AVR Branches and Jumps

Table of contents

Flow of control 1
Conditional branches 2
Setting the flags first 3
Signed and unsigned comparisons 3
Unconditional jumps 4
The idioms 6
See also 8

How an AVR program's control leaves the straight line: the ten **conditional branches** and which status flag each one reads, why *greater than* needs two different instructions for signed and unsigned bytes, the three **unconditional jumps** and how far each can reach, and the assembly idioms — `if`, `if … else`, counted and conditional loops — built from them. First met in 2026-09-14-flow-of-control[1], with the compare instructions and `breq`/`brne` first met in 2026-09-03-instruction-set-architecture-part-2[2]. Procedure calls, the other way control moves, are in procedures-and-the-stack[3].

Flow of control

The **flow of control** is the sequence in which instructions actually execute. By default it is **linear**: every fetch-decode-execute cycle ends with $PC \leftarrow PC + 1$ (fetch-decode-execute-cycle[4]), so the next instruction is the next one in memory. Four kinds of instruction replace that increment with a new value:

[1] courses/csse2010/2026-09-14-flow-of-control.pdf
[2] courses/csse2010/2026-09-03-instruction-set-architecture-part-2.pdf
[3] courses/csse2010/procedures-and-the-stack.pdf
[4] courses/csse2010/fetch-decode-execute-cycle.pdf

		Decides
Branches	conditional	a status-register flag
Jumps	unconditional	nothing — always taken
Procedure calls and returns	unconditional, and they remember where they came from	procedures-and-the-stack[5]
Interrupts / traps	triggered by hardware, not by an instruction	polling-and-interrupts[6] · avr-interrupts[7]

Everything below is one instruction that writes the PC.

Conditional branches

A conditional branch **tests one bit of the status register** (avr-architecture[8]) and, if the bit has the value the instruction wants, loads the PC with the address of a label; otherwise execution falls through to the next instruction. There is one instruction per flag per value:

Branch if flag =	**C**	**N**	**V**	**S** = N V	**Z**	**T**	**H**	**I**
0	`brsh` (= `brcc`)	`brpl`	`brvc`	`brge`	`brne`	`brtc`	`brhc`	`brid`
1	`brlo` (= `brcs`)	`brmi`	`brvs`	`brlt`	`breq`	`brts`	`brhs`	`brie`

The first five columns are the ten branches counted above; the last three are the same pattern applied to the remaining flags and are in the instruction set reference. The mnemonics read as the *condition*, not the flag: `brsh` is *branch if same or higher* and happens to be *branch if carry clear*; `brge` is *branch if greater or equal* and happens to be *branch if S clear*. Learn them as conditions and the flags take care of themselves.

Every branch is relative and short. The offset is seven bits, so a conditional branch can reach only about **64 words** either side of itself. For a farther target, branch on the *opposite* condition around a `jmp`:

[5] courses/csse2010/procedures-and-the-stack.pdf
[6] courses/csse2010/polling-and-interrupts.pdf
[7] courses/csse2010/avr-interrupts.pdf
[8] courses/csse2010/avr-architecture.pdf

```
    cp   r5, r6
    brne skip          ; opposite of the condition wanted
    jmp  far_away      ; taken only when r5 == r6
skip:
```

A taken branch costs two cycles, a not-taken one; the branch itself is one 16-bit word.

Setting the flags first

A branch reads flags; something has to have written them. Usually that is a **compare** (avr-instructions[9]):

	Computes	Keeps
`cp rd, rr`	`rd - rr`	flags only
`cpi rd, K`	`rd - K`	flags only — `rd` must be r16–r31
`tst rd`	`rd AND rd`	flags only — Z and N say whether `rd` is zero or negative

but any ALU instruction sets flags, and a branch straight after an `add`, `sub`, `dec` or `andi` tests the result of *that*. `dec r17` followed by `brne loop` is the standard loop counter, and no `cp` is needed. The rule from avr-program-structure[10] holds either way: **nothing that disturbs the flag may sit between the setter and the branch.**

Signed and unsigned comparisons

Equality is the same question for signed and unsigned bytes, so `breq` and `brne` serve both. **Ordering is not**: whether `1111 1111` is bigger than `0000 0001` depends on whether it means 255 or −1, and the bits alone cannot say (binary-number-representations[11]). The `cp` sets the same flags either way; the *programmer* chooses which pair of branches to read them with, and thereby declares the type:

[9] courses/csse2010/avr-instructions.pdf
[10] courses/csse2010/avr-program-structure.pdf
[11] courses/csse2010/binary-number-representations.pdf

	Branch if `rd  rr`	Branch if `rd < rr`	Reads
unsigned	`brsh` — *same or higher*	`brlo` — *lower*	**C**: an unsigned subtraction needs a borrow exactly when the result went below zero
signed	`brge` — *greater or equal*	`brlt` — *less than*	**S** = N V: the sign of the result, corrected for overflow

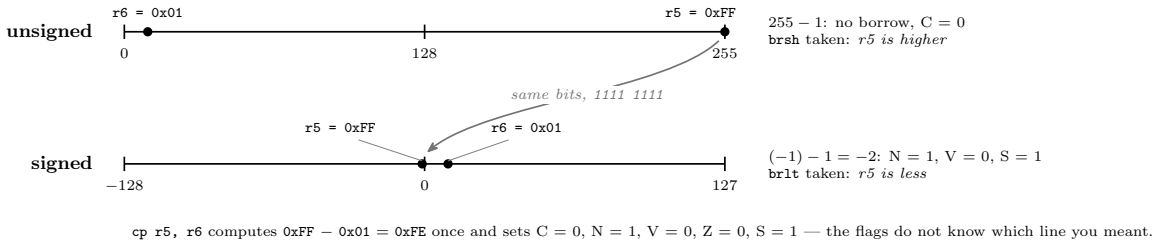


Figure 1: The same two bit patterns on the two number lines an 8-bit register can be read against. As unsigned values, `0xFF` is 255 and sits at the far right, above `0x01`; as two's complement values it is −1 and sits just left of zero, below `0x01`. One `cp r5, r6` produces one set of flags for both readings. The unsigned branches take their answer from the carry flag, which records whether the subtraction had to borrow; the signed branches take theirs from `S`, which is the sign of the difference with any overflow undone. Choosing `brsh` or `brlt` after the compare is the only place the program says which line it meant.

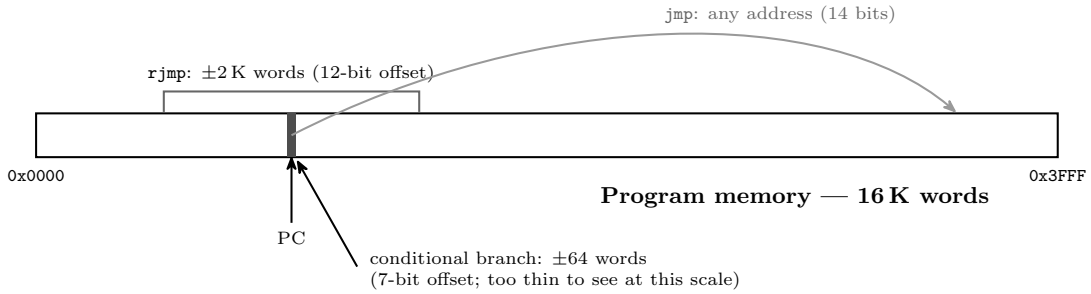
Why S and not N. `N` is the top bit of the 8-bit result. When a signed subtraction overflows — a large positive minus a large negative, say — the true difference does not fit in 8 bits, the stored result wraps, and its top bit is the *opposite* of the true sign. `V` is set in exactly that case. So $S = N \oplus V$ is the true sign whether or not overflow happened, and one flag test decides `<` correctly across the whole range; the four cases are drawn in 2026-09-14-flow-of-control[12]. In C none of this is visible: the compiler reads the variable's declared type and emits `brlo` for `unsigned char` and `brlt` for `char`, the same way it picks `lsr` or `asr` for `>>` (c-operators[13]).

Unconditional jumps

[12] courses/csse2010/2026-09-14-flow-of-control.pdf
[13] courses/csse2010/c-operators.pdf

Instruction	Target	Size	Cycles	Reach
`jmp label`	absolute address	32-bit (two words)	3	anywhere in program memory
`rjmp label`	PC + signed offset	16-bit	2	**±2 K words** (12-bit offset, −2048 to +2047)
`ijmp`	the address held in **Z**	16-bit	2	anywhere; the target is a run-time value

`jmp` carries a full program-memory address — 14 bits for the ATmega324A's 16 K words of flash — which does not fit beside an opcode in one 16-bit word, so it takes two words and an extra cycle. `rjmp` carries a 12-bit offset from the instruction's own position in one word, one cycle less, and reaches ±2 K words; in a program of a few hundred instructions every jump is within range, and it is the normal choice. `ijmp` takes the destination from `Z`, the pointer register of addressing-modes[14], so a program can *compute* where to go — a jump table, or C's `switch`. The assembler resolves labels to addresses or offsets for all three; the programmer writes the label.



out of range for a branch: branch on the opposite condition over a `jmp`. `ijmp` reaches anywhere too, from a value in Z.

Figure 2: How far each kind of jump can reach from the instruction executing it. Program memory is the ATmega324A's 16,K words; the marker is the current PC. A conditional branch carries a 7-bit offset and reaches about 64 words either way, an `rjmp` carries a 12-bit offset and reaches 2,K words either way, and a `jmp` carries the whole 14-bit address and reaches anywhere. The three are drawn to scale against the memory: the branch window is a sliver, the `rjmp` window covers a quarter of the chip, and the labs' programs never leave either. The narrower the reach, the shorter and faster the instruction.

[14] courses/csse2010/addressing-modes.pdf

The idioms

Every C control structure (c-control-flow[15]) compiles to a handful of these shapes. The branch is always on the **opposite** of the source condition, because what the branch does is *skip* the body.

`if:`

```
    cp   r5, r6
    brlt endif          ; skip the body when r5 < r6 (signed)
    ...                 ; body: runs when r5 >= r6
endif:
```

`if … else` — the *else* part is the code the branch does not skip, and it needs a `jmp` past the *if* part:

```
    cp   r5, r6
    brge then
    ...                 ; else part
    rjmp endif
then:
    ...                 ; if part
endif:
```

`while` — test at the top, jump back at the bottom:

```
loop:
    cp   r5, r6
    brsh done           ; exit when r5 >= r6 (unsigned)
    ...                 ; body
    rjmp loop
done:
```

Counted loop — count *down* to zero so the decrement sets Z for free:

```
    ldi  r17, 10        ; ten times
loop:
    ...                 ; body
    dec  r17            ; sets Z when r17 reaches 0
    brne loop
```

[15] courses/csse2010/c-control-flow.pdf

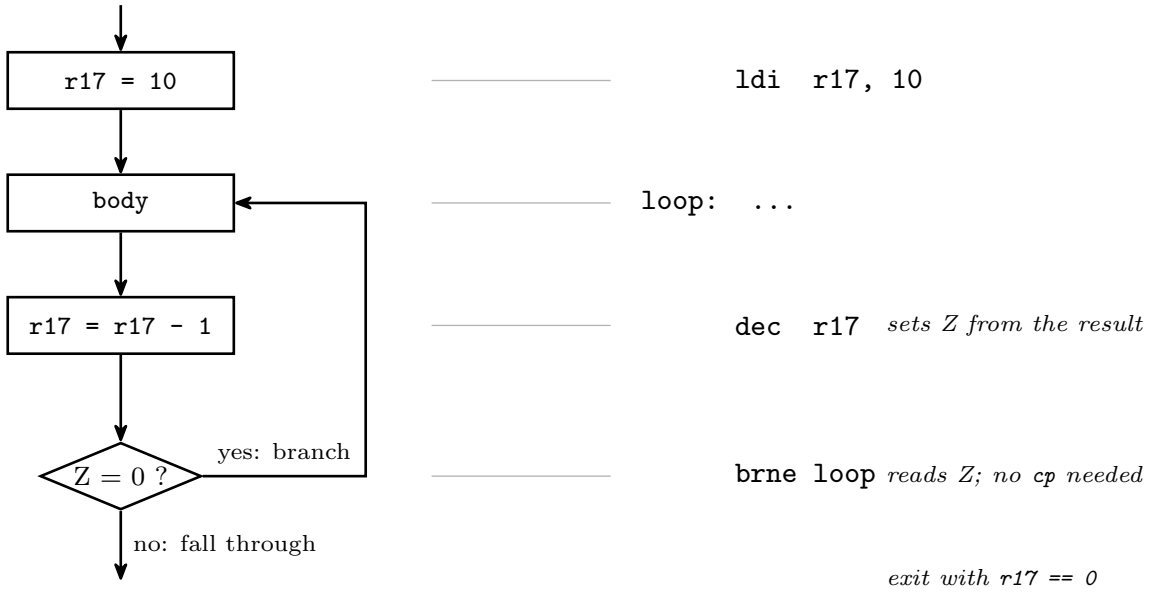


Figure 3: A counted loop, as a flowchart and as the four instructions that implement it. The counter is loaded once; the body runs; the decrement both updates the counter and sets the zero flag from the result; `brne` reads that flag and either returns to the label or falls out. Counting down rather than up saves the compare: `dec` leaves `Z = 1` exactly when the counter has reached zero, so the branch tests the arithmetic directly. The body executes ten times, with `r17` holding 10 down to 1 inside it and 0 on exit — the assembly form of `for (i = 10; i != 0; i--)`.

Picking the branch. Write the C condition, decide signed or unsigned, and take the *opposite* instruction from the table:

Source condition	Unsigned: skip with	Signed: skip with
`a == b`	`brne`	`brne`
`a != b`	`breq`	`breq`
`a < b`	`brsh`	`brge`
`a >= b`	`brlo`	`brlt`
`a > b`	swap operands: `cp b, a` then `brsh`	`cp b, a` then `brge`
`a <= b`	`cp b, a` then `brlo`	`cp b, a` then `brlt`

There is no *greater than* branch; `a > b` is `b < a`, so compare the other way round.

See also

- avr-program-structure[16] — labels, the template, and compare-then-branch drawn out
- procedures-and-the-stack[17] — `call`, `rcall`, `icall` and `ret`: the jumps that come back
- polling-and-interrupts[18] — the fourth way the PC leaves its sequence, with no instruction asking
- avr-instructions[19] — `cp`, `cpi`, `tst`, `dec`, and which flags each instruction sets
- avr-architecture[20] — the status register; $S = N \oplus V$
- c-control-flow[21] — the same shapes in C · binary-number-representations[22] — signed and unsigned bytes
- assembly-process[23] — how the assembler gives a label its address, counting each `jmp` as two words
- 2026-09-14-flow-of-control[24] — Lecture 15 · 2026-09-03-instruction-set-architecture-part-2[25] — Lecture 12

[16] courses/csse2010/avr-program-structure.pdf
[17] courses/csse2010/procedures-and-the-stack.pdf
[18] courses/csse2010/polling-and-interrupts.pdf
[19] courses/csse2010/avr-instructions.pdf
[20] courses/csse2010/avr-architecture.pdf
[21] courses/csse2010/c-control-flow.pdf
[22] courses/csse2010/binary-number-representations.pdf
[23] courses/csse2010/assembly-process.pdf
[24] courses/csse2010/2026-09-14-flow-of-control.pdf
[25] courses/csse2010/2026-09-03-instruction-set-architecture-part-2.pdf