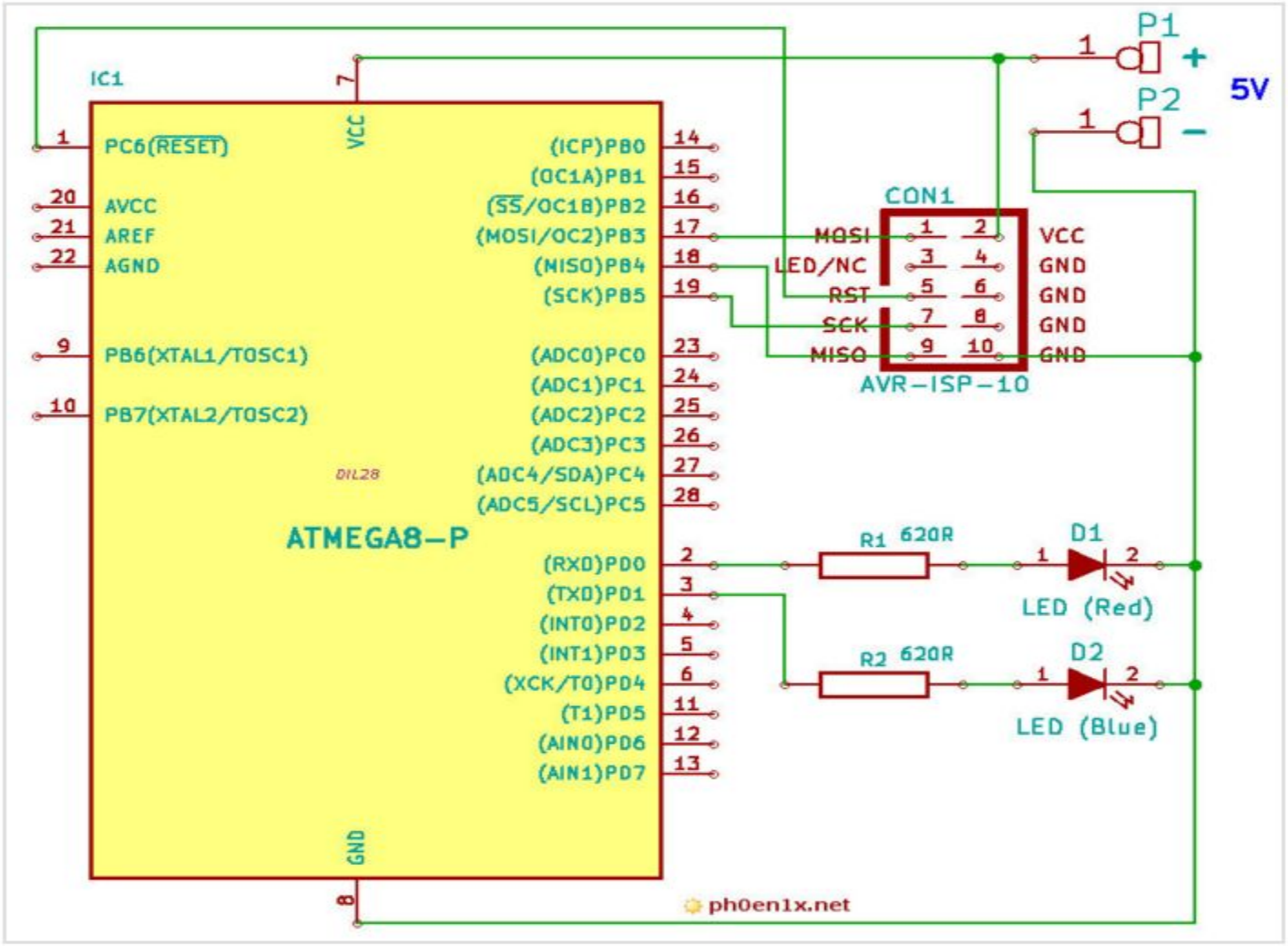


Рис. 1. Принципиальная схема мигалки на светодиодах и микроконтроллере ATmega8.

Примечание: принципиальная схема нарисована за несколько минут в программе **Eeschema,** которая входит в комплекс EDA(Electronic Design Automation) программ **KiCAD** (для Linux, FreeBSD, Solaris, Windows). Очень мощный профессиональный инструмент, и что не мало важно - свободный!

Схема устройства состоит из микроконтроллера ATmega8 и двух светодиодов, которые подключены через гасящие резисторы. К микроконтроллеру подключен ISP-коннектор для осуществления программирования через программатор. Также предусмотрены клеммы для подключения внешнего источника питания напряжением 5В.

То как выглядит данная схема в сборе на макетной паяльной панели (BreadBoard) можно посмотреть на рисунке ниже:

Исходный код программы на Ассемблере

Разработанная нами программа будет попеременно зажигать и гасить два светодиода. Светодиоды подключены к двум пинам PD0 и PD1 микроконтроллера.

Ниже приведен исходный код программы на Ассебмлере(Assembler, Asm) для микроконтроллера ATmega8. Сохраните этот код в файл под названием leds_blinking.asm для последующей работы.

```
; Светодиодная мигалка на микроконтроллере ATmega8
; https://ph0en1x.net

.INCLUDEPATH "/usr/share/avra/" ; путь для подгрузки INC файлов
.INCLUDE "m8def.inc"            ; загрузка предопределений для ATmega8
.LIST                           ; включить генерацию листинга

```

Начнем с **комментариев.** В **ассемблере** как и в других языках, традиционно есть два вида комментариев, **однострочный** «;» и **многострочный** «/* */«.

Код начинается с директивы **«.include «m8def.inc»»**, которая **подключает ассемблерный файл «m8def.inc»** с константами **имен регистров** и их **битов**. Для каждого контроллера заголовочный файл **свой**.

Директивы **«.nolist»** и **«.list» не обязательны,** они несут **вспомогательную** роль. Ассемблерный **код**, который находится **между** данными **директивами не выводится** в выходной **листинг** (т.е. в файл *.lss). Это нужно для отладки.

Директивы **«.set»**, **«.equ»**, **«.def»** задают **константы** с **значениями**. В основном используют:

- **«.equ»** — для обозначения **имен регистров** и их **бит** для портов **ввода/вывода** и **периферии**;
- **«.def»** — для обозначения **имен регистров общего назначения** (т.е. для **R0** … **R31**)

Кстати, подключаемый ранее файл **«m8def.inc»**, как раз содержит **«.equ»** и **«.def»** описывающие **регистры**, для удобного обращения к ним по имени, не используя адрес. Например, регистр **DDRB** (для настройки пинов порта **B** на **вход** или **выход)**, это просто адрес **0x17**.

Вы можете найти файл **«m8def.inc»** в директории вашего компилятора, если посмотрите в него, то увидите

```
; -- основной цикл программы --
Start:
    SBI PORTD, PORTD0 ; подача на пин PD0 высокого уровня
    CBI PORTD, PORTD1 ; подача на пин PD1 низкого уровня
    RCALL Wait        ; вызываем подпрограмму задержки по времени
    SBI PORTD, PORTD1 ; подача на пин PD1 высокого уровня
    CBI PORTD, PORTD0
    RCALL Wait
    RJMP Start        ; возврат к метке Start, повторяем все в цикле
```

Начиная с метки "**Start:**" идет основной код программы.

При помощи инструкции "**SBI**" выполняем установку бита **PORTD0** (предопределен в файле m8def.inc) в порте **PORTD** чем подаем на пин PD0 высокий уровень. Используя инструкцию "CBI" выполняется очистка указанного (**PORTD1**) бита в порте PORTD и тем самым подаем низкий уровень на пин PD1.

```
    RJMP Start          ; возврат к метке Start, повторяем все в цикле

; -- подпрограмма задержки по времени --
Wait:
    LDI  R17, Delay     ; загрузка константы для задержки в регистр R17
WLoop0:
    LDI  R18, 50        ; загружаем число 50 (0x32) в регистр R18
WLoop1:
    LDI  R19, 0xC8      ; загружаем число 200 (0xC8, $C8) в регистр R19
WLoop2:
    DEC  R19            ; уменьшаем значение в регистре R19 на 1
    BRNE WLoop2         ; возврат к WLoop2 если значение в R19 не равно 0
    DEC  R18            ; уменьшаем значение в регистре R18 на 1
    BRNE WLoop1         ; возврат к WLoop1 если значение в R18 не равно 0
    DEC  R17            ; уменьшаем значение в регистре R17 на 1
    BRNE WLoop0         ; возврат к WLoop0 если значение в R18 не равно 0
RET                     ; возврат из подпрограммы Wait
```

Дальше с помощью инструкции **RCALL** выполняем относительный вызов подпрограммы которая начинается с метки "**Wait:**".

После завершения подпрограммы (в нашем случае это задержка по времени) программа вернется к позиции где был выполнен вызов подпрограммы и с этого места продолжится выполнение последующих операторов.

После вызова подпрограммы задержки "Wait" следуют вызовы инструкций **SBI** и **CBI** в которых выполняется установка битов порта **PORTD** таким образом что теперь на пин **PD0** у нас поступит низкий уровень, а на пине **PD1** будет высокий уровень.

По завершению данных инструкций следует снова вызов подпрограммы задержки "Wait", а дальше следует инструкция "**RJMP**" которая выполнит относительный переход к указанной метке - "Start", после чего программа снова начнет установку битов в порте с задержками по времени.

Таким образом выполняется реализация бесконечного цикла в котором будут "дергаться" пины порта **PORTD** микроконтроллера и поочередно то зажигаться то гаснуть светодиоды что подключены к каналам данного порта (пины PD0, PD1).

После основного цикла программы следует наша **подпрограмма задержки по времени.** Принцип ее работы заключается в выполнении трех вложенных циклов, в каждом из которых происходит вычитание (**DEC**) единички из числа что хранится в отдельном регистре пока значение не достигнет нуля. Инструкция "**DEC**" декрементирует значение указанного регистра и требует для этого 1 рабочий такт процессора.

При помощи инструкций "**BRNE**" (переход на метку если не равно) производит сравнение операндов и анализ установленных флагов процессора. При ее помощи выполняется контроль за значением флага нуля (Zero Flag, ZF), который будет установлен если после выполнения команды "DEC" над значением регистра в нем появится 0. Инструкция "BRNE" требует 1/2 такта процессора.

Таким образом, использовав несколько вложенных циклов, ми заберем у ЦПУ некоторое количество тактов и выполняем некоторую задержку по времени которая требуется на обработку данных во вложенных циклах. По умолчанию, без установки фьюзов что задают источник и частоту тактового генератора, в микроконтроллере ATmega8 используется откалиброванный внутренний RC-генератор с частотой 1МГц. Если же мы изменим частоту МК на 4Мгц то наши светодиоды начнут мигать в 4 раза быстрее, поскольку на каждую операцию вычитания и сравнения будет тратиться в 4 раза меньше времени.

Завершается подпрограмма инструкцией "**RET**", которая выполняет возврат из подпрограммы и продолжение выполнения инструкций с того места где эта подпрограмма была вызвана (на основе сохраненного адреса возвращения, который сохранился в стеке при вызове инструкции "RCALL").

Стек – специальным образом организованная последовательность ячеек памяти с дисциплиной обслуживания «последним пришёл – первым вышел» (**LIFO**: Last-In – First-Out). При занесении в стек новых данных предыдущие данные сохраняются, но становятся временно недоступными («опускаются»). Выгружаются из стека («поднимаются», «выталкиваются») данные в обратном порядке. Стек может быть реализован как аппаратно, так и программно. Аппаратный стек (Hardware Stack) выполняется непосредственно в составе процессора в виде набора специальных регистров и, как правило, имеет небольшую глубину. Программный стек (Software Stack) организуется в оперативной памяти; глубина определяется размером и степенью использования памяти.

Адрес очередной свободной ячейки стека содержится в специальном регистре – указателе стека **SP** (Stack Pointer). При записи в стек число помещается в ячейку с адресом, содержащимся в указателе стека, после чего содержимое указателя стека уменьшается на единицу (рис. 23, *а*). При чтении из стека производится выборка содержимого ячейки по адресу, на единицу большему содержимого указателя стека (рис. 23, *б*). Таким образом, при записи стека и чтении из него содержимое указателя стека изменяется.

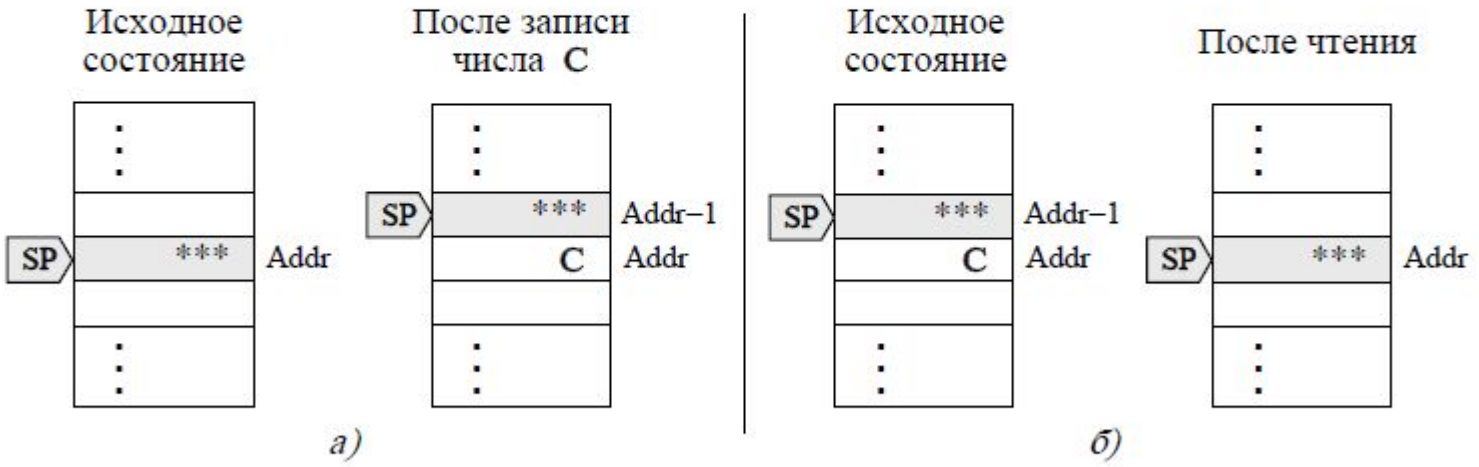


Рис. 23. *Операции записи в стек (а) и чтения из стека (б):*
*** – очередная свободная ячейка стека; Addr – адрес

В большинстве **AVR**-микроконтроллеров стек размещается в оперативной памяти. Указатель стека представляет собой пару 8-разрядных регистров **SPH** (старший байт указателя стека) и **SPL** (младший байт указателя стека), находящихся в адресном пространстве регистров ввода-вывода. В микроконтроллерах, размер оперативной памяти которых не превышает 256 байт, для хранения указателя стека используется только один регистр **SPL** (**SP**). Микроконтроллеры, не имеющие оперативной памяти, содержат трёхуровневый аппаратный стек.

Организуемый в оперативной памяти стек «растёт» от старших адресов к младшим. Учитывая, что начальное значение указателя стека после сброса микроконтроллера равно нулю, в инициализирующей (начальной) части программы необходимо произвести его установку, если предполагается использование хотя бы одной подпрограммы. При организации стека во внутренней оперативной памяти это может быть сделано, например, следующим образом:

```
ldi    R16, low(RAMEND)        ; младшая часть адреса RAMEND
out    SPL, R16                ; инициализация SPL
ldi    R16, high(RAMEND)       ; старшая часть адреса RAMEND
out    SPH, R16                ; инициализация SPH
```

где команда `OUT` осуществляет запись содержимого регистра общего назначения в регистр ввода-вывода; `RAMEND` – символическое имя адреса последней

ячейки внутренней оперативной памяти. Для того, чтобы использовать в про-

СТЕК.

Еще один вариант использования памяти - это стек. Данные в стек (из стека) можно поместить командами PUSH / POP. Кроме того данные помещаются в стек при вызове (выходе) подпрограмм командами RCALL / RET.

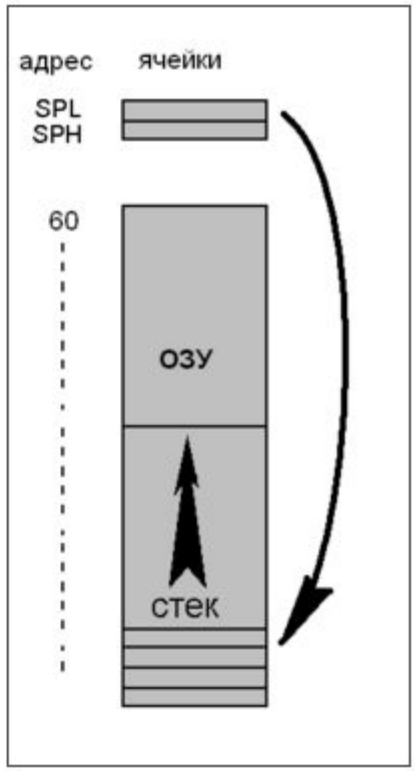
Сам стек расположен в «нижней» части ОЗУ и «растет» в верх. Для работы процессора со стеком предусмотрен указатель в виде регистров ввода вывода SPH:SPL. Это указатель на вершину стека. То есть на байт следующий за «загнанным» в стек последним. Изначально он должен указывать на самый последний байт памяти (константа RAMEND)

```
ldi     temp,Low(RamEnd)
out     SPL,temp
ldi     temp,HIGH(RamEnd)
out     SPH,temp
```

Этот кусок кода должен быть выполнен при инициализации процессора, если мы хотим использовать вызовы подпрограмм и прерывания. Однако следует учесть, что в некоторых моделях, не имеющих ОЗУ, реализован аппаратный стек, не требующий инициализации.

Кроме того, что стек позволяет временно хранить данные, он может быть использован для

передачи данных в подпрограммы и из них.




```
      push R10
      push R15
      call Shift
      pop   R10
///
      push R14
      push R16
      call Shift
      pop   R14
///
Shift:
      pop   temp1
      pop   temp2
      pop   Shifter
      pop   Shifted
(shifted<<shifter)
      push Shifted
      push temp2
      push  temp1
      ret
```

Команда out работает только с регистрами

Команда OUT - записать данные из регистра в порт I/O

Описание:

Команда сохраняет данные регистра Rr в регистровом файле пространства I/O (порты, таймеры, регистры конфигурации и т.п.).

Операция

(i) P <-- Rr

Синтаксис	Операнды:	Счетчик программ:
(i) OUT P,Rr	0 < r <31, 0 < P <63	PC<-- + 1

Запомните приведенный выше пример и НИКОГДА так не делайте. Хотя данный кусок кода призван продемонстрировать возможности работы со стеком, он, по сути, является ярким примером «вирусного» стиля написания программ, хоть и эффективного, но весьма неряшливого.

```
.CSEG                           ; начало сегмента кода
.ORG 0x0000                     ; начальное значение для адресации

; -- инициализация стека --
LDI R16, Low(RAMEND)  ; младший байт конечного адреса ОЗУ в R16
OUT SPL, R16          ; установка младшего байта указателя стека
LDI R16, High(RAMEND) ; старший байт конечного адреса ОЗУ в R16
OUT SPH, R16          ; установка старшего байта указателя стека

.equ    Delay   = 5   ; установка константы времени задержки
```

Директива "**.CSEG**" определяет начало программного сегмента (программный код). Для определения сегмента данных или памяти EEPROM используются директивы ".DSEG" и ".ESEG" соответственно. Таким образом выполняется распределение памяти по сегментам. Каждый из сегментов может использоваться в программном коде только раз, по умолчанию если не указана ни одна из директив используется **сегмент кода**.

При помощи директивы "**.ORG**" компилятору указывается начальный адрес "0x0000" сегмента данных, в данном случае это сегмент кода (Code**SEG**ment).

Дальше в коде происходит **инициализация стека**. Стек (**Stack**) - это область памяти (как правило у всех AVR чипов размещается в SRAM), которая используется микропроцессором для хранения и последующего считывания адресов возврата из подпрограмм, а также для других пользовательских нужд. Стек работает по принципу LIFO (Last In - First Out, последним пришёл - первым вышел). Для адресации вершины стека используется указатель стека - SP (Stack Pointer), это может быть однобайтовое или двухбайтовое значение в зависимости от доступного количества SRAM памяти в МК.

При помощи инструкции "**LDI**" мы загружаем значение младшего байта конечного адреса ОЗУ "**Low(RAMEND)**" (предопределенная константа в файле m8def.inc что содержит адрес последней ячейки SRAM) в регистр **R16**, а потом при помощи инструкции OUT выполняем вывод данного значения из регистра R16 в порт **SPL** (Stack Pointer Low). Таким же образом производим инициализацию старшего байта адреса в указателе стека **SPH**.

Если не произвести инициализацию стека то возврат из подпрограмм станет невозможным, к примеру в приведенном коде после выполнения инструкции перехода к подпрограмме "RCALL Wait" ворзврат не будет произведен и программа не будет работать как нужно. Новички часто упускают эту особенность и появляются вопросы на подобии: "В программе у меня все верно но она почему-то не работает в МК?".

Директива "**.equ**" выполняет присвоение указанному символьному имени "Delay" числового значения "5". Имя должно быть уникальным, а присвоенное значение не может быть изменено в процессе работы программы.

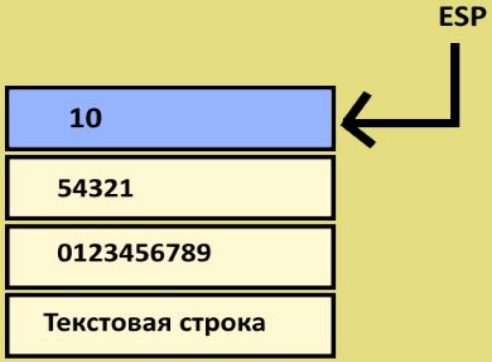
Стек

Стек это можно сказать полка книжек, в регистре ESP хранится адрес последней добавленной “книги”.

Командой PUSH мы добавляем новую “книжку” а командой POP убираем. В этих “книжках” хранятся значения 4 байта. Но как же тогда там содержаться строки наверное спросите вы, ответ таков в стек помещаются не сами строки а указатели на них, то есть адреса памяти где они лежат.

Вот так будет выглядеть наша стопка “книжек” после выполнения следующей команды:

PUSH 10



Как вы видите сверху добавилась еще одна “книга” которая имеет синюю заливку. В ней содержится значение аргумента нашей команды. В языке ассемблера все числа хранятся в шестнадцатеричной системе счисления, то есть мы только что добавили элемент со значением 16 а не 10 как вы могли сначала подумать. Для того чтобы перевести число из одной системы в другую можно использовать обычный Windows калькулятор, или любую другую программу которая это делает.

Теперь посмотрим что будет со стеком после выполнения следующей команды

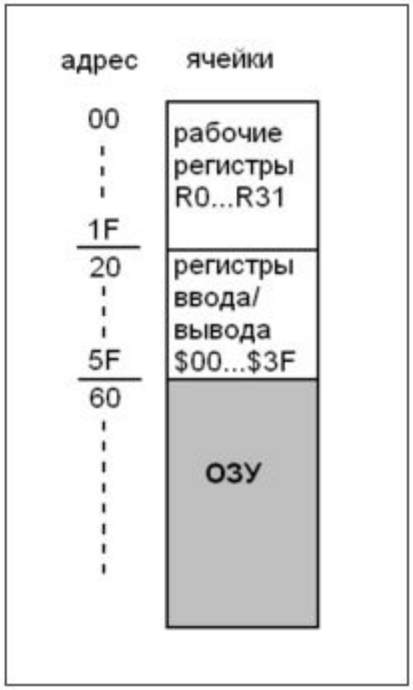
POP EAX

Как вы видите самый верхний элемент удалился и его значение сохранилось в регистре EAX. Думаю что особых проблем с освоением материала у вас не возникло, а даже если проблемы все же возникли и вы не понимаете что то из перечисленного выше то на практике все встанет на свои места. Это я просто рассказываю основы чтобы вы хотя бы имели представление про этот язык программирования.

ПАМЯТЬ.

По сути, внутренняя память устройства - это те же регистры, но с несколько иной системой доступа. Мы не можем обращаться к ним напрямую как к рабочим регистрам. Если следовать аналогии с фирмой - то ОЗУ это некий архив, в который мы не можем пойти и поработать, но заказанную нами информацию из архива нам принесут, и если нужно - поместят туда нашу информацию.

То есть для работы, с какой либо ячейкой памяти нам необходимо сначала скопировать информацию из нее в рабочий регистр, затем выполнив необходимые действия скопировать информацию обратно в память.



В системе команд микропроцессора AVR есть различные варианты обращения к оперативной памяти.

- прямая адресация.
- косвенная адресация
- косвенная адресация со смещением.
- косвенная адресация с авто инкриментом/декриментом.

Самая простая - это прямая адресация. выполняется командами

lds reg,number

sts reg,number

Эти команды работают с регистром (reg) и ячейкой памяти с номером (number). Команда **lds** грузит (**l**oa**d**) значение из памяти в регистр, а команда **sts** соответственно устанавливает (**s**e**t**) значение регистра в память.

Косвенная адресация - это адресация по указателю(адресу) записанному в рабочем регистре.

ld reg,longreg

st reg,longreg

Здесь адрес читаемой (записываемой) ячейки находится в регистре **longreg**. Так как размер ОЗУ чипа может быть больше 256 байт, то используется двойной регистр. это регистровые пары R26:R27(регистр X), R28:R29(регистр Y), R30:R31(регистр Z).

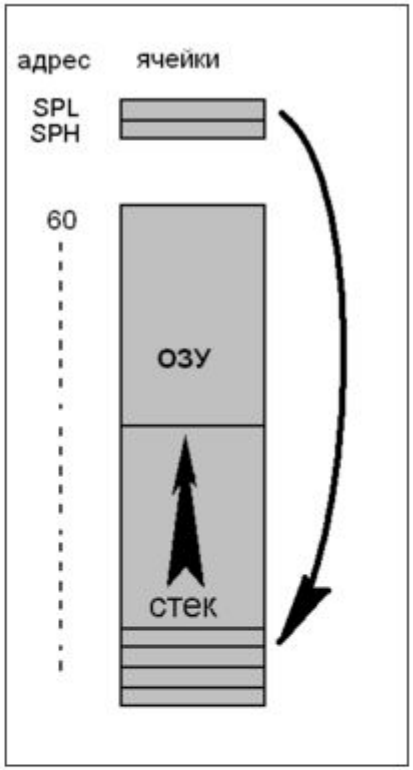
Косвенная адресация со смещением - это гибрид первых двух способов. когда при записи и чтении используется указатель **longreg** и смещение **number**. Такие форму адресации удобно применят, когда речь идет о полях структуры или о массиве. В первом случае регистровая пара указывает на начало структуры, а константа - на смещение поля структуры относительно начала. Во втором - начало таблицы - константа, а индекс - указатель.

С авто инкрементом/декрементом - это практически а же косвенная адресация, только позволяющая одновременно изменять указатель. Таким образом, экономится несколько команд и машинных циклов, что достаточно важно при решении задач реального времени.

Следует отметить, что некоторые команды применимы и в моделях микрочипов, где нет памяти. Как видно из рисунка и рабочие регистры, и регистры ввода вывода и память расположены в одном адресном пространстве. По этому, часть регистров общего назначения можно использовать как небольшой массив.

СТЕК.

Еще один вариант использования памяти - это стек. Данные в стек (из стека) можно поместить командами PUSH / POP. Кроме того данные помещаются в стек при вызове (выходе) подпрограмм командами RCALL / RET.



Сам стек расположен в «нижней» части ОЗУ и «растет» в верх. Для работы процессора со стеком предусмотрен указатель в виде регистров ввода вывода SPH:SPL. Это указатель на вершину стека. То есть на байт следующий за «загнанным» в стек последним. Изначально он должен указывать на самый последний байт памяти (константа RAMEND)

```
ldi     temp,Low(RamEnd)
out     SPL,temp
ldi     temp,HIGH(RamEnd)
out     SPH,temp
```

Этот кусок кода должен быть выполнен при инициализации процессора, если мы хотим использовать вызовы подпрограмм и прерывания. Однако следует учесть, что в некоторых моделях, не имеющих ОЗУ, реализован аппаратный стек, не требующий инициализации.

Кроме того, что стек позволяет временно хранить данные, он может быть использован для

передачи данных в подпрограммы и из них.

```
      push R10
      push R15
      call Shift
      pop   R10
///
      push R14
      push R16
      call Shift
      pop   R14
///
Shift:
      pop   temp1
      pop   temp2
      pop   Shifter
      pop   Shifted
(shifted<<shifter)
      push Shifted
      push temp2
      push  temp1
      ret
```

Запомните приведенный выше пример и НИКОГДА так не делайте. Хотя данный кусок кода призван продемонстрировать возможности работы со стеком, он, по сути, является ярким примером «вирусного» стиля написания программ, хоть и эффективного, но весьма неряшливого.

```
;Программа управления светодиодом
;Тактовая частота микроконтроллера 1 МГц

;Команды управления
.include"m8def.inc"          ;Присоединение файла описаний микроконтроллера ATmega8

.def temp = R16              ;Присвоить имя temp регистру общего назначения - R16

;Начало программного кода

.cseg                        ;Выбор сегмента памяти для хранения текущей программы
.org 0                       ;Установка адреса начала программы на 0

;Инициализация портов ВВ

ldi temp, 0b00000001         ;Запись единицы в младший разряд регистра temp
out DDRC, temp               ;Подключение младшего вывода порта C (PC0) на выход
out PORTC, temp              ;устанавливаем в единицу младший вход порта C (зажигаем светодиод)
```

Объектный файл это:

Толкование | Перевод

Объектный файл

Обье́ктный мо́дуль (также — *объектный файл*, англ. *object file*) — файл с промежуточным представлением отдельного модуля программы, полученный в результате обработки исходного кода компилятором. Объектный файл содержит в себе особым образом подготовленный код (часто называемый *бинарным*), который может быть объединён с другими объектными файлами при помощи редактора связей (компоновщика) для получения готового исполнимого модуля либо библиотеки.

Объектные файлы представляют собой блоки машинного кода и данных, с неопределенными адресами ссылок на данные и процедуры в других объектных модулях, а также список своих процедур и данных. Компоновщик собирает код и данные каждого объектного модуля в итоговую программу, вычисляет и заполняет адреса перекрестных ссылок между модулями. Также в процессе компоновки происходит связывание программы со статическими и динамическими библиотеками (являющихся архивами объектных файлов).

Цикл «ПОКА»

While X > 0 do S

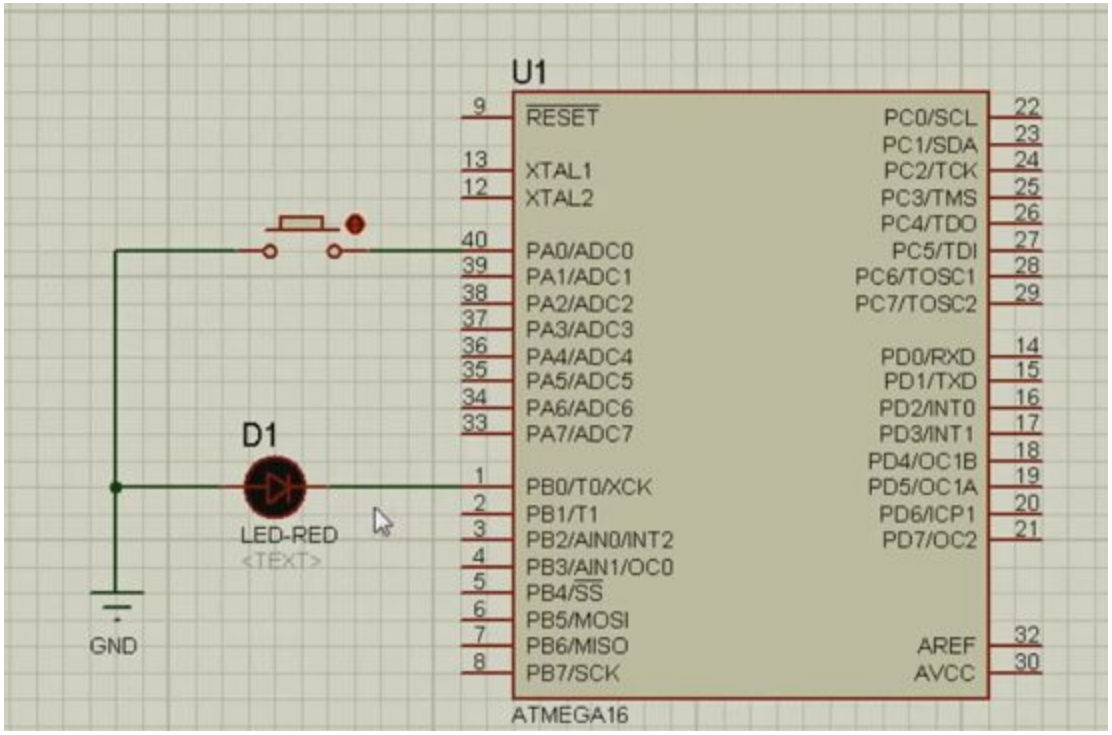
```
A:   CMP X , 0
     JLE A2
     S; тело цикла
     JMP A
A2:  …
```

```
.include "m16def.inc"
.def temp = r16
.def razr1 = r17
.def razr2 = r18
.def razr3 = r19

.dseg

.cseg
.org 0
rjmp Reset

Reset:
ldi temp, 0xff // 0b11111111 // 255
out DDRB, temp
ldi temp, 0x00 // 0b11111111 // 255
out DDRA, temp


Proga:
ldi temp, 0xff
out PORTB, temp
rcall Delay
ldi temp, 0x00
out PORTB, temp
rcall Delay
rjmp Proga

Delay:
ldi razr1, 255
ldi razr2, 255
ldi razr3, 10
PDelay:
dec razr1
brne PDelay
dec razr2
brne PDelay
dec razr3
brne PDelay
ret
```

Текст этой программы находится на диске F:jorj10.asp.

В контроллер необходимо загрузить программу jorj10.hex

Работающая программа



Алгоритм запуска: запуск isis,открываем в папке начало ассемблера программу jorj11,запускаем плеер.

3 Директивы ассемблера

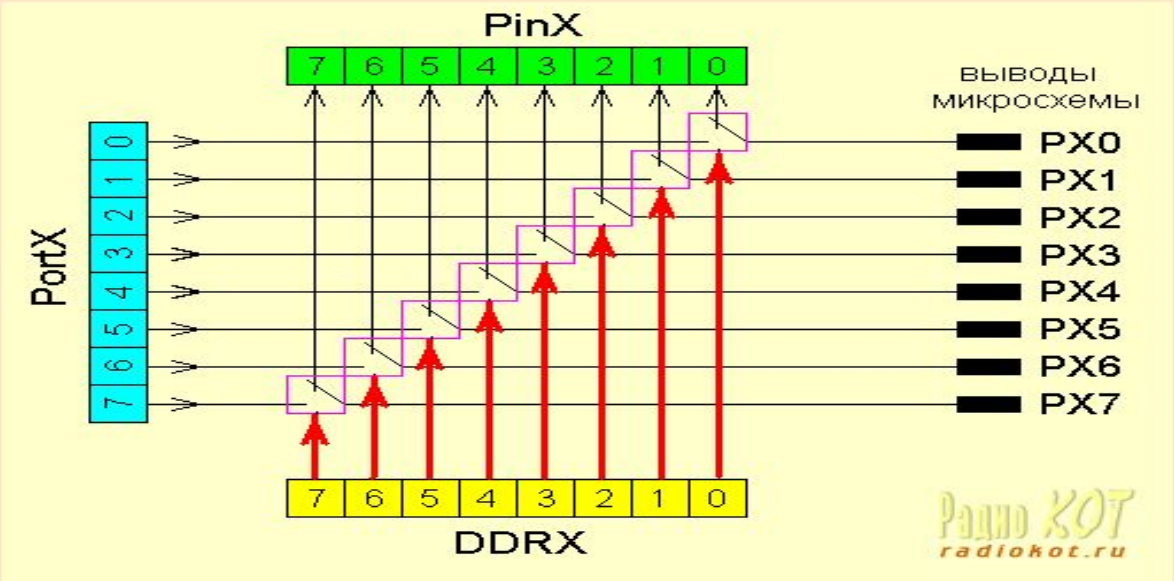
Ассемблер поддерживает множество директив. Директивы не транслируются непосредственно в коды операции. Напротив, они используются, чтобы корректировать местоположение программы в памяти, определять макрокоманды, инициализировать память и так далее. То есть это указания самому ассемблеру, а не команды микроконтроллера.
Все директивы ассемблера приведены в табл. 1.2.

Директива	Описание
BYTE	Зарезервировать байт под переменную
CSEG	Сегмент кодов
DB	Задать постоянным(и) байт(ы) в памяти
DEF	Задать символическое имя регистру
DEVICE	Задать для какого типа микроконтроллера компилировать
DSEG	Сегмент данных
DW	Задать постоянное(ые) слово(а) в памяти
EQU	Установите символ равный выражению
ESEG	Сегмент EEPROM
EXIT	Выход из файла
INCLUDE	Включить исходный код из другого файла
LIST	Включить генерацию .lst – файла
NOLIST	Выключить генерацию .lst – файла
ORG	Начальный адрес программы
SET	Установите символ равный выражению

Синтаксис всех директив следующий:
.[директива]
То есть перед директивой должна стоять точка. Иначе ассемблер воспринимает это

Порты устроены очень хитроумно. Для работы с любым портом используется три регистра:

PortX
PinX
DDRX
(где X – буква порта, например PortB, PinD и т.д.)



PortX содержит информацию, предназначенную для вывода.
PinX содержит вводимую информацию
DDRX содержит информацию о том, какой канал настроен на ввод, какой – на вывод.

То есть, DDRX определяет, грубо говоря, какая ножка микросхемы будет подключена к PinX, какая – к PortX:

0 – ввод
1 – вывод

Соответственно, если, скажем, PX3 настроен на ввод, то бесполезно писать что-либо в 3-й бит PortX, поскольку оно не будет выведено.
И наоборот, если например, PX5 настроен на вывод, то прочитав 5-й бит PinD, мы всегда обнаружим 0.
Порты – дело тонкое...

По умолчанию, все каналы порта настроены на ввод.

Нам же надо, чтобы порт B был целиком настроен на вывод. Значит, все биты DDRB должны равняться 1. То есть, в **DDRB** надо записать **'11111111'**.

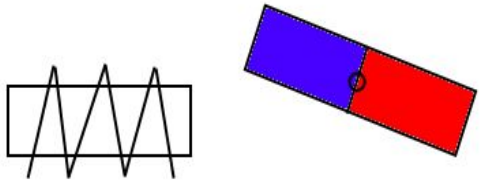
Мы не можем напрямую записать константу в регистр, не являющийся РОН. Но мы можем вывести значение из РОНа в этот регистр.

Урок 25. Управление шаговым двигателем

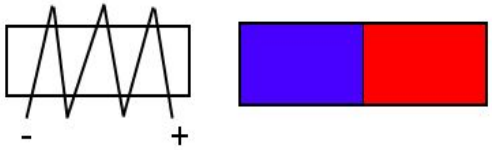
admin | 20.06.2014 | Статьи\ AVR Учебный курс | Комментарии (16)

Некоторые читатели уже давно просили рассмотреть работу шагового двигателя. Моторчик был приобретен еще полгода назад, алгоритм изучен. Хотелось совместить много всего интересного в одной статье и как обычно, чем больше планируешь, тем ниже шанс доделать устройство. В общем, я решил снова вернуться к этому вопросу и сделать статью как можно проще.

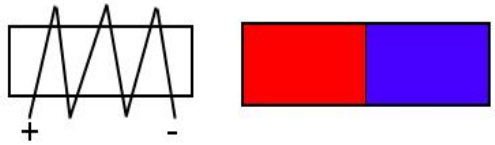
Представим себе постоянный магнит (ПМ), с осью в центре, относительно которой он может вращаться, синий — север, красный — юг. Рядом с ним электромагнит, который жестко закреплен и пока никуда не подключен, поэтому положение ПМ произвольное.



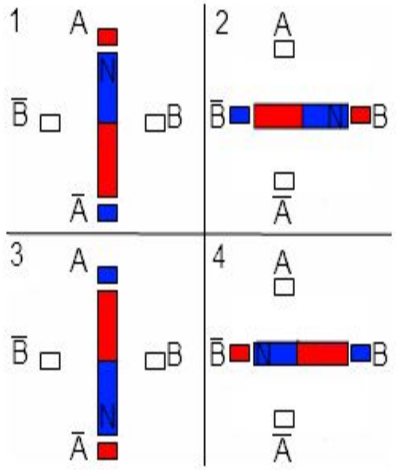
В следующий момент, подаем на начало обмотки минус, на конец плюс. Условимся, что по правилу правой руки (буравчика 😀) север у электромагнита будет слева, юг справа, поэтому ПМ развернется севером к электромагниту.



Если поменять полярность электромагнита — полюса поменяются, ПМ развернется. Таким образом, в зависимости от полярности электромагнита у вращающегося ПМ будет два устойчивых состояния, т.е. шаг будет равен 180 градусов.

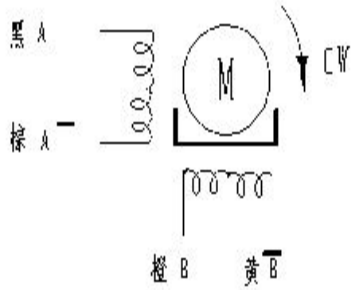


Если добавить еще один электромагнит, то будет уже четыре устойчивых состояния, т.е. шаг будет 90 градусов.



Картинки весьма условны и не отражают реальной конструкции двигателя, просто на мой взгляд они более наглядны. Двигатели с двумя независимыми обмотками, без выводов от центра обмотки называются биполярными, бывают еще униполярные и четырехобмоточные, но их пока рассматривать не будем.

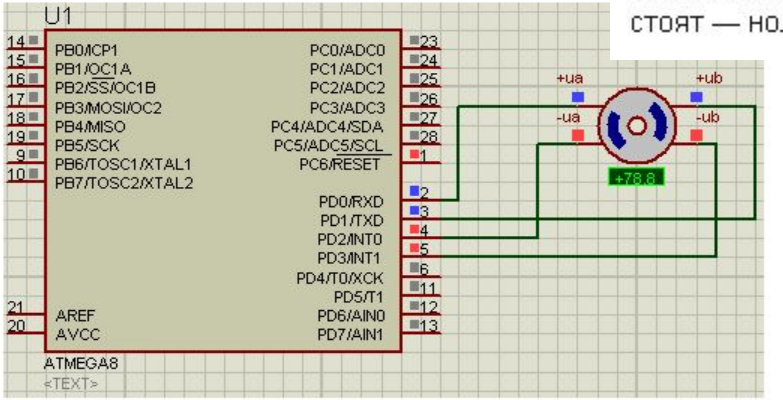
Мне достался ST-PM35-15-11C, каждый шаг — 7,5 градусов, т.е. 360/7,5=48 шагов на оборот. Номинальное напряжение 12В, сопротивление обмотки 4 Ома.



Для управления шаговым двигателем производитель предлагает такую таблицу.

```
#include <mega8.h>
#include <delay.h>

void main(void)
{
PORTD=0x00;
DDRD=0x0F;

while (1)
      {
      PORTD=0b00000011; //+a +b
      delay_ms(1000);
      PORTD=0b00000110; //+b -a
      delay_ms(1000);
      PORTD=0b00001100; //-a -b
      delay_ms(1000);
      PORTD=0b00001001; //-b +a
      delay_ms(1000);
      }
}
```

Ради теста можно покрутить это дело в протеусе.

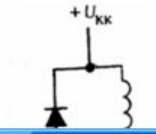


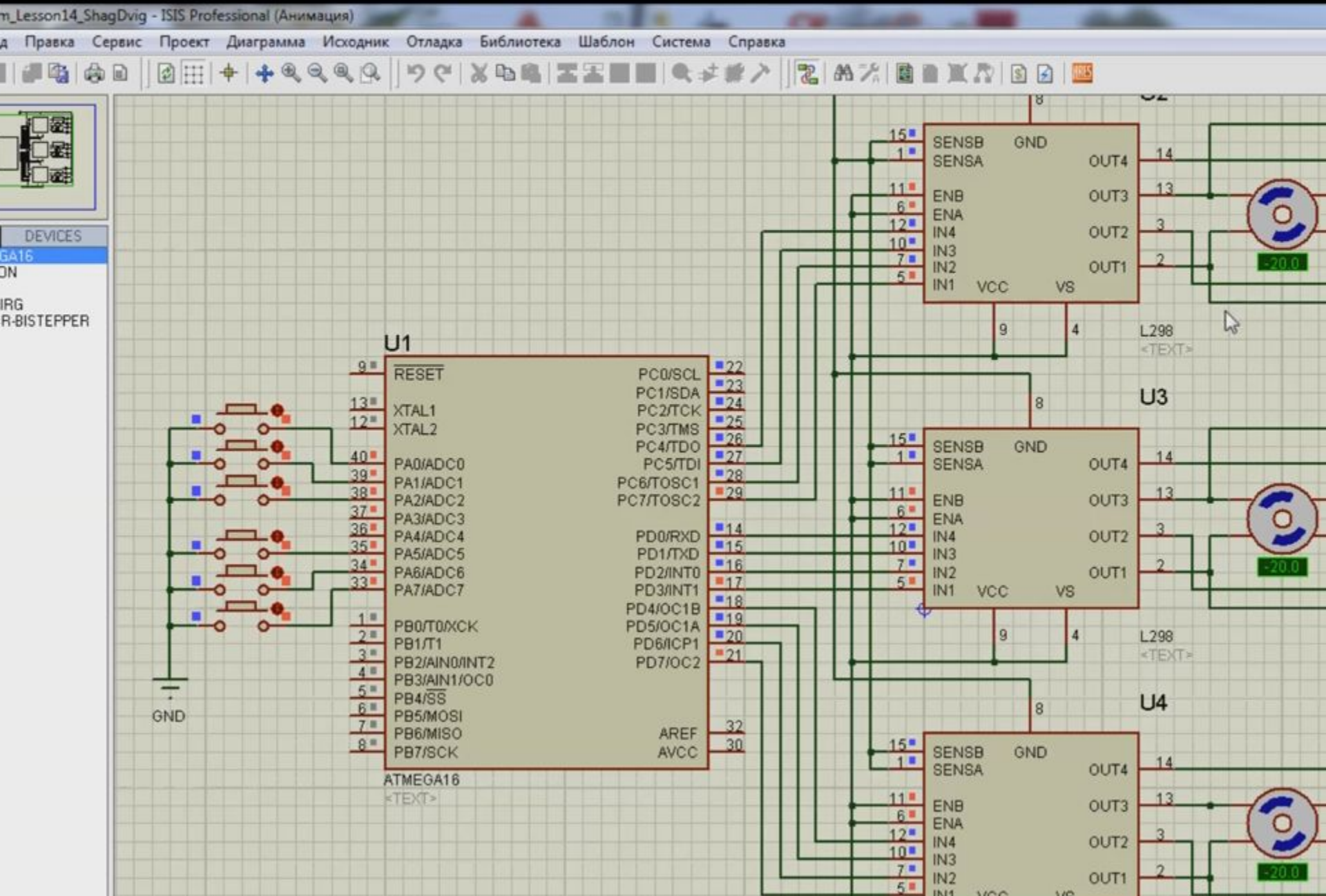
2-2 Pole Excitation Order

Lead Wire Color	Terminal Code	1	2	3	4
Black	1	-	-		
Orange	3		-	-	
Brown	2			-	-
Yellow	4	-			-

Таким образом все что нам нужно — это подавать напряжение на провода в соответствии с этой таблицей, там где стоят галочки — подать напряжение, там где не стоят — ноль.

Естественно, ножки микроконтроллера не в состоянии обеспечить достаточный ток. Поэтому можно поставить транзистор, но на каждые два шага полярность напряжения обмотки меняется на противоположную, значит одного транзистора недостаточно, поэтому на каждый вход нужно будет поставить по два транзистора(пуш пул). Кроме того, при отключении обмотки возникают броски обратного напряжения способные пробить транзистор. Поэтому понадобится защитный диод.




m_Lesson14_ShagDvig - ISIS Professional (Анимация)
Правка Сервис Проект Диаграмма Исходник Отладка Библиотека Шаблон Система Справка
DEVICES
U1
RESET
XTAL1
XTAL2
PA0/ADC0
PA1/ADC1
PA2/ADC2
PA3/ADC3
PA4/ADC4
PA5/ADC5
PA6/ADC6
PA7/ADC7
PB0/T0/XCK
PB1/T1
PB2/AIN0/INT2
PB3/AIN1/OC0
PB4/SS
PB5/MOSI
PB6/MISO
PB7/SCK
PC0/SCL
PC1/SDA
PC2/TCK
PC3/TMS
PC4/TDO
PC5/TDI
PC6/TOSC1
PC7/TOSC2
PD0/RXD
PD1/TXD
PD2/INT0
PD3/INT1
PD4/OC1B
PD5/OC1A
PD6/ICP1
PD7/OC2
AREF
AVCC
ATMEGA16
GND
SENSB
SENSA
ENB
ENA
IN4
IN3
IN2
IN1
VCC
VS
OUT4
OUT3
OUT2
OUT1
L298
U3
U4
-20.0

AVR Ассемблер. Урок 4. АЦП. AVR Assembler. Lesson 4. ADC.