

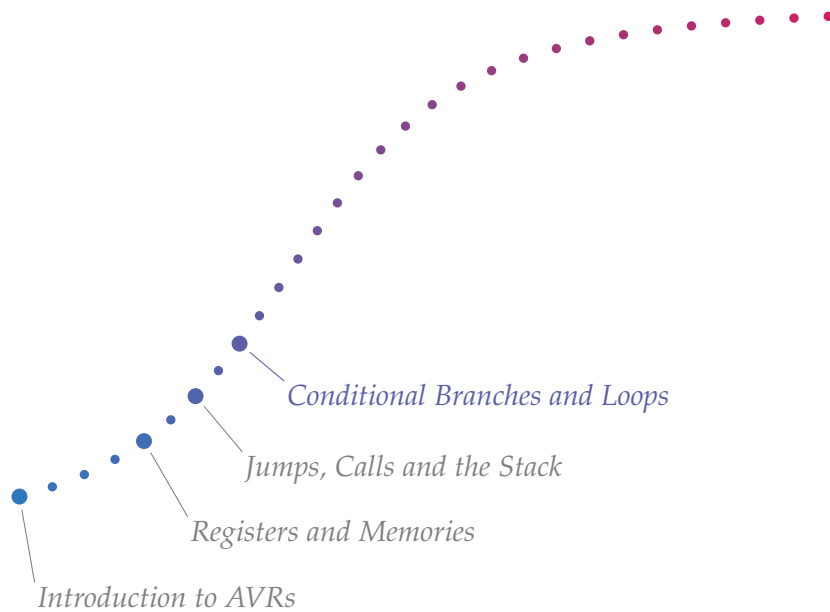
www.avrbeginners.net

Assembler Tutorial

Conditional Branches and Loops

Author: Christoph Redecker

Version: 1.0.1



This tutorial is licensed under a Creative Commons Attribution–NonCommercial–NoDerivs 3.0 Unported License:
<http://creativecommons.org/licenses/by-nc-nd/3.0/>.
Permissions beyond the scope of this license may be available at
<http://www.avrbeginners.net>.



CONDITIONAL BRANCHES AND LOOPS are closely connected in assembler. Understanding conditional branches is one key to understanding loops. Both come in a variety of ways, and some of the different branch and loop types will be discussed in this tutorial, including the loops known from the C programming language. Bare asm code is provided that can be used to construct simple `for{;;}`– or `while()`–loops.

No in–depth knowledge of C or assembler is required to understand this tutorial. It is targeted at beginners and attempts have been made to point out common pitfalls caused by the AVR instruction set or architecture. Some technical aspects are extremely simplified.

The choice of AVR registers in the examples is done in such a way that a register with the minimum of required features is used. If the `ldi` instruction is used, it is used with one of the registers `R16` to `R23`. Those below that range don't support `ldi`, those above can be used for register pair operations like `adiw`.

In short: Register numbers should be interpreted as “this or higher”.

Conditional branches

As the name implies, a conditional branch relies on some condition. From the instruction set summary, three main condition categories can be identified:

Branches that test flags in the status register The status register (`SREG`) contains information about the result of the last operation — if that result actually set any bit (a flag) in `SREG`. If that is the case is specified in the instruction set summary (and manual). The program can branch if any one flag in `SREG` is set or cleared.

Branches that test bits in general purpose registers The general purpose registers `R0` to `R31` can be tested for a specific bit being set or cleared, and the program can skip the next instruction if such a condition is true.

Branches that test bits in I/O registers Similar to the usage of general purpose registers, the state of bits I/O registers can also be used for skipping an instruction.

Note that the bit–tests in registers (general purpose and I/O) can not be used to *branch*, but merely to *skip* an instruction!

Branches that test flags in the status register

The first category, *flags in SREG*, is the largest. A great number of instructions leave information in `SREG`, and it is worth knowing what this information can be. The different branch instructions operate on this information, and the instruction set manual provides a table that summarises the branch instructions and

what the relevant flags are. To the unenlightened, this table is no help; all others don't need it because they know what the status bits mean, and what conditional branch has to be used in a particular situation.

Let's start with a short example:

```
ldi r16, 5
ldi r17, 5
cp r16, r17
breq equal
... // not executed
equal:
... // executed
```

In the above code, two registers (R16 and R17) are loaded with the same value (5) and are compared with each other, using `cp`. The following branch (`breq`, *branch if equal*) branches to the code that follows the label `equal`. If the register did not have equal values, the branch instruction would *not* branch, and the code following branch instruction would be executed.

So what is going on in `SREG`? When `cp` is used to compare two registers, the core is basically subtracting them from each other. If they are equal, the result is zero and the *Zero Flag* (Z) in `SREG` is set. The following `breq` tests this flag and, if it is set, branches. In fact, substituting `cp r16, r17` with `sub r16, r17` would result in the same behavior for the branch, but the result of the subtraction would be stored in R16. There is a complementary instruction, `brne` (*branch if not equal*), which branches when the Z flag is cleared.

There are more flags than just Z in `SREG`, there are more branch instructions that operate on them, and even more instructions that set or clear one or more flags — certainly too many to list all possibilities here, and that indeed wouldn't help much. Instead, some examples are shown here that demonstrate the usage of flags and branches.

Usage of the simple compare instruction is now extended by just altering the values loaded in the two registers, but different branches are used. First, the two unsigned tests `brlo` (*branch if lower*) and `brsh` (*branch if same or higher*):

```
ldi r16, 5
ldi r17, 7
cp r16, r17
brlo lower // branch if r16 < r17
... // not executed
lower:
... // executed
```

Example with
`cp`
`SREG:Z`
`breq`

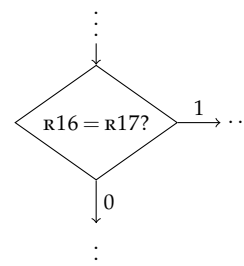


Figure 1: Effect of `cp` and `breq`

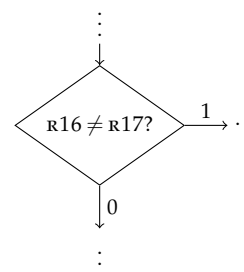


Figure 2: Effect of `cp` and `brlo` interact.

`cp`
`SREG:C`
`brlo`

In this case, `cp` sets the *Carry Flag* (C) flag in `SREG`. From the instruction set manual, `cp` section: “C: Set if the absolute value of the contents of `Rr` is larger than the absolute value of `Rd`; cleared otherwise” (in our case `R16` is `Rd` and `R17` is `Rr`). The branch instruction `brlo` tests C and, if it is set, branches.

The second “unsigned branch”, `brsh`, works just like `brlo`, but branches if the carry flag is cleared:

```
ldi r16, 5
ldi r17, 7
cp r16, r17
brsh sameOrHigher // branch if r16 >= r17
... // executed
sameOrHigher:
... // not executed
```

If `R16` had been loaded with 7 or a higher number, `cp` would have left the carry flag cleared and `brsh` would have branched. The carry flag is also useful in arithmetic operations, for example when an overflow must be detected (see the instruction manual section on `add`), or for shifting a bit out of one register and shifting it into another (see `lsl`, `rol` and similar instructions).

The two unsigned branches have the signed equivalents `brlt` (*branch if less than*) and `brge` (*branch if greater or equal*). The sign of a value is stored in the MSB, which is bit 7 for 8-bit values. The core does not know how to interpret the values stored in the registers, so the person writing the code is responsible for telling the core if a value has to be interpreted as a signed value. Not doing so can lead to an error:

```
ldi r16, -2 // stored as 254
ldi r17, 7
cp r16, r17
brsh sameOrHigher // branch if r16 >= r17 (unsigned)
... // executed this should not be executed!
sameOrHigher:
... not executed this should be executed instead!
```

The MSB of a value is set if it is negative, so if it is interpreted as an unsigned number, it is treated like a number that is higher than 128. This results in the erroneous behavior of the above code, which misinterprets a negative number as being higher than 128, and not lower than 7. Obviously, the carry flag is not made for signed tests.

The *Sign Flag* (S) is used for signed tests. It is set when the result of an operation was negative or a two’s complement overflow occurred (that is an exclusive or). `brlt` and `brge` examine just this flag, and branch appropriately:

Operand names like `Rd` and `Rr` are defined in the Instruction Set Manual.

```
cp
SREG:C
brsh
```

Example of an error with signed values

```
cp
SREG:C
brsh
```

```

ldi r16, -2
ldi r17, 7
cp r16, r17
brlt lessThan // branch if r16 < r17
...
lessThan:
... // executed

```

The usage of `brge` is equal to that of `brsh` (see above), the only difference is that it interprets signed numbers correctly:

```

ldi r16, 7
ldi r17, -2
cp r16, r17
brge greaterOrEqual // branch if r16 >= r17
... // not executed
greaterOrEqual:
... // executed

```

A special branching instruction is `cpse` (*compare, skip if equal*). It is a comparison and a branch in one: two registers are compared and, if they were equal, the next instruction is *skipped* (this works for 1- and 2-word instructions). At first glance, this seems a bit limiting, but see:

```

ldi r16, 1
ldi r17, 1
cpse r16, r17
rcall somethingBig // executed, because r16 = r17
... // executed even if r16 <> r17

```

Note that `cpse` does not set *any* `SREG` flags, unlike `cp`. Strictly spoken, it doesn't belong to the category of `SREG` flag branches, but it is related to `cp` and it doesn't fit into any of the other categories either.

Branches that test bits in general purpose registers

General purpose registers can be used to store bitfields. If that is the case, it might be useful to branch if a specific bit in the register is set or cleared. Single bits can be interesting in other situations as well, such as when the sign of a register's value needs to be known.

Two branching instructions can be used for single bits in general purpose registers. These are `sbrc` (*skip if bit in register is cleared*) and `sbrs` (*skip if bit in register is set*):

```

ldi r16, 0b10110000
sbrc r16,7
rcall BitSevenIsSet // executed

```

```

signed values
cp
SREG:S
brlt

```

```

signed values
cp
SREG:S
brge

```

`cpse`

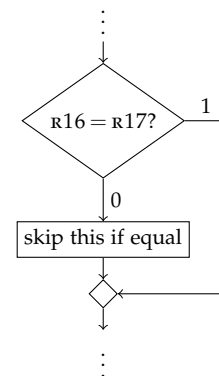


Figure 3: Flowchart for `cpse`

```

sbrc
sbrs

```

```

sbrc r16,6
rcall BitSixIsSet // not executed
sbrs r16,5
rcall BitFiveIsCleared // not executed
... // normal execution from here

```

The two instructions can be used for any general purpose register from R0 to R31.

Branches that test bits in I/O registers

Using bits in I/O registers works just like using bits in general purpose registers. This is handy if, for example, the state of an input needs to be known or while waiting for an internal peripheral to complete a data transfer (UART, TWI, ...).

Again, there are two branching instructions for these purposes, *sbic* (*skip if bit in I/O register is cleared*) and *sbis* (*skip if bit in register is set*):

```

sbi PORTA,0
sbic PORTA,0
rcall PortABitIsSet // executed
sbis PORTA,0
rcall PortABitIsSet // not executed
... // normal execution from here

```

sbic
sbis

The usage of *sbic* and *sbis* is limited to the lower 32 I/O registers. Testing single bits in the extended I/O space must be done with multiple instructions, for example:

```

lds r0, TWSR
sbrc r0, TWIE
rcall TwiInt // executed if TWI interrupt flag is set
... // normal execution from here

```

sbrc
extended I/O space

However, if I/O flags are tested in this way, special care should be taken not to interfere with interrupts service routines, which could alter the state of the tested flag before it is handled by other code.

This is more of an interrupt handling issue and not discussed in detail here.

The section about branches ends here. It's time for loops, and more branch instructions and SREG flags will be used along the way.

Loops

Loops are code structures which can (but don't have to) repeat the same piece of code multiple times. Variables can be altered before, during, or after each loop iteration. They need initialisation. They might or might not be used by the loop code. All these differences are important.

Endless loop

The simplest of all loops is the endless loop. An AVR executes instructions while it is not in any sleep mode, even if does not have anything to accomplish. Ending a program must therefore be done with an endless loop:

```
...
end:
    rjmp end
```

This loop doesn't contain any branch instruction, but most loops that actually *do* something rely on them.

An endless loop is also often used for the main part of an application; it then contains the application body which is executed over and over. The application can be initialised and also decide to perform some kind of shutdown:

```
.org 0 // the cpu jumps to here after reset
... // initialisation code, executed once
main:
    ... // body, might eventually jump to shutdown
    rjmp main
shutdown:
... // shutdown code, executed once
end:
    rjmp end
```

The cpu might also be configured to jump to the boot reset vector!

The loop body can contain code that deliberately jumps out of the loop, *breaking* it. The break statement in C does just that.

break

while-loop

It is time to add a condition to the loop. The *while-loop* executes a code block as long as some condition is true. This condition is tested *before* the body code is executed. The following example waits until the AVR's internal EEPROM is ready for writing:

```
while:
    sbic EECR, EEPE
    breq eepromIsReady
    rjmp while
eepromIsReady:
... // this code could write to the EEPROM
```

while-loop
sbic

Of course, it is possible to use a subroutine that performs a test, and then returns the result in the status register's T flag:

```
while:
    rcall testEe
    brts eepromIsReady
```

while-loop
SREG:T
brts
sbic

```

    rjmp while
eeIsReady:
... // this code could write to the EEPROM

testEe:
    clt // clear T flag
    sbic EECR, EEPE
    set // set T flag if EEPROM is ready
    ret // and return

```

Using a subroutine instead of inline code is beneficial when the subroutine is also used in other parts of code, outside the while condition. Also, the subroutine shown here is very small and simple, but overly complex compared to the code in the previous example. In a practical application, replacing a one-liner by a subroutine is a waste of both cpu time and program memory.

do...while-loop

Constructing a *do ... while-loop* from the while-loop is straight forward. In the following code, the subroutine `isLcdReady` is assumed to query the state of an LCD. If that LCD is ready for new commands, `isLcdReady` sets the carry flag and returns. If the LCD is not ready, the carry flag is cleared:

```

do:
... // possibly some more body code
    rcall isLcdReady // check if the LCD is ready
    breq do
... // send a command to the LCD

```

```

do ... while-loop
breq

```

Note that the T flag could also be used instead of the carry flag — it depends on the needs of the code that surrounds the loop.

for-loop

For-loops can be used for simple counting, for indexing arrays, iterating over lists and other things. There are many ways to implement such a loop, but in simple cases they rely on a loop variable and

- the initialisation of the loop variable,
- a condition that must be true for an iteration to be executed,
- the body code and
- a counting expression.

The following loop executes the body code ten times:

```
ldi r16, 10
for:
    ... // body code
    dec r16
    brne for
... // normal execution from here
```

```
for-loop
dec
brne
```

Special attention must be paid to the registers that are used by the body code. If R16 is altered, this may lead to errors, but can also be desired in some cases (for adding an interaction, for example). If the body code is large or confusing, the loop variable can be “protected” by pushing it onto the stack before, and popping it from the stack after the body code is executed (example follows below). However, the above code is not equivalent to

```
for(i = 10; i > 0; i--)
    // body code
```

In C, the loop condition must be true *before* the body code is executed. A small change in the assembler code makes them equivalent (in this example the loop variable is also protected by storing it on the stack):

```
ldi r16, 10
clz // zero flag is not cleared by ldi!
for:
    breq end
    push r16 // save
    ... // body code
    pop r16 // restore
    dec r16
    rjmp for
end:
... // normal execution from here
```

```
for-loop (C equivalent)
dec
breq
save and restore (on the stack)
```

Unfortunately, this is more complicated than the variant above.

It’s also possible to increment the loop variable in every iteration. If the loop variable’s value is not important (when the task is only to have a specific number of iterations), it can start with a negative value and roll over to zero. The branch at the end of the loop can then test for zero:

Imitating C in assembler is not always a good idea. Not doing so can quite often make life easier.

```
ldi r16, -10
for:
    ... // body code
    inc r16
    brne for
... // normal execution from here
```

```
for-loop
negative initialisation value
inc
brne
```

If the value *is* important, the initialisation value must make sense:

```
ldi r16, 0 // or use clr or a different value
for:
... // body code
inc r16
cpi r16, 10
brne for
... // normal execution from here
```

```
for-loop
inc
cpi
brne
```

This is almost the same as the first down-counting example, but uses an additional `cpi` in the loop and therefore is slower. On the other hand, it is now possible to initialise it with any desired value and to alter the final value as required, or to use it for special purposes in the loop body.

There are many many more ways to implement a loop, but these examples should get you started.