C Operators

Table of contents

Binary operators . . . 1
Unary operators . . . 2
Bitwise is not logical . . . 3
`==` is not `=` . . . 3
Assignment shorthands . . . 4
Shifts . . . 4
Prefix and postfix `++` and `--` . . . 5
Precedence . . . 6
See also . . . 6

The operators the course uses, in groups — **arithmetic, comparison, bitwise, logical, assignment** — with the two families that look alike and are not, the shift operators and what they compile to, and the prefix/postfix rule. First met in 2026-09-07-introduction-to-embedded-c-programming[1]; the signed/unsigned shift rule is in 2026-09-10-more-c-programming-on-the-avr[2]. Types are in c-language-basics[3].

Binary operators

Group	Operator	Meaning	AVR instruction behind it
arithmetic	`+`	addition	`add`, `adc`
	`-`	subtraction	`sub`, `sbc`
	`*`	multiplication	`mul`

[1] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[2] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[3] courses/csse2010/c-language-basics.pdf

Group	Operator	Meaning	AVR instruction behind it
	`/`	division (integer, truncates toward 0)	a library routine — the AVR has no divide
	`%`	remainder (integer)	likewise
comparison	`> >= < <=`	ordering	`cp` then a branch on the flags
	`==`	equal	`cp` then `breq`
	`!=`	not equal	`cp` then `brne`
bitwise	`&`	bitwise AND	`and`, `andi`
	`|`	bitwise OR	`or`, `ori`
	`^`	bitwise XOR	`eor`
logical	`&&`	logical AND	compare-and-branch sequences
	`||`	logical OR	
assignment	`=`	assignment	a `mov`, `ld`/`st`, or `in`/`out` — whatever the two sides are

The instructions in the last column are in avr-instructions[4].

A comparison **produces a value**: 1 if true, 0 if false. So `a > b` is an ordinary expression that can be assigned, and `(a > b) + (c > d)` counts how many are true.

Unary operators

Operator	Meaning	Example	AVR
`!`	logical not	`!A` $\bar{A}$	test-and-branch
`~`	one's complement (invert every bit)	`~A`	`com`
`-`	two's complement (negate)		`neg`
`++`	increment (prefix or postfix)		`inc`, or `subi rd, -1`
`--`	decrement (prefix or postfix)		`dec`

[4] courses/csse2010/avr-instructions.pdf

Bitwise is not logical

The two pairs `&`/`&&` and `|`/`||`, and the pair `~`/`!`, are the easiest mistake in C to make and the hardest to see.

- **Bitwise** `&`, `|`, `^`, `~` work **column by column**: bit i of the result depends only on bit i of each operand. This is the eight-parallel-columns picture of bit-masking[5] and the `AND`/`OR`/`EOR`/`COM` of avr-instructions[6].
- **Logical** `&&`, `||`, `!` treat each whole operand as **true or false** (non-zero or zero) and yield **1 or 0**.

```
unsigned char a = 0x0F, b = 0xF0;

a & b      /* 0x00 - no column has a 1 in both  */
a && b     /* 1    - both are non-zero, so true */

~a         /* 0xF0 - every bit flipped           */
!a         /* 0    - a is non-zero, so "not a" is false */
```

`a & b` is zero and `a && b` is one, from the same two operands. Inside an `if`, `if (PINC & 0x01)` tests one bit, which is what is usually meant; `if (PINC && 0x01)` is true whenever *anything* on port C is high. The compiler accepts both.

== is not =

Read `a = b` as `a  b`.

- **`=` does.** `a = b` copies `b` into `a`. The arrow reading is the honest one: data flows right to left, the same *destination first* rule as `mov rd, rr` in avr-instructions[7].
- **`==` asks.** `a == b` is 1 if they are equal, 0 if not, and changes nothing.

`if (a = 0)` is legal C. It stores 0 in `a`, tests the stored value, and never takes the branch — silently.

[5] courses/csse2010/bit-masking.pdf
[6] courses/csse2010/avr-instructions.pdf
[7] courses/csse2010/avr-instructions.pdf

Assignment shorthands

`a += b` is shorthand for `a = a + b`, and **similarly** `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`. `a += 1` is the same as `a++`.

The three that matter most on the AVR are the bitwise ones, because they are the read-modify-write of a port in one line:

```
PORTA |= 0x01;      /* set bit 0, leave the rest    */
PORTA &= ~0x01;     /* clear bit 0, leave the rest  */
PORTA ^= 0x01;      /* toggle bit 0                 */
```

each a mask operation from bit-masking[8] — and the `~` in the second is the reason the mask can be written as the bit to clear rather than as its complement.

Shifts

		AVR
`a << b`	`a` shifted left by `b` bits; `0` enters at the bottom	`lsl`, `b` times
`a >> b`	`a` shifted right by `b` bits — **logical if `a` is unsigned, arithmetic if `a` is signed**	`lsr` or `asr`, `b` times

A left shift by n multiplies by 2^n; a right shift by n divides by 2^n, rounding toward $-\infty$ for signed values. Two examples: `1 << 5` is $1 \times 2^5 = 32$ and `3 << 4` is $3 \times 2^4 = 48$ — worked in bits in the shift figure of 2026-09-07-introduction-to-embedded-c-programming[9].

The type picks the instruction. In assembly the programmer chose `lsr` or `asr` and thereby declared whether the byte was unsigned or signed (2026-09-03-instruction-set-architecture-part-2[10]). In C there is one `>>`, and the declaration made the choice already:

`1 << n` is the mask with only bit n set. The AVR headers name every flag and pin as a *bit number*, so `1 << FLAG_NAME` — with `FLAG_NAME == 5` giving `00100000` — is how a mask is written from a name rather than a magic constant (c-preprocessor-and-avr-headers[11]). Combined with the shorthands above:

[8] courses/csse2010/bit-masking.pdf
[9] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[10] courses/csse2010/2026-09-03-instruction-set-architecture-part-2.pdf
[11] courses/csse2010/c-preprocessor-and-avr-headers.pdf

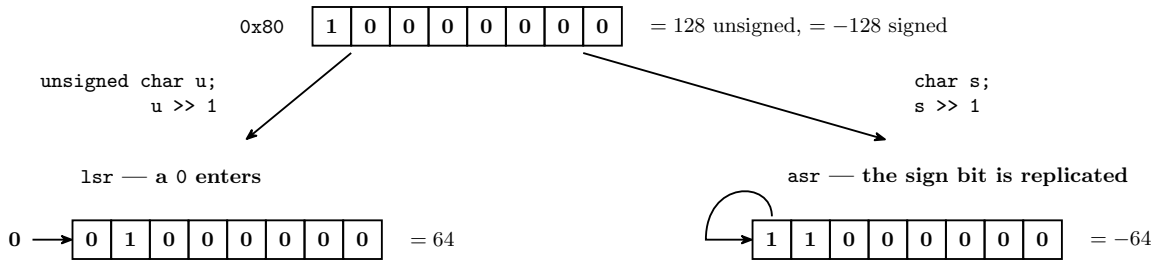


Figure 1: One operator, two instructions. The same bit pattern `10000000` is shifted right by one under two declarations. As `unsigned char` it means 128, the compiler emits `lsr`, a `0` enters at the top, and the result is 64. As `char` it means −128, the compiler emits `asr`, the sign bit is copied back into the top, and the result is −64. Both are the correct halving of what the bits were declared to mean.

```
PORTA |= (1 << 5);      /* set bit 5      */
PORTA &= ~(1 << 5);     /* clear bit 5    */
if (PINC & (1 << 3))    /* test bit 3     */
```

Prefix and postfix ++ and --

`a++` and `++a` both add one to `a`. They differ in **what the expression is worth**:

	Order	Rule
postfix `a++`	*use value, change value*	the expression is the **old** `a`; then `a` becomes `a + 1`
prefix `++a`	*change value, use value*	`a` becomes `a + 1`; the expression is the **new** `a`

Every use is two statements in disguise, and only the order differs:

```
int a,b,c,d,e;
a = 4;
b = a++;    /* b=a;   a=a+1; */   /* b=4, a=5 */
c = --a;    /* a=a-1; c=a;   */   /* a=4, c=4 */
d = ++a;    /* a=a+1; d=a;   */   /* a=5, d=5 */
e = a--;    /* e=a;   a=a-1; */   /* e=5, a=4 */
```

Final values: `a=4, b=4, c=4, d=5, e=5`.

Worked: `a = 1; b = 3; b += a++;` leaves `b = 4`, because `b += a++` is `b = b + a` with the old `a`, and only then `a = a + 1`. An answer of 5 applies the increment first, which is what `++a` does.

On its own line, `a++;` and `++a;` are indistinguishable: the value of the expression is thrown away, and only the change remains. The difference exists only when the expression is *used* — assigned, passed, indexed, as in `values[9] = values[8]++;` (c-pointers-and-arrays[12]).

Precedence

The one place precedence bites is the `SQUARE(1+2)` example in c-preprocessor-and-avr-headers[13], where `*` binding tighter than `+` turns `(1+2*1+2)` into 5. The rule the course relies on is the schoolroom one — `*`, `/`, `%` before `+`, `-` — and the practical advice is **parenthesise anything you would have to think about**. Two cases to know because the natural reading is wrong:

- `&` binds *looser* than `==`, so `x & 1 == 0` is `x & (1 == 0)`, i.e. `x & 0`. Write `(x & 1) == 0`.
- `<<` binds *looser* than `+`, so `1 << n + 1` is `1 << (n + 1)`. Write `(1 << n) + 1` if that is what is meant.

These are the two that catch AVR code in particular.

See also

- c-language-basics[14] · c-control-flow[15] · c-pointers-and-arrays[16] · c-preprocessor-and-avr-headers[17]
- bit-masking[18] — the set/clear/toggle/extract patterns the bitwise operators write
- avr-instructions[19] — the instructions each operator compiles to, including `lsl`/`lsr`/`asr`
- boolean-algebra[20] — the laws behind `&`, `|`, `^` and `~`
- floating-point-numbers[21] — integer `/` against floating point on the AVR: `PINA / 3` and `PINA * 0.334f`

[12] courses/csse2010/c-pointers-and-arrays.pdf
[13] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[14] courses/csse2010/c-language-basics.pdf
[15] courses/csse2010/c-control-flow.pdf
[16] courses/csse2010/c-pointers-and-arrays.pdf
[17] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[18] courses/csse2010/bit-masking.pdf
[19] courses/csse2010/avr-instructions.pdf
[20] courses/csse2010/boolean-algebra.pdf
[21] courses/csse2010/floating-point-numbers.pdf

- 2026-09-07-introduction-to-embedded-c-programming[22] — Lecture 13 · 2026-09-10-more-c-programming-on-the-avr[23] — Lecture 14

[22] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[23] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf