CSSE2010 — Week 7 Notes

Table of contents

Lecture 13 — Introduction to Embedded C Programming **1**

Course overview . . . 1
What the course expects in C . . . 2
C program — basic structure . . . 2
Declaring variables . . . 4
Function example . . . 5
Function definitions . . . 5
C constants . . . 6
Some C operators . . . 7
More operators — bit-shifting and assignment . . . 8
Postfix/prefix increment and decrement . . . 10
Worked: what value will `b` have? . . . 10
`if` statement . . . 11
See also . . . 12

Lecture 14 — More C: C Programming on the AVR **12**

Bit shifts in C . . . 13
Arrays in C . . . 14
Pointers . . . 14
Pointers and arrays . . . 16
Operations on pointers . . . 16
Traversing an array . . . 18
Preprocessor . . . 18
Header (`.h`) files . . . 20
Preprocessor examples . . . 20
Worked: `SQUARE(1+2)` . . . 21
AVR C macro examples . . . 21
I/O register names as variables . . . 22
Seven segment display programming example . . . 22

Seven segment display on IO board . . . 24
The program . . . 24
See also . . . 25

Reference material **26**
Addressing Modes . . . 26
The Assembly Process . . . 29
AVR Instructions . . . 44
AVR I/O Ports . . . 60
Binary Number Representations . . . 69
Bit Masking . . . 77
Boolean Algebra . . . 85
C Control Flow . . . 87
C Language Basics . . . 92
C Operators . . . 99
C Pointers and Arrays . . . 105
C Preprocessor and AVR Headers . . . 109
Compilation and Linking . . . 115
Fetch-Decode-Execute Cycle . . . 130
Floating-Point Numbers . . . 137
Instruction Set Architecture . . . 154
The I/O Board . . . 162
Memory Models . . . 170
Seven-Segment Display . . . 181

Lecture 13 — Introduction to Embedded C Programming

The first lecture of the high-level programming block: what a C program is made of, set against the assembly the course has just finished. A C program is made of five things — **variables, functions, constants, operators, control flow** — and the sections below take them in that order. The evergreen treatment is split across c-language-basics[1] (structure, variables, functions, constants), c-operators[2] and c-control-flow[3].

Course overview

The three blocks of the course, from Lecture 1:

[1] courses/csse2010/c-language-basics.pdf
[2] courses/csse2010/c-operators.pdf
[3] courses/csse2010/c-control-flow.pdf

Block	Status
High-level Programming — Embedded C programming for the ATmega324A	the current block
Low-level Programming — ISA and Assembly language programming for the ATmega324A	finished as a topic, still in use
Digital Logic Design — Hardware aspects of a microprocessor	done

Assembly is finished as a topic but not left behind, because every C construct here is explained by what it compiles down to.

What the course expects in C

The expectation in this course is to:

- be able to write small C programs and functions from scratch
- understand the meaning of C programs
- be able to write/modify C programs

Lectures can't teach programming. It takes practice.

- There will be some self-learning examples on **EdForum**
- Coding examples in **Labs 11–16** will be using embedded C

From Lab 11 on, the lab work is C, and the assembly of Labs 9 and 10 (week5-lab-avr-assembly-one[4], week6-lab-avr-io-ports[5]) is what it sits on.

C program — basic structure

- There must be a function **called `main`** — this function is executed when the program starts
- Blocks of code enclosed by braces `{ }`
- C statements must end with a **semicolon** `;`
- C statements are **case sensitive** — `variable` is not the same as `Variable`
- Comments are within `/* … */` or from `//` to end of line

[4] courses/csse2010/week5-lab-avr-assembly-one.pdf
[5] courses/csse2010/week6-lab-avr-io-ports.pdf

The program sits in a file such as `name.c`. `main()` is at the top with its `args` in the brackets, then a brace and a column of statement lines, each ending in a semicolon. `C = A + B` is an expression, the thing C lets you write that assembly does not. There are two routes into program memory, drawn in Figure 1.

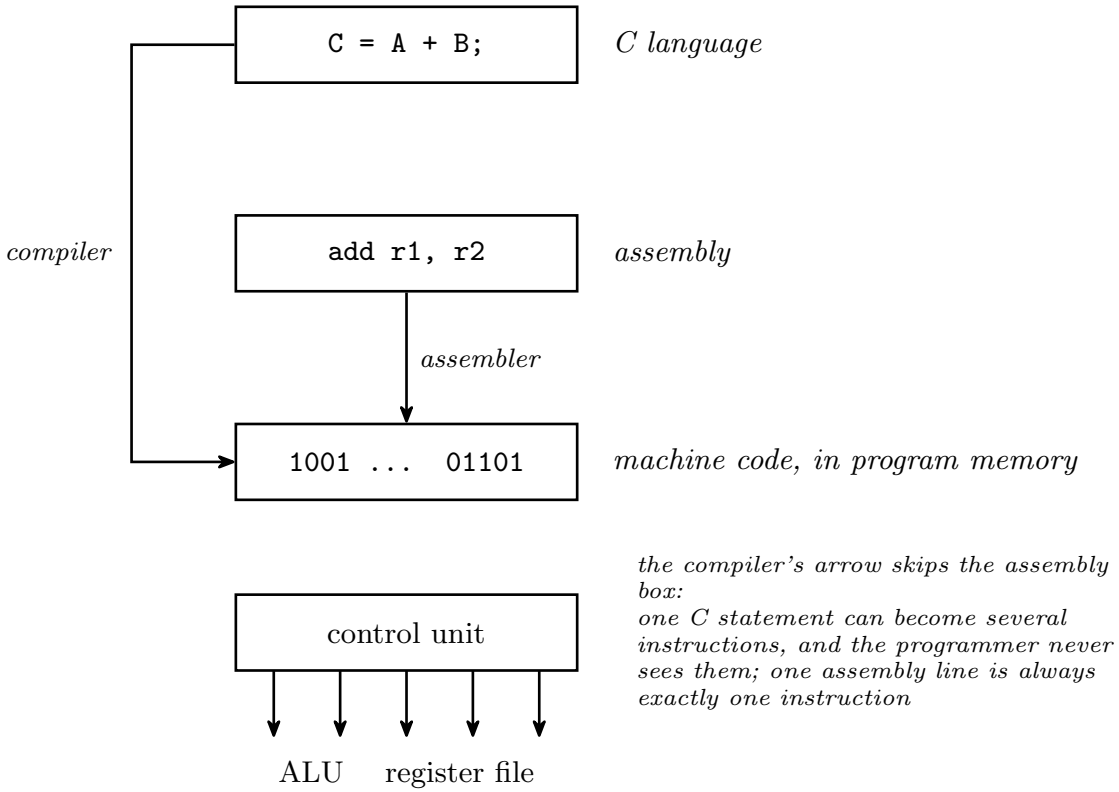


Figure 1: What happens to one line of C. Two routes converge on program memory: the long *compiler* arrow runs from the C statement straight to the machine code, and a short *assembler* arrow drops the assembly line into the same place. Under the machine code sits the control unit, fanning out to the ALU and register file. The bit string `1001 ... 01101` is a placeholder, not a real encoding.

Figure 1 is the frame for the whole block. The bottom three boxes are Lectures 9–12 (fetch-decode-execute-cycle[6], avr-instructions[7]); C adds one more layer on top, and a **compiler** to cross it. The compiler's arrow goes *past* the assembly box straight into program memory, because the crossing is no longer one-to-one — a single `C = A + B;` may become several instructions, and the programmer never chooses which registers or even sees the assembly. The asymmetry in short: *assembly* $\neq$ *machine code, 1:1; C* $\rightarrow$ *machine code* — see 2026-09-10-more-

[6] courses/csse2010/fetch-decode-execute-cycle.pdf
[7] courses/csse2010/avr-instructions.pdf

c-programming-on-the-avr[8].

The five things

variables
functions
constants
operators
control flow, e.g. `if … else`

The area-of-a-circle formula $A = \pi r^2$ is the running example for the first three: A and r are variables, π is a constant, and the multiplication and squaring are operators. Five sections below carry the matching number — variables, the function example, constants, operators and `if`.

Declaring variables

Declaration: `type-name variable-name, variable-name …;` — the comma separates names of the same type, the semicolon ends the declaration.

Example	Type	Size and range
`char c;`	Single byte (two's complement)	**8 bit, −128 … 127**
`unsigned char x;`	Single byte (unsigned)	**0 … 255**
`int day;`	Integers (size is machine dependent). Integers can be unsigned or signed (two's complement).	**AVR: 16 bit**
`unsigned int count;`	*(same)*	

Two things to remember:

- **`char` is a number.** It is the same 8-bit two's complement integer the whole hardware block was about (binary-number-representations[9]); the name just says it is *big enough* for a character. `unsigned char` is the 0–255 reading of the same byte.

[8] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[9] courses/csse2010/binary-number-representations.pdf

- **`int` has no fixed size.** On the AVR it is **16 bits** — two of the machine's 8-bit cells, which is why Lecture 14's pointer arithmetic steps by two. On a desktop it is usually 32. Code that assumes one or the other breaks when moved.

`char`, `int`, `float` and `double` are among the data types supported by C. `float` and `double` are the floating-point types the AVR has no hardware for (instruction-set-architecture[10] has the tables), so every floating-point operation is a library call (floating-point-numbers[11]).

Any value can be treated as a boolean — zero means false, non-zero means true. In short: `== 0 FALSE`, `0 TRUE`. There is no separate boolean type in the C the course uses. The `if` statement below comes back to this.

Function example

```
/*
  Return the average of two integers
  (result will be rounded towards 0)
*/
int average(int a, int b) {
    int avge;

    avge = (a+b)/2;

    return avge;
}
```

`int` at the front is the **return type**, `average` is the **name**, and `int a, int b` are the **args**. The three lines inside the braces are the **statements**, each ending in a semicolon. In `avge = (a+b)/2;` the `=` is **an assignment operator**, and `(a+b)`, `(a+b)/2` and the whole line are **expressions; the outer expression is a statement**. `return avge;` is where **the function returns a result when finished**.

The comment says *rounded towards 0*. $\frac{1+2}{2} = 1.5$, but `average(1, 2)` is `3/2`, integer division, and the `.5` is simply dropped, giving `1`. Division of two `int`s in C is integer division; the fraction never exists to be rounded.

Function definitions

The template:

[10] courses/csse2010/instruction-set-architecture.pdf
[11] courses/csse2010/floating-point-numbers.pdf

```
return-type function-name (arg-decl)
{
    variable-declarations;
    …
    statements;
}
```

Part	Meaning
`return-type`	What type is returned by the function, e.g. `int` or `void`
`function-name`	Name of the function, e.g. `main`
`arg-decl`	Argument (parameter) declaration, e.g. `int a, float b`
`variable-declarations`	Variables used **only** within the function
`statements`	The code which actually does stuff

`void` is the return type for a function that returns nothing. The **only within the function** is scope: `avge` in the example exists while `average` runs and is gone afterwards — it has no address anyone outside can name.

C constants

Character constants — use single quotes, e.g. `'a'`, `'b'`, `'1'` etc. Some special characters are **backslash escaped**:

`'\n'`	newline
`'\''`	single quote
`'\t'`	tab
`'\\'`	backslash

String constants — use double quotes (can include backslash escapes), e.g. `"abc \n \" hello\t"`.

Integer constants

Base	Examples	
Decimal	`3`, `-27`, `65535`, `+5`	
Hexadecimal (leading `0x`)	`0x5F`, `0xFFFF`, `0xDEADBEEF`	
Octal (leading `0`)	`0377`	= 255 decimal

Floating point constants — include a decimal point (`.`) and/or `e` for exponent: `3.1416`, `-7.`, `6.02e23`, `-5.2e-2`. **`7` is an integer, `7.` is floating point.**

In ASCII, **A–Z is 65–90, a–z is 97–122, 0–9 is 48–57**. That is what makes `'1'` a *character* constant and not the number one: `'1'` is 49. In $A = \pi r^2$, π is the constant in the running example.

The octal row is the trap to remember. **A leading zero changes the base.** `0377` is 255, not 377, and `010` is 8. Nobody writes octal on purpose in this course, but a zero-padded decimal will be read as it.

Some C operators

Binary operators, by group:

		Group
`+`	addition	*arithmetic* — e.g. `A + B`, `A - B`
`-`	subtraction	
`*`	multiplication	
`/`	division	
`%`	remainder (integer)	
`>`	greater than	*comparison*
`>=`	greater than or equal	
`==`	equal	
`!=`	not equal	
`<`	less than	
`<=`	less than or equals	
`&`	bitwise AND	*logical* — this one group takes in both the bitwise and the logical rows
`|`	bitwise OR	
`^`	bitwise XOR	
`&&`	logical AND	
`||`	logical OR	
`=`	assignment	

Unary operators

		Example
`!`	logical not	`!A` $\quad \bar{A}$

		Example
`~`	one's complement (invert)	`~A`
`-`	two's complement (negate)	
`++`	increment (prefix or postfix)	
`--`	decrement (prefix or postfix)	

The pair to keep apart is `==` and `=`: **`==` asks, `=` does.** `=` copies the right side into the left, so `a = b` is best read as `a  b`. Writing `=` where `==` was meant is legal C — it assigns, then tests the assigned value — and is the classic silent bug.

Two more pairs look alike and are not: `&` against `&&` and `|` against `||`. The single forms are the eight-parallel-columns operations of bit-masking[12] — `AND`, `OR`, `EOR` in avr-instructions[13] — and the double forms are the true/false connectives of boolean-algebra[14]. `~` is the `COM` instruction and unary `-` is `NEG`. The full reference is c-operators[15].

More operators — bit-shifting and assignment

`a << b`	means `a` shifted left by `b` bits
`a >> b`	means `a` shifted right by `b` bits
`a = b`	means `a` is assigned the value of `b`, i.e. `a  b`
`a += b`	is shorthand for `a = a + b`

Similarly `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`.

Examples, with the shifts worked in bits:

Example	Worked
`1 << 5` is $1 \times 2^5 = 32$	`000001 << 5  100000` 32, with five zeros now to the right of the `1`
`3 << 4` is $3 \times 2^4 = 48$	`0000011 << 4  0110000` 48, with four zeros now to the right of the `11`
`a += 1` same as `a++`	`a = a + 1`, `a++`

`1 << n` is the idiom that matters: it is **the mask with only bit n set**, and Lecture 14 connects it to named flags. This is `lsl` from 2026-09-03-instruction-set-architecture-part-2[16]

[12] courses/csse2010/bit-masking.pdf
[13] courses/csse2010/avr-instructions.pdf
[14] courses/csse2010/boolean-algebra.pdf
[15] courses/csse2010/c-operators.pdf
[16] courses/csse2010/2026-09-03-instruction-set-architecture-part-2.pdf

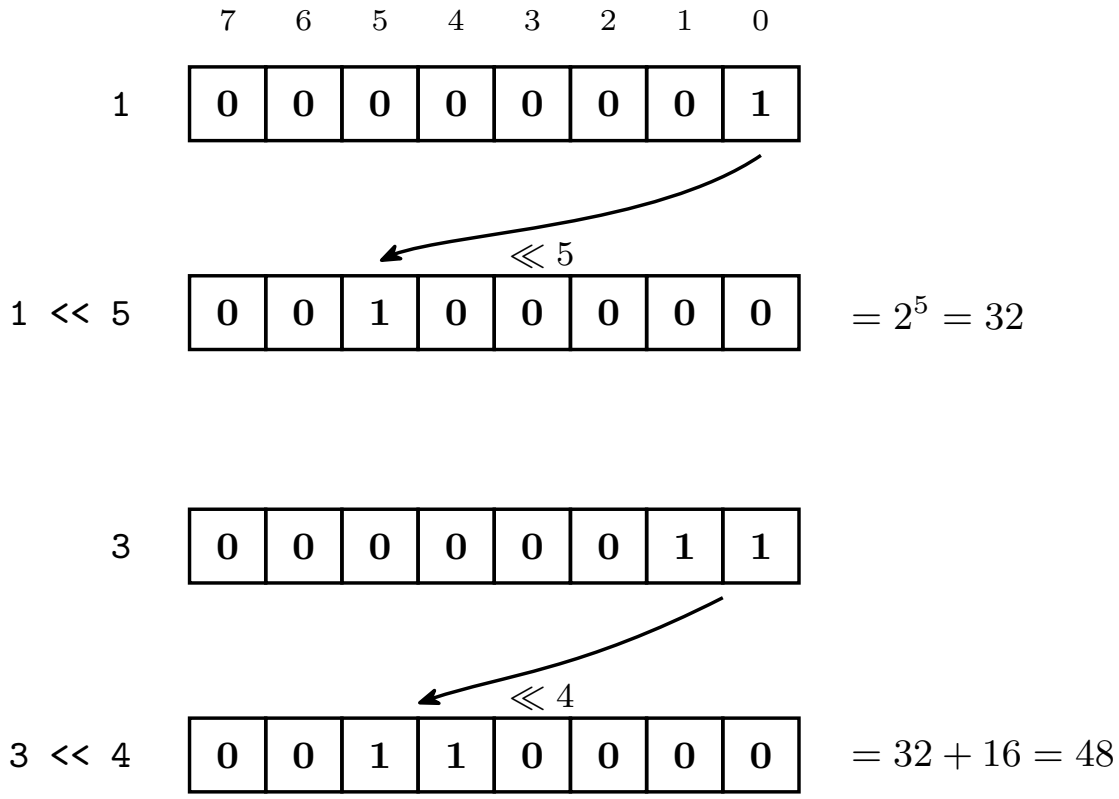


Figure 2: The two shift examples worked in eight bits. The `1` in `1 << 5` walks five places to the left and lands at bit 5, which is $2^5 = 32$; the `11` of `3 << 4` walks four places and lands at bits 5 and 4, $32 + 16 = 48$. The curved arrows span the zeros the 1s hop over.

with the count written as a number instead of repeated n times — and the same $\times 2$-per-place reading applies.

Postfix/prefix increment and decrement

An example, with the running value of `a` for each line:

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

Line	`a` before	What happens	`a` after
`b = a++;`	4	`b=4`, then `a=5`	5
`c = --a;`	5	`a=4`, then `c=4`	4
`d = ++a;`	4	`a=5`, then `d=5`	5
`e = a--;`	5	`e=5`, then `a=4`	4

After these statements, values are `a=4, b=4, c=4, d=5, e=5`.

The rule:

- **Postfix** (`a++`) — use value, change value. The change happens **after** the value is used.
- **Prefix** (`++a`) — change value, use value. The change happens **before** the value is used.

The comments in the code are the whole definition: each line is two statements, and the only question is which comes first. The two-statement expansion is also why `values[9] = values[8]++;` in Lecture 14's array example copies the *old* `values[8]`.

Worked: what value will `b` have?

```
a = 1;
b = 3;
b += a++;
```

The choices are 1, 2, 3, 4, 5, 6 or none of the above.

`b` is 4. The line is `b = b + a++`, and it splits into two statements exactly as above: `b = b + a` `b = 4`, `a = a + 1` `a = 2`. Postfix means the `+ 1` on `a` lands *after* `a`'s old value has already gone into the sum. The answer 5 comes from applying the increment first, which is what `++a` would have done.

`if` statement

The `if` statement is **control flow**, the fifth and last of the five things.

`if (expression) stmt else stmt` — the first `stmt` runs when the expression is `TRUE` and the `else stmt` runs when it is `FALSE`.

- The **`else` clause is optional**.
- **Expressions:** C interprets any 0 value as false, anything else as true.
  - `if(a)` means `if(a != 0)` — *0 TRUE*
  - `if(!a)` means `if(a == 0)` — *== 0 FALSE*
- **`stmt` can be replaced by multiple statements enclosed in braces `{ }`.**

There are three kinds of control flow, drawn in Figure 3:

sequential	*selection (decision)*	*repetition*
statements, one after another	`if … else`	`for; while`

Selection is the `if` statement above. Repetition arrives in Lecture 14 as the `for` loops that walk an array and the `while(1)` that every embedded program runs inside — see c-control-flow[17] for all three together.

The `if(a)` shorthand and the boolean rule under variables are the same fact stated twice, and they connect straight back to hardware: the compiled test is a compare against zero followed by a conditional branch, `cp`/`tst` then `breq`/`brne` in avr-instructions[18] — the **Z** flag is what "is it zero" means to the machine.

[17] courses/csse2010/c-control-flow.pdf
[18] courses/csse2010/avr-instructions.pdf

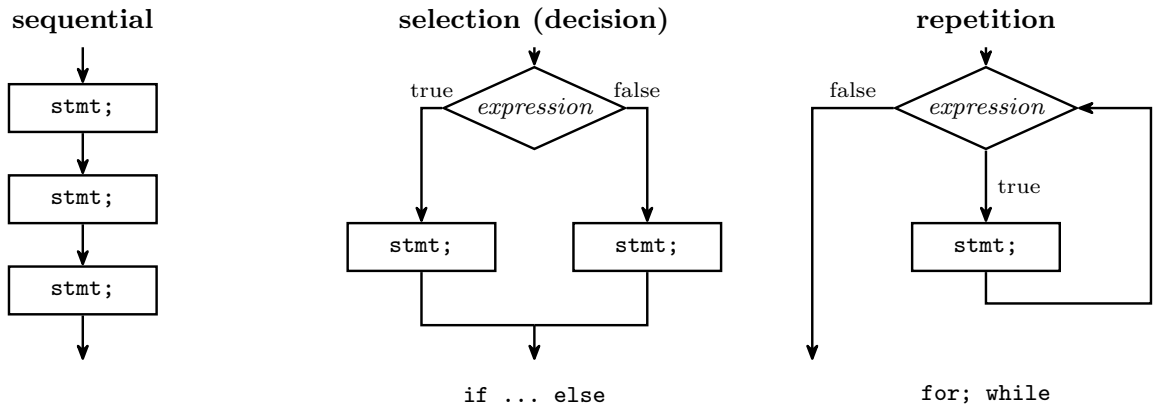


Figure 3: The three shapes a C program's control can take. Left, the default: statements run one after another. Middle, selection: the expression is tested and exactly one of the two branches runs. Right, repetition: the body runs and control returns to the test until the test is false.

See also

- c-language-basics[19] — structure, variables and types, functions, constants
- c-operators[20] — every operator here, with the bitwise/logical split and the shift rules
- c-control-flow[21] — `if`, `while`, `for`, and the truthiness rule
- 2026-09-10-more-c-programming-on-the-avr[22] — Lecture 14, which puts this on the AVR
- compilation-and-linking[23] — what lies behind the compiler's arrow: object files, the linker and libraries
- 2026-09-03-instruction-set-architecture-part-2[24] — the shifts and 16-bit arithmetic this lecture's operators compile to
- bit-masking[25] · avr-instructions[26] · binary-number-representations[27]

[19] courses/csse2010/c-language-basics.pdf
[20] courses/csse2010/c-operators.pdf
[21] courses/csse2010/c-control-flow.pdf
[22] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[23] courses/csse2010/compilation-and-linking.pdf
[24] courses/csse2010/2026-09-03-instruction-set-architecture-part-2.pdf
[25] courses/csse2010/bit-masking.pdf
[26] courses/csse2010/avr-instructions.pdf
[27] courses/csse2010/binary-number-representations.pdf

Lecture 14 — More C: C Programming on the AVR

This is where C meets the hardware: arrays, pointers and what they look like in the AVR's data memory; the preprocessor and the header file that turns `DDRA` into a memory address; and a complete program that drives the IO board's seven-segment display. The evergreen notes are c-pointers-and-arrays[28], c-preprocessor-and-avr-headers[29] and seven-segment-display[30]; the shift rules go into c-operators[31].

Bit shifts in C

The three assembly shifts are `lsl`, `lsr` and `asr`, from Lecture 12 (2026-09-03-instruction-set-architecture-part-2[32]):

Instruction	Movement of the bits	Effect
`lsl`	`C` `[ register ]` `0`	**× 2**
`lsr`	`0` `[ register ]` `C`	**÷ 2 unsigned**
`asr`	the top bit looped back into itself, `C`	**÷ 2 signed**

The C equivalents:

`a << b`	`a` shifted left by `b` bits
`a >> b`	`a` shifted right by `b` bits — **a logical shift right if `a` is unsigned, an arithmetic shift right if `a` is signed**

In assembly the programmer chooses `lsr` or `asr` by hand and thereby declares what the bits mean. **In C the declaration is the variable's type**, and the compiler picks the instruction: `unsigned char` gets `lsr`, `char` gets `asr`. There is one `>>` operator and two instructions behind it. (Strictly, the C standard leaves `>>` on a negative signed value implementation-defined; avr-gcc chooses the arithmetic shift, and treats plain `char` as signed, which is what makes the rule hold on this toolchain.)

Two more points:

[28] courses/csse2010/c-pointers-and-arrays.pdf
[29] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[30] courses/csse2010/seven-segment-display.pdf
[31] courses/csse2010/c-operators.pdf
[32] courses/csse2010/2026-09-03-instruction-set-architecture-part-2.pdf

- `1 << 5` **is** `00100000`, a single 1 in bit 5. This is the idiom the AVR headers are built on: a flag is a *bit number*, say `FLAG_NAME == 5`, and `1 << FLAG_NAME` is the mask for it. The worked bits are drawn in the shift figure of 2026-09-07-introduction-to-embedded-c-programming[33]; the idiom is in bit-masking[34].
- **Assembly to machine code is** `1:1`: one assembly line is one machine instruction. C also ends up as machine code, but by a route where one statement may become several instructions (the C-to-machine-code figure of 2026-09-07-introduction-to-embedded-c-programming[35]).

Arrays in C

An array is declared as `type variable_name[size];`

- `char message[16];`
- `int values[10];`
- `float BT_26[365];` — 365 entries, presumably one per day of a year.

An element is accessed as `variable_name[index]`, where **index = 0 ... size−1**. This is **zero-based indexing**.

- `message[0] = 'c';`
- `values[9] = values[8]++;`
- `BT_26[0] = 36.3;` down to `BT_26[364] = 33.8;` — the first and last legal indices of a 365-element array.

`BT_26[365]` **does not exist**, and C will not stop you writing to it: it will write to whatever happens to sit after the array in memory. The memory layouts below make that concrete.

`values[9] = values[8]++;` is postfix, so it is two statements: `values[9] = values[8];` and then `values[8] = values[8] + 1;` — the copy gets the **old** value (2026-09-07-introduction-to-embedded-c-programming[36]).

Pointers

C has pointers. A pointer is declared as `type * variable_name;`, which makes `variable_name` a pointer to something of the given `type`. **A pointer variable stores a memory address.**

For example:

[33] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[34] courses/csse2010/bit-masking.pdf
[35] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[36] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf

```c
char a, b;
char *ptr;      /* can write these on one line: char a,b,*ptr; */

ptr = &a;       /* & is address-of operator - creates a pointer  */
b = *ptr;       /* * is indirection operator - returns value pointed to */
```

Figure 4 works it in data memory. Say `a = 8`, with `a` at address **400**, `b` at address **434** and `ptr` at address **576**. After the two assignments, `ptr == 400` and `b == 8`.

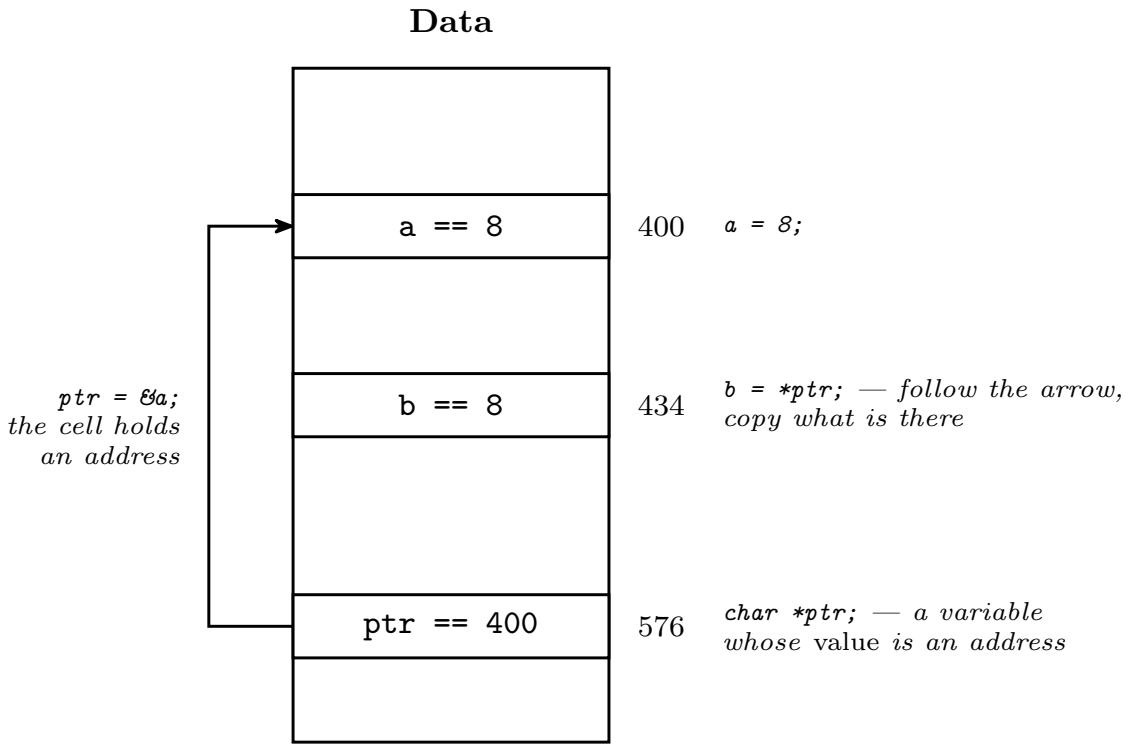


Figure 4: The pointer example in data memory. Three variables, three cells: `a` holds 8, `b` holds 8 (copied through the pointer), and `ptr` holds 400 — which is not a value but *the address of* `a`, hence the arrow. The decimal addresses 400, 434 and 576 are only examples; where the compiler actually places variables is its own business. The lowest address is at the top and addresses are written down the right, as in the course's memory maps, but in decimal here.

The two operators are inverses, and Figure 4 is the way to keep them straight. **`&a` goes from a variable to its address** — the number 400. **`*ptr` goes from an address back to the value stored there** — follow the arrow, read the cell. `b = *ptr;` therefore lands 8 in `b`. `ptr`

is itself an ordinary variable with its own cell at 576; a pointer is just a variable whose contents happen to be an address.

This is the *indirect* addressing mode of addressing-modes[37] made into a language feature: `ptr` plays the part of `X`, and `b = *ptr;` compiles to something like `ld r_b, X`.

Pointers and arrays

An array name can be treated as a pointer to the first element, i.e. the address of the first element.

```
int a[10];
int *ptr;

/* following statements are same */
ptr = a;
ptr = &a[0];
```

The array is a contiguous run of cells in data memory, and its name means the address where that run starts: **300**, say.

Operations on pointers

Addition and subtraction on pointers work in multiples of the size of the object being pointed to.

```
int a[10];
int *ptr;

/* following statements are same */
ptr = a+5;
ptr = &a[5];
```

An `int` on the AVR is **16 bits**, and a data memory cell is **8 bits** wide. With `a` at 300, `a + 5` is not `300 + 5 = 305` but `300 + 2·5 = 310`. In memory `a[0]` lies across addresses 300 and 301, `a[1]` across 302 and 303, and so on down to `a[9]`. `a` is 300, and so is `ptr` after `ptr = a`. Figure 5 draws it out.

The compiler does the multiplication. `a + 5` means *five elements on*, and since an `int` is two cells on this machine that is ten cells, so 310. Had the array been `char`, `a + 5` would be 305 — the same source expression, a different address, decided entirely by the declared type. This is

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

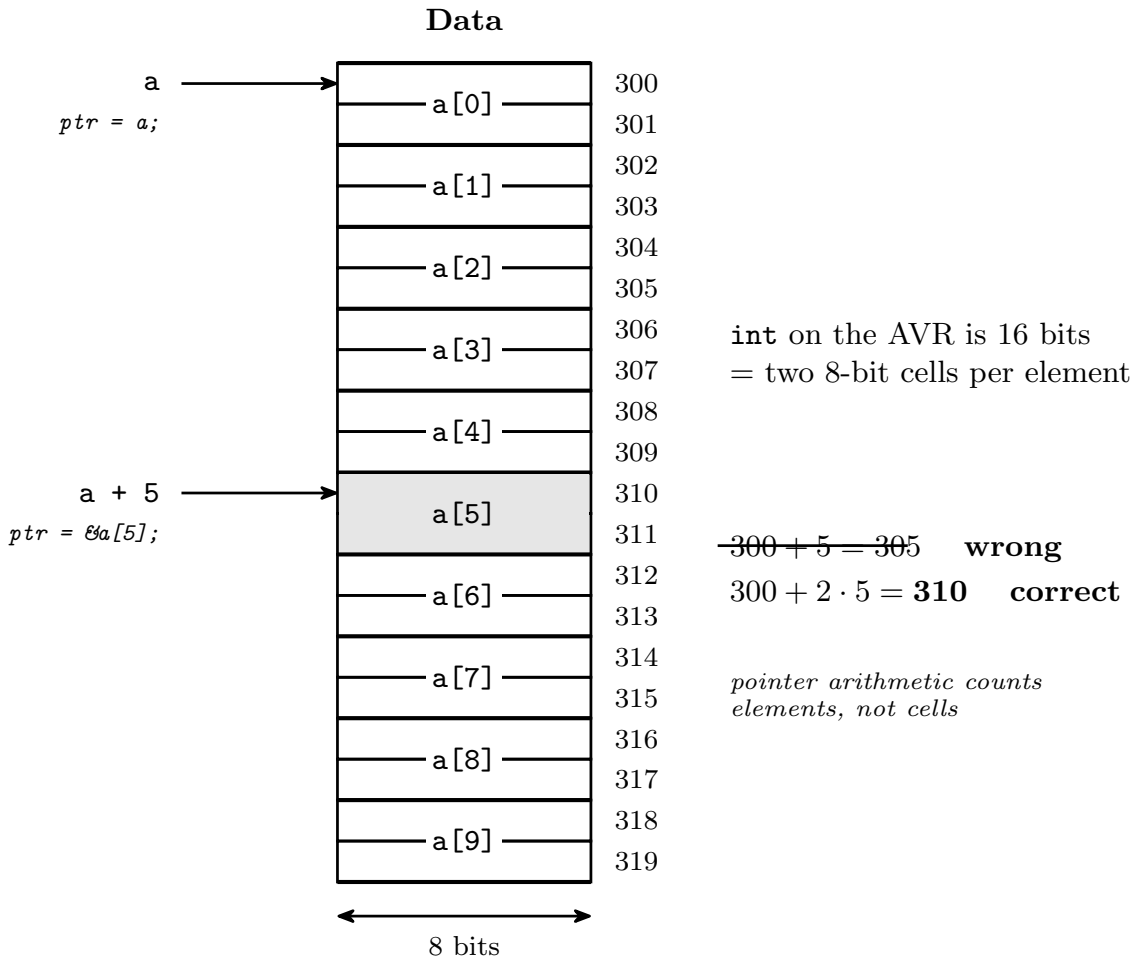


Figure 5: `int a[10]` in the AVR's data memory. Each cell is 8 bits wide; an `int` on the AVR is 16 bits, so every element takes *two* cells and the array runs from 300 to 319. That is why `a + 5` is address 310, not 305: pointer arithmetic counts in elements, and the compiler multiplies by the element size. The shaded element is `a[5]`, the one the arithmetic lands on.

also why the rule says *the size of the object being pointed to* rather than a number: the number is machine-dependent, exactly as the size of `int` itself is (Lecture 13). See memory-models[38] for the AVR's 8-bit data cells, which are what make the two-cells-per-`int` happen.

Traversing an array

Two ways of clearing an array, each filling its cells with 0 one after another.

Using an index:

```
int a[10];
int index;
for(index=0; index<10; index++) {
    a[index] = 0;
}
```

Using a pointer:

```
int a[10], *ptr;
for(ptr=a; ptr < a+10; ptr++) {
    *ptr = 0;
}
```

`ptr++` works because **adding one to an array pointer makes it point to the next element in the array**.

The two loops are the same loop. The index version computes `a + index` on every iteration and dereferences it; the pointer version keeps that address in `ptr` and moves it by one *element* — two cells — each time round. `ptr < a+10` is the same bound as `index < 10`: the address one past the last element, which is the first address the loop must not touch. Both stop *before* it. The `for` itself is the repetition column of Lecture 13's three-shapes figure; the anatomy is in c-control-flow[39].

Preprocessor

A C program is first parsed by a **preprocessor**, which replaces preprocessor directives with substitute text. The two directives used here are `#include …` and `#define …` (macros); there are many others, such as `#if`, `#ifdef`, `#else` and `#undef`.

Figure 6 shows `#include`. A file `name.c` has `#include <header.h>` at its top, the directive names a second file `header.h`, and the combined text is compiled to machine code (M/C).

[38] courses/csse2010/memory-models.pdf
[39] courses/csse2010/c-control-flow.pdf

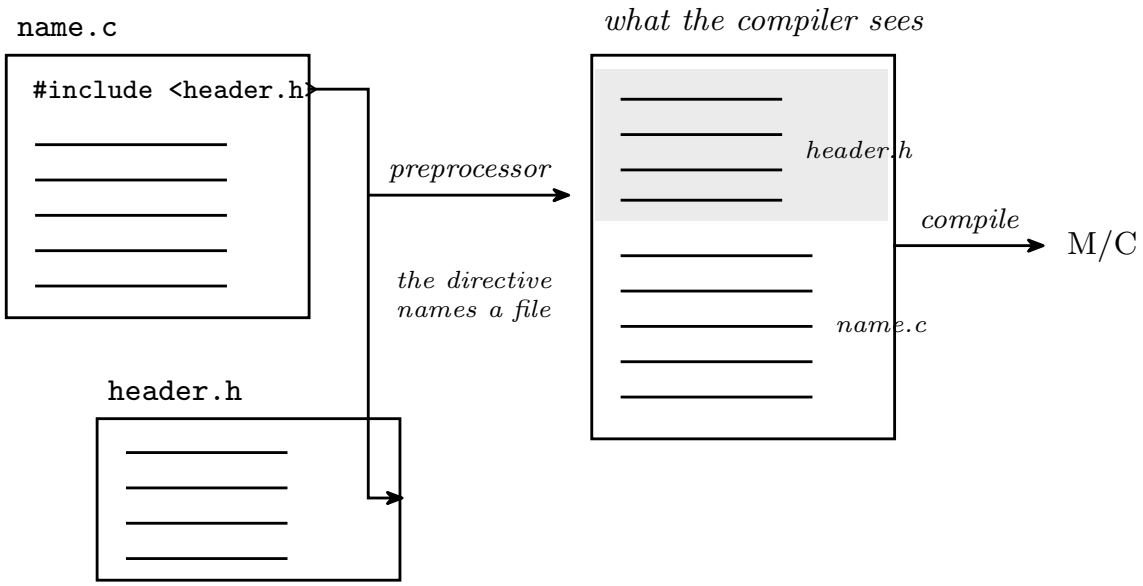


Figure 6: What `#include` does. The directive is not an instruction to the compiler; it is replaced, textually, by the contents of the named file before the compiler sees anything — so what gets compiled is one file, `name.c` with `header.h` pasted in at the top, drawn on the right.

Header (`.h`) files

Header files (`.h` files) hold common macros, function prototypes and so on, like `m324Adef.inc` for assembly (avr-io-ports[40]).

Two examples:

- **`stdio.h`** — includes the prototypes of `printf()` etc. The AVR has a simplified version of this (e.g. no files), and `printf()` will only work if you specify where the characters go, e.g. sent via the serial port.
- **`avr/io.h`** — the AVR's header of useful constants and functions/macros. **Its definitions depend on the actual device targeted.**

`avr/io.h` is what makes `DDRA = 0xFF` a legal line of C. The macro sections below explain why.

Preprocessor examples

Form	Kind of header
`#include<filename.h>`	**system**
`#include"filename.h"`	**user-defined**

Angle brackets look in the system's include directories — where `avr/io.h` and `stdio.h` live; double quotes look in your own project first.

A macro is a textual substitution:

Macro	Use
`#define pi 3.1415`	in `A = pi * r^2;` the `pi` is replaced by `3.1415`
`#define EVER ;;`	in `for(EVER) { … }` the `EVER` is replaced by `;;`, giving `for( ;; ) { … }`
`#define sum(a,b) (a+b)`	

`EVER` shows the mechanism: `for(EVER)` is not a language feature, it is `for(;;)` after two semicolons have been pasted in, and `for(;;)` with no condition loops forever. **The preprocessor does not know C.** It replaces text, and only afterwards does the compiler read the result as a program.

[40] courses/csse2010/avr-io-ports.pdf

Worked: `SQUARE(1+2)`

Given `#define SQUARE(x) (x*x)`, what is the result of `SQUARE(1+2)`?

5. The substitution is literal, `(1+2*1+2)`, and it is evaluated as written: multiplication first, so $1 + 2 + 2 = 5$. Not 9.

The fix is to define the macro as `((x)*(x))`. Then `SQUARE(1+2)` is `((1+2)*(1+2)) = 9`. The rule: **parenthesise every parameter, and the whole body**, because the argument is pasted in as text and inherits whatever operators surround it. The AVR headers below are written this way for exactly this reason, with brackets around `(mem_addr)` and `(io_addr)`.

AVR C macro examples

From `avr/io.h` (via the device-specific header it pulls in):

```
#define _MMIO_BYTE(mem_addr) \
                (*(volatile uint8_t *)(mem_addr))
#define _SFR_IO8(io_addr) \
                _MMIO_BYTE((io_addr) + 0x20)

/* Input Pins, Port A */
#define PINA      _SFR_IO8(0x00)

/* Data Direction Register, Port A */
#define DDRA      _SFR_IO8(0x01)

/* Data Register, Port A */
#define PORTA     _SFR_IO8(0x02)
```

The macros chain: `DDRA` is defined with `_SFR_IO8`, and `_SFR_IO8` with `_MMIO_BYTE`, adding `0x20` on the way. Figure 7 unrolls the chain for `DDRA = 0xFF`.

Read the innermost line from the inside out: `(uint8_t *)(0x21)` says *treat the number `0x21` as the address of an 8-bit unsigned byte*; the outer `*` dereferences it, so the whole expression *is that byte*; and `= 0xFF` stores into it. `volatile` tells the compiler the byte can change behind its back — a hardware register does — so it must not cache the value in a register or skip a store it thinks redundant.

The `+ 0x20` is the same fact as the two columns of the register table in avr-io-ports[41]: the address an `out` instruction takes is the *I/O space* number, and the same register sits `0x20` higher in the *data* space where `lds`/`sts` — and a C pointer — reach it. `DDRA` is I/O `0x01`, data `0x21`, and the macro converts one to the other.

[41] courses/csse2010/avr-io-ports.pdf

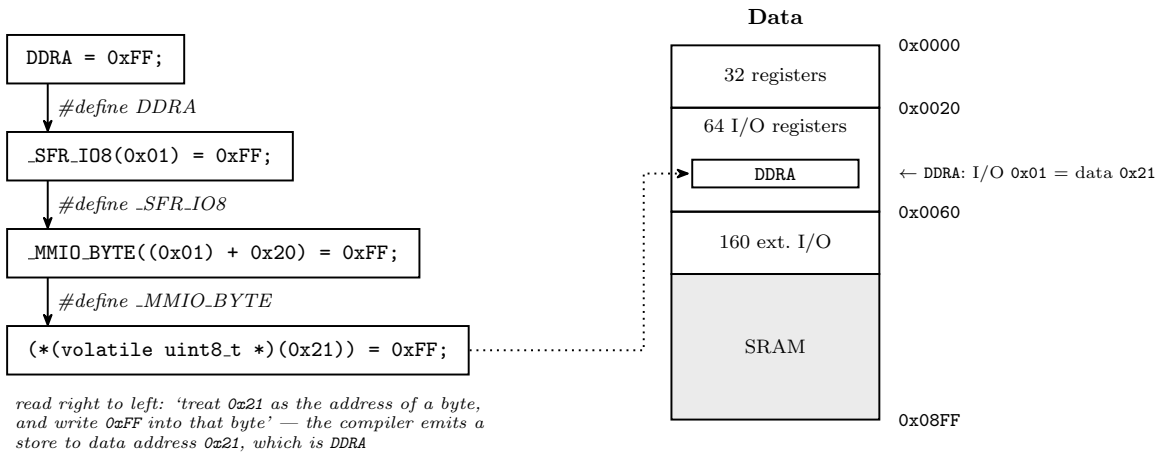


Figure 7: The macro chain behind `DDRA`, unrolled from the three `#define` lines. `DDRA` expands to `_SFR_IO8(0x01)`, which expands to `_MMIO_BYTE(0x21)`, which expands to a dereferenced pointer to data address `0x21`. The `+ 0x20` is the offset between the I/O-space number an `out` instruction uses and the data-space address of the same register — the two columns of the I/O register table in the AVR I/O ports note. The memory map on the right is the datasheet's, as in Lecture 11.

I/O register names as variables

The upshot of these macro definitions is that **I/O register names can be used as variables**:

```
DDRA = 0xFF;
PORTA = PINC | 0xF0;
```

Compare the assembly for the first (avr-io-ports[42]): `ser r16` then `out DDRA, r16`. The C is one line, and the compiler chooses the register. The second line is a read-modify-write across two ports — read PINC, OR in the mask, write PORTA — and the `| 0xF0` is a *set* mask in the sense of bit-masking[43]: the top four bits are forced to 1, the bottom four pass through from PINC.

Seven segment display programming example

Figure 8 draws the display and the byte that drives it.

[42] courses/csse2010/avr-io-ports.pdf
[43] courses/csse2010/bit-masking.pdf

The display has seven bars, lettered **a** across the top, **b** upper right, **c** lower right, **d** across the bottom, **e** lower left, **f** upper left, **g** across the middle, and a small circle **dp** for the *decimal point*. Setting the five segments `a`, `b`, `g`, `e` and `d` to 1 lights the digit **2**.

The microcontroller (`MC`) drives the display from port A: eight wires, one per segment. The bit order is:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
dp	**g**	**f**	**e**	**d**	**c**	**b**	**a**

Two worked bytes:

Digit	dp g f e d c b a	Value	Statement
0	`0 0 1 1 1 1 1 1`	$32 + 16 + 15 = 63$	`PORTA = 63;`
2	`0 1 0 1 1 0 1 1`	$64 + 16 + 8 + 3 = 91$	`PORTA = 91;`

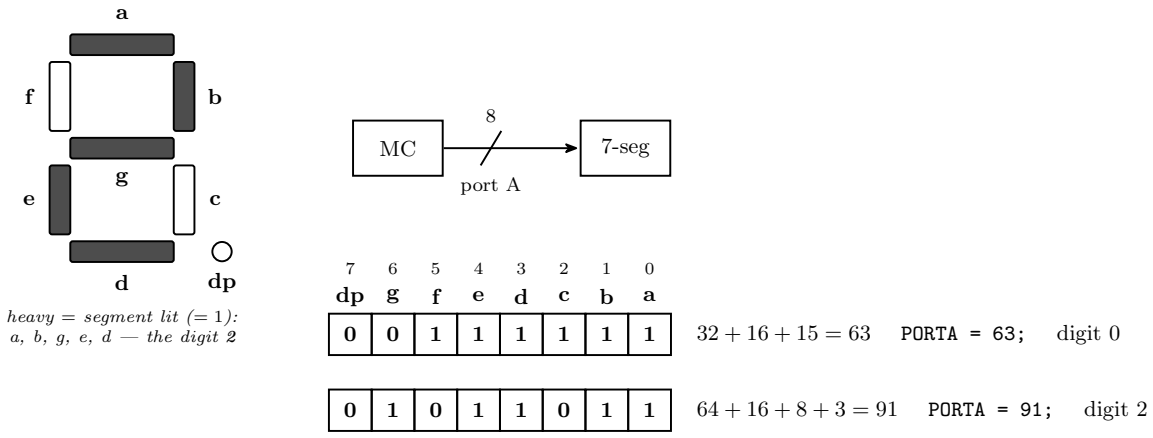


Figure 8: The seven-segment display and its byte. Left: the seven segments lettered `a` to `g` clockwise from the top with `g` across the middle, plus the decimal point; the segments drawn heavy are the ones set to 1, which spell a **2**. Right: the byte written to port A, one bit per segment, `dp` at bit 7 down to `a` at bit 0, with two worked bytes: 63 for a 0 and 91 for the 2 on the left.

The sums read each byte in chunks: `00111111` is $32 + 16$ for bits 5 and 4 and then 15 for the low nibble `1111`; `01011011` is $64 + 16 + 8$ for the three high ones and `11` $= 3$ at the bottom. Full encoding table for all ten digits in seven-segment-display[44].

[44] courses/csse2010/seven-segment-display.pdf

Seven segment display on IO board

The header pins along the board's top edge are `TX RX CC DP G F E D C B A B3 B2 B1 B0 PR`. The display's pins are `CC` to `A`; the silk-screen brackets the display pins as **SSD**. `CC` is the **common cathode** pin, and it selects the digit:

left digit **ON**	**CC = 1**
right digit **ON**	**CC = 0**

Two digits, one set of segment lines. The `A`–`G` and `DP` pins go to *both* digits; `CC` selects which one is actually lit. So a two-digit number is shown by alternating — drive the left digit's segments with CC high, then the right digit's with CC low, fast enough that both look steady. The header itself is in io-board[45].

The program

```
#include <avr/io.h>

/* Seven segment display values */
uint8_t seven_seg[10] = { 63,6,91,79,102,109,125,7,127,111};

int main(void) {
  uint8_t digit;

  /* Set port A pins to be outputs, port C pins to be inputs */
  DDRA = 0xFF;
  DDRC = 0;    /* This is the default, could omit. */
  while(1) {
    /* Read in a digit from lower half of port C pins */
    /* We read the whole byte and mask out upper bits */
    digit = PINC & 0x0F;
    /* Write out seven segment display value to port A */
    if(digit < 10) {
      PORTA = seven_seg[digit];
    } else {
      PORTA = 0;
    }
  }
}
```

Everything above is in this one listing:

[45] courses/csse2010/io-board.pdf

- `#include <avr/io.h>` — angle brackets, a system header; it is what makes `DDRA`, `DDRC`, `PINC` and `PORTA` legal names, and `uint8_t` too.
- `uint8_t seven_seg[10] = { … }` — an array, declared with its ten values in braces, *outside* `main` so it is not one of `main`'s locals. Its index is the digit, its contents the byte from Figure 8: the first three values, 63, 6 and 91, are the digits 0, 1 and 2, and `seven_seg[0]` and `seven_seg[2]` are the two bytes worked above. `uint8_t` is the header's name for an unsigned 8-bit integer — `unsigned char` with its size written on it.
- `DDRA = 0xFF; DDRC = 0;` — direction registers, port A all outputs, port C all inputs, exactly as in week6-lab-avr-io-ports[46] but without the `ldi`/`out` pairs.
- `while(1)` — the repetition column of Lecture 13's three-shapes figure with a condition that is always true. An embedded program has nowhere to return to, so its `main` never finishes; every AVR program from here on has this loop.
- `digit = PINC & 0x0F;` — read the input port, keep the low nibble: bit-masking[47], the *extract* case, in C.
- `if(digit < 10)` — selection, guarding the array: indices 10–15 are possible from four input bits and `seven_seg[10]` does not exist. The `else` blanks the display instead of reading past the end of the array.

See also

- c-pointers-and-arrays[48] — arrays, pointers, pointer arithmetic and traversal
- c-preprocessor-and-avr-headers[49] — `#include`, `#define`, the macro pitfalls and `_SFR_IO8`
- seven-segment-display[50] — the segment layout, all ten codes, and the two-digit `CC` trick
- c-operators[51] — `>>` on signed and unsigned, and `1 << n`
- c-control-flow[52] — `for`, `while(1)` and the `if` guard
- 2026-09-07-introduction-to-embedded-c-programming[53] — Lecture 13
- avr-io-ports[54] · bit-masking[55] · memory-models[56] · addressing-modes[57] · io-board[58]

[46] courses/csse2010/week6-lab-avr-io-ports.pdf
[47] courses/csse2010/bit-masking.pdf
[48] courses/csse2010/c-pointers-and-arrays.pdf
[49] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[50] courses/csse2010/seven-segment-display.pdf
[51] courses/csse2010/c-operators.pdf
[52] courses/csse2010/c-control-flow.pdf
[53] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[54] courses/csse2010/avr-io-ports.pdf
[55] courses/csse2010/bit-masking.pdf
[56] courses/csse2010/memory-models.pdf
[57] courses/csse2010/addressing-modes.pdf
[58] courses/csse2010/io-board.pdf

Reference material

Addressing Modes

Instruction operands can come from different places, and how an instruction says where to find its operand is its *addressing mode*. It is part of the instruction encoding, so it is part of the ISA (instruction-set-architecture[59]) — see avr-instructions[60] for the instructions named here.

> The names below are **as per Tanenbaum, *Structured Computer Organization***. The **AVR instruction set manual uses different names** for the same ideas, so match on the behaviour rather than the word when reading the datasheet.

The four modes

Mode	What is in the instruction	Where the operand is	AVR example
Immediate	the operand **value** itself	in the instruction word	`andi r17, 0xBA`
Register	only **register numbers**	in a register	`and r18, r19`
Direct	the **memory address**	in memory, at that address	`lds r15, $1234`
Indirect	a **register number**; that register holds the address	in memory, at the address the register holds	`ld r5, X`

Figure 9 is the whole comparison in one line: **immediate, register, direct, indirect is a list in increasing order of indirection**, and every step down it either costs an instruction word or costs a hop through some storage on the way to the operand.

Immediate addressing

An operand value is in the instruction. `andi r17, 0xBA` carries `0xBA` inside the instruction word itself — nothing is fetched from anywhere, the bits *are* the number. On AVR the immediate instructions are the ones whose mnemonic ends in `i` (`ldi`, `andi`, `ori`, `cpi`) and they pay for the immediate with encoding space: only four bits are left for the register, which is exactly why they reach **r16–r31 only**.

[59] courses/csse2010/instruction-set-architecture.pdf
[60] courses/csse2010/avr-instructions.pdf

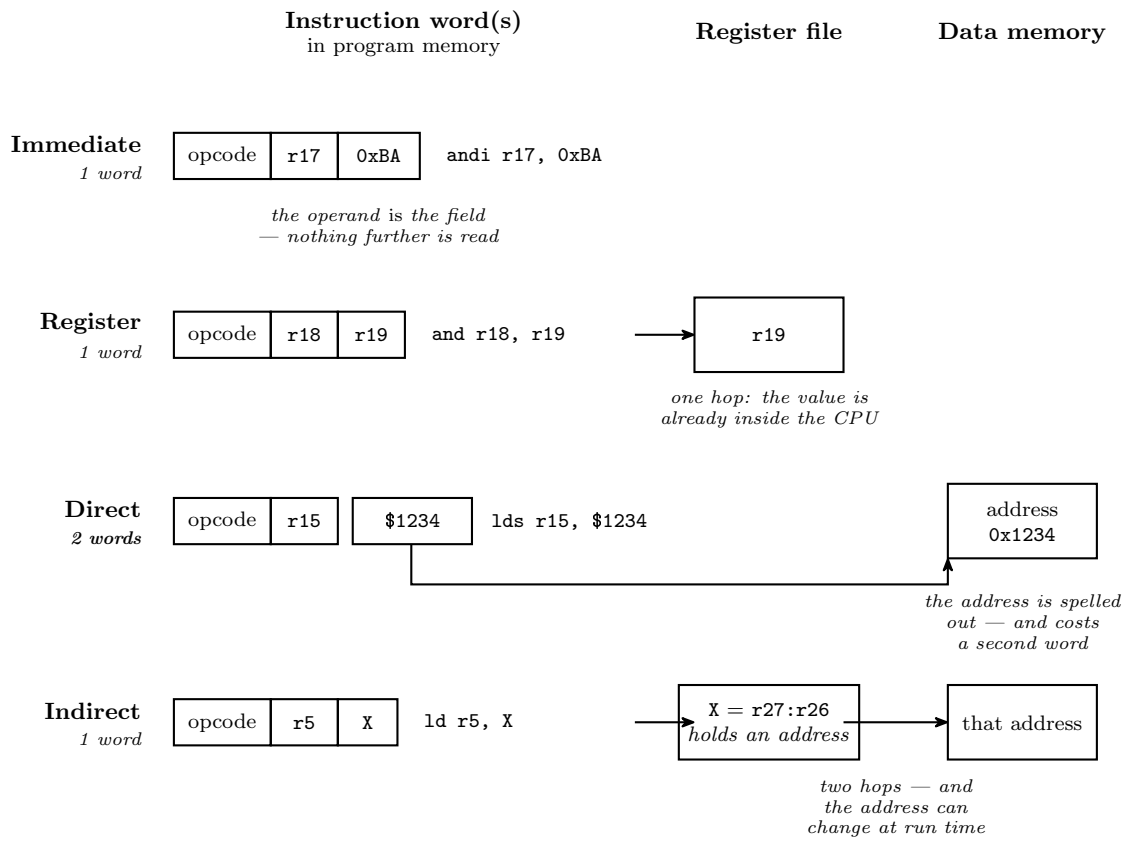


Figure 9: The four modes as one question: how far does the machine have to go to reach the operand? Immediate stops at the instruction itself. Register takes one hop, into the register file. Direct takes one hop into memory, but needs a second instruction word to carry the address. Indirect takes two hops and stays one word, and because its address lives in a register it can change while the program runs. The field splits shown inside the instruction words are schematic, not the real AVR encodings.

Register addressing

Only register numbers in the instruction. `and r18, r19` names two registers and nothing else; the values are already inside the CPU. This is the cheapest mode there is — no memory access at all — and it is why the AVR is a **load/store machine**: no arithmetic or logic instruction touches memory, its operands are either registers (`and r18, r19`) or an immediate carried in the instruction word (`andi r17, 0xBA`). Anything in SRAM has to be loaded into a register first.

Direct addressing

Memory address is in the instruction. `lds r15, $1234` spells the address out in full.

These AVR instructions occupy two instruction words (32 bits): sixteen bits are nowhere near enough for both an opcode and a 16-bit address, so the address gets a word of its own. That is the standing trade — a direct access is simple to read and needs no setup, but it costs double the program memory and cannot be computed at run time.

Indirect addressing

Memory address in register; register number is in the instruction. `ld r5, X` says "load from wherever X points", and X is the pointer register pair of avr-architecture[61].

The instruction stays one word, because it only has to name *which* pointer. The address itself is data now, not code — which is what makes loops over arrays possible: increment the pointer and the same instruction reaches the next element.

Reading the trade-off

	Instruction words	Address decided	Costs
Immediate	1	when you write the program	encoding space, hence `ldi`'s r16–r31 limit
Register	1	—	nothing; the value is already in the CPU
Direct	**2**	when you write the program	an extra program-memory word per access
Indirect	1	**at run time**	a pointer register, and loading it first

[61] courses/csse2010/avr-architecture.pdf

The pattern: **anything the instruction has to *say* costs instruction bits; anything it can *point at* costs a register and a setup step instead.** Direct and indirect are the same access to the same memory cell — they differ only in which of those two prices you pay.

What the extra word does *not* cost

`ld`/`st` (indirect) are **one instruction word, two cycles**; `lds`/`sts` (direct) are **two instruction words, two cycles**. So direct addressing's extra cost is paid entirely in **program memory, not in time** — both memory forms take the same two cycles, because both make one data-memory access and that access is what dominates.

For contrast, `in`/`out` use neither mode — they name an I/O register number directly in the instruction — and take **one word, one cycle**. That one-cycle figure is the reason `in`/`out` exist at all rather than everything going through `lds`/`sts` (avr-io-ports[62]).

See also

- instruction-set-architecture[63] — addressing modes as part of the "instructions" pillar
- avr-instructions[64] — `ld`/`lds`/`st`/`sts` and the immediate instructions
- avr-architecture[65] — the X, Y and Z pointer registers indirect mode uses
- memory-models[66] — the address space being addressed
- 2026-09-03-instruction-set-architecture-part-2[67] — Lecture 12

The Assembly Process

How an assembly language program becomes machine code. It covers the four fields of a statement, the directives that tell the assembler what to do as opposed to the CPU, the three memory segments of the AVR and the location counter that walks through each, and the two-pass assembler with its symbol table, which is how a program can jump to a label it has not reached yet. The program skeleton these pieces fit into is avr-program-structure[68], and the memories the segments map onto are in memory-models[69] and avr-architecture[70]. First met in 2026-10-08-assembly-process[71].

[62] courses/csse2010/avr-io-ports.pdf
[63] courses/csse2010/instruction-set-architecture.pdf
[64] courses/csse2010/avr-instructions.pdf
[65] courses/csse2010/avr-architecture.pdf
[66] courses/csse2010/memory-models.pdf
[67] courses/csse2010/2026-09-03-instruction-set-architecture-part-2.pdf
[68] courses/csse2010/avr-program-structure.pdf
[69] courses/csse2010/memory-models.pdf
[70] courses/csse2010/avr-architecture.pdf
[71] courses/csse2010/2026-10-08-assembly-process.pdf

What an assembler does

An assembler turns a text file of assembly language into the bits that go into the chip's memories. Most of that is translation, one statement at a time: `add r1, r2` has a fixed binary form, `0000 1100 0001 0010`, and nothing else in the file can change it (avr-instructions[72]). Each instruction becomes exactly one machine instruction (instruction-set-architecture[73]).

Two things make it more than a lookup table:

- **A program is more than instructions.** It has to say where things go in memory, reserve space for variables, put constant tables in flash and give registers and addresses readable names. None of that is an instruction for the CPU. These lines are **directives**, commands to the assembler itself.
- **Names stand for numbers that are not known yet.** `jmp RESET` needs the address of `RESET`, and if `RESET:` is further down the file, a single read from top to bottom cannot supply it. The cure is to read the file twice.

C reaches machine code by a different road: the compiler turns one line of C into many instructions (2026-09-07-introduction-to-embedded-c-programming[74]), and how that output is joined into a program is compilation-and-linking[75].

The format of a statement

An assembly program is a sequence of **one-line statements**. A statement has up to **four fields**, and any of them may be empty:

Label. A name for a position in memory. Labels exist so that statements can be **jumped to** and so that **data can be referred to** by name. Its value is the address the statement lands at: a word address in program memory for code, a byte address in data memory for a variable. By convention a label starts in column 1 and everything else is indented, so labels stand out down the left edge. In AVR assembly it ends with a colon.

Operation. A **symbolic abbreviation** for an instruction's opcode, a mnemonic: `LDS` for *Load Direct from data Space*, `ldi` for *load immediate*. Or a directive, which starts with a dot.

Operands. The **registers and addresses** the instruction uses. Register names belong to the machine:

Machine	Register names
Pentium	`eax`, `ebx`, `esi`, `edi`, …

[72] courses/csse2010/avr-instructions.pdf
[73] courses/csse2010/instruction-set-architecture.pdf
[74] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[75] courses/csse2010/compilation-and-linking.pdf

Machine	Register names
SPARC	`o0`..`o7`, `i0`..`i7`, `l0`..`l7`, `g0`..`g7` (out, in, local and global)
AVR	simply `r0`..`r31`

An address operand is usually written as a label, `lds r16, count`, and the assembler substitutes the number.

Comment. Space for explanation, **only for people**: the assembler discards it. In assembly comments are not optional politeness. `ldi r20, 13` says nothing about why; the comment is the only place the intent is written down.

Ending a statement. Some assemblers want a semicolon at the end of every statement and others do not. AVR assembly does not: the end of the line ends the statement, and a semicolon *starts a comment*.

Directives

Besides instructions, the assembler needs other information, for example:

- **where in memory** the code and data should go;
- **nicer names** for registers and addresses, such as `temp` for `r16` or `SPH` for I/O address `0x3E`;
- **space for variables** and **tables of constants**.

That information is given as a **directive**, also called a **pseudo-instruction**: a command to the assembler itself, written where an instruction would be and **preceded by a dot**, as in `.byte` or `.def temp = r16`. Most directives produce no machine code at all. They change what the assembler knows or where it is putting things.

The assembler in Microchip Studio, **AVRASM2**, has these, among others:

Directive	What it does
`.byte`	reserve bytes for a variable
`.cseg`	switch to the code segment
`.csegsize`	set the program memory size
`.db`	define constant byte(s)
`.def`	give a register a symbolic name
`.device`	name the exact processor the program is for
`.dseg`	switch to the data segment
`.dw`	define constant word(s)

Directive	What it does
`.equ`	set a symbol equal to an expression, permanently
`.eseg`	switch to the EEPROM segment
`.exit`	stop processing this file
`.include`	read source from another file, here
`.list`, `.nolist`	turn generation of the list file on, off
`.listmac`	show macro expansions in the list file
`.macro`, `.endm` (or `.endmacro`)	begin, end a macro
`.org`	set the location counter: the address of what comes next
`.set`	set a symbol equal to an expression, changeably
`.if`, `.else`, `.elif`, `.endif`	conditional assembly: include lines only if a condition holds

All of this is **AVRASM2's syntax**. Other assemblers, even ones for the AVR, spell things differently. The GNU assembler used behind avr-gcc has a `.byte` directive too, and there it *stores* the bytes listed instead of reserving space.

Segments and location counters

The AVR has three memories (avr-architecture[76]), and to the assembler each is a **segment**. Directives choose which segment the following lines go into:

Segment	Directive	Memory	What can go there
code	`.cseg`	flash, program memory	instructions, and constant data
data	`.dseg`	SRAM, data memory	**space only**, reserved for variables
EEPROM	`.eseg`	EEPROM	constants

The data segment cannot be given values. The assembler's output is programmed into flash and EEPROM, which keep their contents with the power off. SRAM is empty at power-up and holds nothing until the running program writes to it, so all the assembler can do there is decide **which addresses** each variable will use.

Each segment has its own **location counter**: the address the next thing in that segment will be placed at. A label takes the current value of its segment's counter, and every statement that

[76] courses/csse2010/avr-architecture.pdf

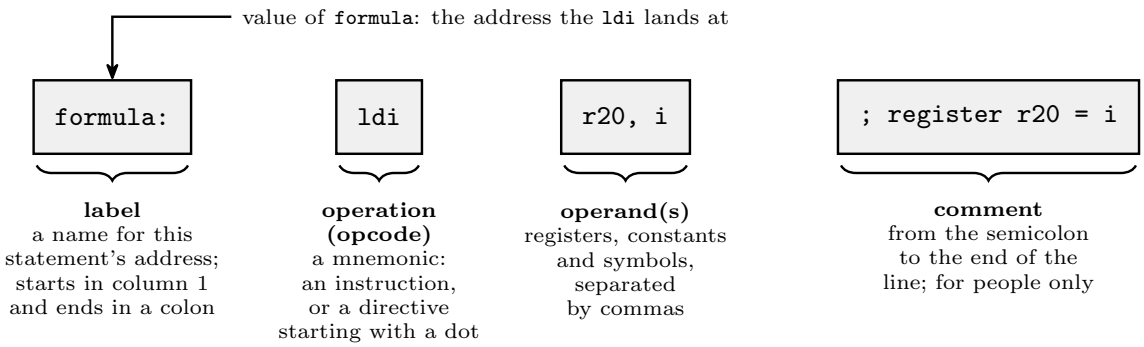


Figure 10: The four fields of one assembly statement. The label names the address this statement ends up at, so other statements can jump to it or refer to the data there. The operation field holds the mnemonic, which is either an instruction for the CPU or, if it starts with a dot, a directive for the assembler. The operands are what the operation works on: registers, constants or symbols, separated by commas. The comment runs from the semicolon to the end of the line and is thrown away by the assembler.

places something advances the counter by its size. Switching segment does not reset anything. It only changes which counter is in use, so **a segment can be built up in pieces**, and each piece carries on where the last piece of the same segment stopped.

Where each counter starts:

- **Code**: at 0, the reset address, and the assembler starts in the code segment, so a file with no segment directives is all code. The counter counts **16-bit words**, because each program-memory address names a word (memory-models[77]).
- **Data**: at the first free SRAM address, `0x0100` on the ATmega324A, since `0x0000` to `0x00FF` are the registers and I/O registers. The counter counts **bytes**.
- **EEPROM**: at 0, counting bytes.

`.org` sets the counter of the current segment to a given value, so whatever follows is placed there. It works in any segment, in that segment's units.

Reserving and defining data

Three directives put data in memory:

[77] courses/csse2010/memory-models.pdf

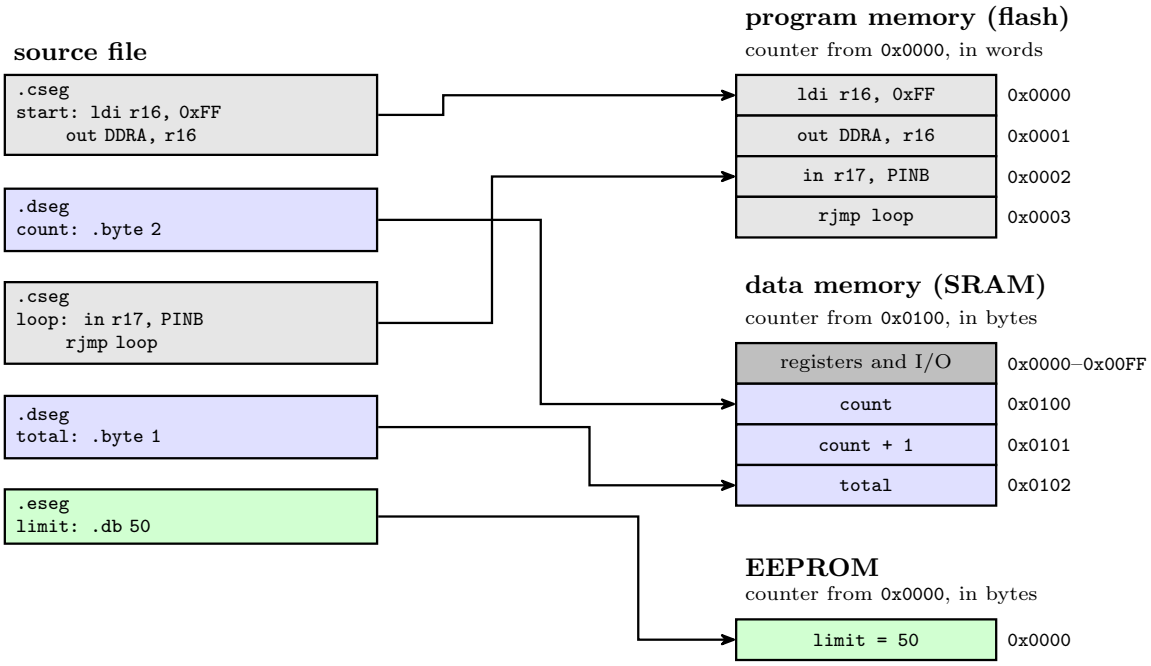


Figure 11: One source file feeding three memories. The file switches segment four times, and each piece goes to the memory its directive names, carrying on from where the previous piece of the same segment stopped: the second code piece follows the first in flash, and `total` follows `count` in SRAM. Each segment has its own location counter. The code counter starts at 0 and counts 16-bit words. The data counter starts at `0x0100`, the first address after the registers and I/O, and counts bytes. The EEPROM counter starts at 0 and counts bytes. Grey marks code, blue data and green EEPROM.

Directive	Segment	Places	Size
`.byte n`	data	nothing: **reserves** `n` bytes	`n` bytes
`.db a, b, …`	code or EEPROM	constant **bytes**, 8 bits each	one byte each
`.dw a, b, …`	code or EEPROM	constant **words**, 16 bits each	two bytes each

`.byte` is allowed only in the data segment: it is how a variable gets an address. `.db` and `.dw` initialise constants, so they belong where constants can live: the code segment or the EEPROM segment.

`.db` in the code segment packs two bytes per word. Program memory is 16 bits wide, so consecutive bytes share a word, the first in the **low byte** and the second in the high byte. A `.db` line with an **odd number of bytes** leaves half a word over, and the assembler fills it with a **zero pad byte** so that the next line starts on a whole word. Each `.db` line is padded on its own, so two 5-byte lines take 3 words each, not 5 words between them. AVRASM2 warns when it pads. Writing the 0 yourself gives the same layout without the warning.

`.dw` takes one whole word per value, so it is the natural way to store numbers bigger than 255, or addresses.

Worked: a lookup table in flash

A **lookup table** (LUT) is a table of constants a program reads instead of computing. Lay out:

```
        .cseg
table1: .db  11, 12, 30
table2: .dw  100, 133
        .org 500
table3: .dw  ...
```

The code counter starts at 0.

- `table1` = **0**. Three bytes: 11 and 12 share word 0, 30 takes the low half of word 1, and a pad 0 fills its high half. The counter moves to 2.
- `table2` = **2**. Two words, 100 at word 2 and 133 at word 3. The counter moves to 4.
- `.org 500` sets the counter to 500. Words 4 to 499 are skipped, and nothing is placed there.
- `table3` = **500** (`0x01F4`).

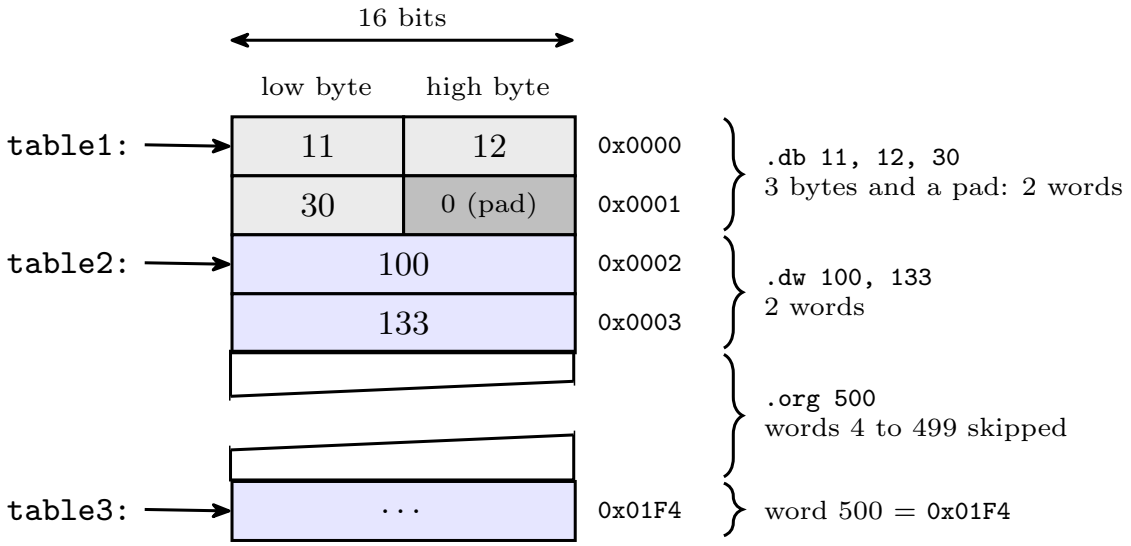


Figure 12: The lookup table laid out in program memory, one 16-bit word per row, lowest address at the top. A `.db` puts two bytes in each word, the first in the low byte, and pads an odd count with a zero byte so that the next label starts on a whole word: here 30 shares word 1 with a pad. Each `.dw` value fills a whole word. The `.org` moves the location counter forward and leaves words 4 to 499 untouched, so `table3` lands at word 500. The diagonal break means the gap is not drawn to scale. Grey marks the `.db` bytes, dark grey the pad and blue the `.dw` words.

On the data side, two `.byte 2` lines in separate `.dseg` pieces end up next to each other, at `0x0100` and `0x0102`, whatever code sits between them in the file. That is worked in 2026-10-08-assembly-process[78].

Naming things: `.def`, `.equ` and `.set`

Three directives give a name to something. None of them places anything or moves a location counter.

Directive	Names	Can it be changed later?	Example
`.def`	a **register**, and only a register	yes, by another `.def`	`.def ZH = r31`
`.equ`	a **value**: a number, an I/O address, an expression	**no**	`.equ sreg = 0x3f ; Status register`
`.set`	a value	**yes**, by another `.set`	`.set io_offset = 0x23 ; For now`

`.equ` is the usual choice for a constant, because a second definition is an error and catches a clash of names. `.set` is for a value that a later part of the file deliberately redefines, which is what *for now* in the comment above says.

Check. The same program defines `.equ SPH = $3E` and `.def ZH = r31`. Why `.equ` for one and `.def` for the other? `SPH` is an **I/O register**, which instructions such as `out` reach by its I/O address, 0x3E. To the assembler that is just a number, so `.equ`. `ZH` is **general-purpose register** `r31`, the high byte of the Z pointer, which instructions name as a register operand, so `.def`.

Two more directives are about files and the processor rather than names:

- `.include "file"` reads another file in at that point, as if its text were typed there. `.include "m324Adef.inc"` is how `PORTA`, `SPH`, `RAMEND` and the rest get defined: the file is mostly `.equ` and `.def` lines (avr-io-ports[79]).
- `.device ATmega324A` names the exact processor. The assembler then rejects any instruction that chip does not implement.
- `.exit` stops reading the file at that line. Anything after it is ignored.

[78] courses/csse2010/2026-10-08-assembly-process.pdf
[79] courses/csse2010/avr-io-ports.pdf

The forward reference problem

Read top to bottom, most statements can be translated on the spot. `add r1, r2` names two registers and nothing else, so its machine word is known at once.

A label used **before** it is defined is different. A program usually starts with

```
        jmp  RESET
        ...
RESET:  ldi  temp, low(RAMEND)
```

When the assembler reaches the `jmp`, it knows the opcode but not the operand: `RESET` has not been seen, so there is no address to put in the instruction. This is the **forward reference problem**. One forward reference anywhere in a file is enough to stop a single top-to-bottom pass from finishing the job.

The two-pass assembler

The fix is to **process the file twice**.

Pass one: define every symbol. Go through the file without generating anything. Keep a location counter for each segment, starting at 0 for code and EEPROM and at `0x0100` for data. For each line:

- if it has a label, enter the label in the **symbol table** with its segment and the current value of that segment's counter;
- if it is `.equ`, `.set` or `.def`, enter that name and its value;
- if it is `.cseg`, `.dseg` or `.eseg`, switch counters; if it is `.org`, set the current counter;
- advance the current counter by the line's size: 1 or 2 words for an instruction, the padded number of words for `.db`, a word per value for `.dw`, `n` bytes for `.byte n`, and 0 for a line that only names something.

Pass one only needs each line's **size**, and every instruction's size is known without knowing its operands: `jmp RESET` takes 2 words whatever `RESET` turns out to be.

Pass two: assemble. Go through the file again with the counters reset. Now every symbol's value is in the table, so each instruction can be **assembled**, turned into actual machine code, with its operands filled in, and each `.db` and `.dw` value can be written.

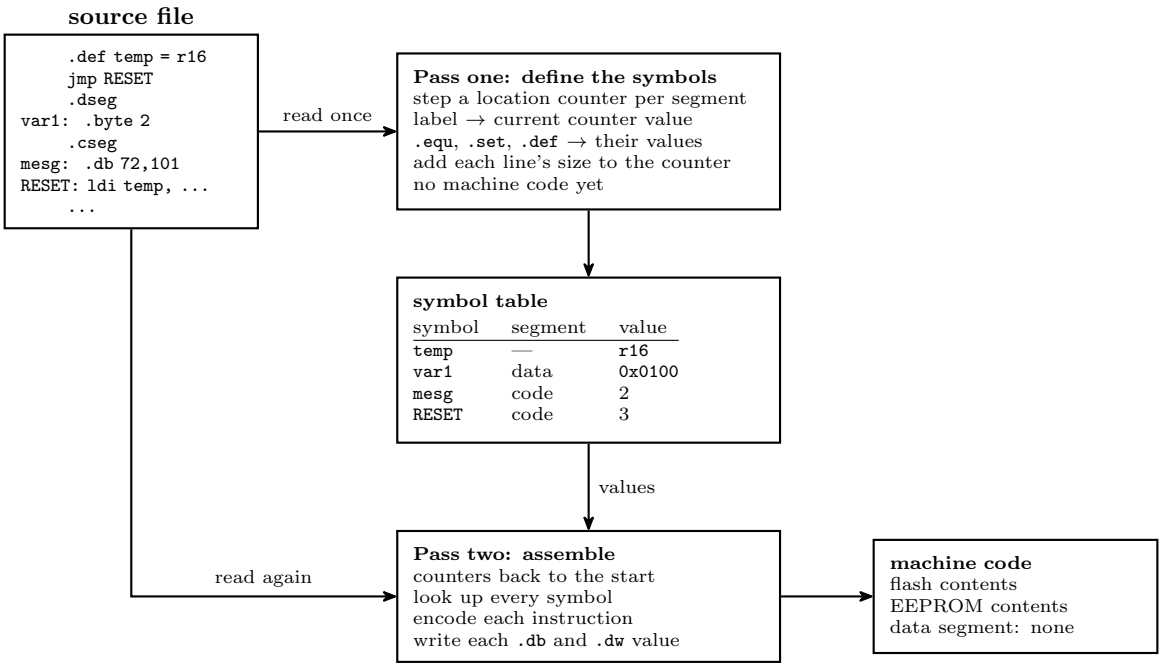


Figure 13: The two-pass assembler. Pass one reads the whole source and produces no code: it steps the location counters and records each symbol's segment and value in the symbol table. Pass two reads the same source again from the top and, with every value now in the table, encodes each instruction and writes each constant. Its output is the contents of flash and of EEPROM. The data segment produces no output, only the addresses its labels were given.

Worked: a two-pass trace

The program:

```
        .equ    RAMEND = $08FF
        .equ    SPH = $3E
        .equ    SPL = $3D
        .def    ZH = r31
        .def    ZL = r30
        jmp     RESET
        .dseg
var1:   .byte   2
var2:   .byte   2
        .cseg
        .def    temp = r16
mesg:   .db     72,101,108,108,111
table:  .dw     var1, var2
RESET:  ldi     temp, low(RAMEND)
        out     SPL, temp
        ldi     temp, high(RAMEND)
        out     SPH, temp
        ...
        ldi     ZH, high(mesg*2)
        ldi     ZL, low(mesg*2)
        lpm                     ;R0 <= M[Z]
```

Pass one. The code counter starts at 0 and the data counter at `0x0100` (256).

- The five `.equ` and `.def` lines enter `RAMEND`, `SPH`, `SPL`, `ZH` and `ZL` in the table and place nothing.
- `jmp RESET` is a 2-word instruction, so it takes words 0 and 1 and the code counter goes to 2. `RESET` is not known yet, and pass one does not need it to be.
- `.dseg` switches to the data counter. `var1` = 256 (`0x0100`) and `.byte 2` moves the counter to 258. `var2` = 258 (`0x0102`) and the counter goes to 260.
- `.cseg` switches back to the code counter, still at 2. `.def temp = r16` enters `temp`.
- `mesg` = **2**. Five bytes need three words with a pad, words 2 to 4, and the counter goes to 5.
- `table` = **5**. Two words, 5 and 6, and the counter goes to 7.
- `RESET` = **7**. Each of the four instructions from here is one word: `ldi` at 7, `out` at 8, `ldi` at 9, `out` at 10.

The counters run code 0, 2, 5, 7, 8, 9, 10, 11 and data 256, 258, 260. The EEPROM counter is never used.

So the instruction at `RESET` is at program address **7**, `mesg` is at **2**, `table` at **5**, and the variables at **256** and **258**.

code address	data address	source line	
			.equ RAMEND = $08FF
			.equ SPH = $3E
			.equ SPL = $3D
			.def ZH = r31
			.def ZL = r30
0–1			jmp RESET
			.dseg
	0x0100–0x0101	var1:	.byte 2
	0x0102–0x0103	var2:	.byte 2
			.cseg
			.def temp = r16
2–4		mesg:	.db 72,101,108,108,111
5–6		table:	.dw var1, var2
7		RESET:	ldi temp, low(RAMEND)
8			out SPL, temp
9			ldi temp, high(RAMEND)
10			out SPH, temp

code counter: 0, 2, 5, 7, 8, 9, 10, 11
data counter: 0x0100, 0x0102, 0x0104 (256, 258, 260)

symbol table

symbol	segment	value
RAMEND	—	0x08FF
SPH	—	0x3E
SPL	—	0x3D
ZH	—	r31
ZL	—	r30
var1	data	0x0100 (256)
var2	data	0x0102 (258)
temp	—	r16
mesg	code	2
table	code	5
RESET	code	7

Figure 14: Pass one over the example, line by line. The two address columns show where each line is placed in its own segment, so a range means the line takes several words or bytes: the `jmp` two words, the 5-byte `.db` three words with its pad, the `.dw` two words, each `.byte 2` two bytes. Lines that only name something or switch segment take no space and have no address. The code and data counters advance independently, which is why `mesg` comes straight after the `jmp` even though the data lines sit between them in the file. On the right is the symbol table pass one leaves behind: every name, the segment its value belongs to, and the value. The `.equ` and `.def` names have no segment.

Only one reference in the program is forward: `jmp RESET`. Every other symbol (`RAMEND`, `var1`, `var2`, `mesg`, `SPL`, `SPH`, `ZH`, `ZL`, `temp`) is defined above the line that uses it. One is enough to need the second pass.

Worked: what pass two writes

With the table complete, pass two replaces every symbol by its value and encodes:

Address	Statement	With symbols replaced	Machine word(s)
0–1	`jmp RESET`	`jmp 7`	`0x940C 0x0007`
2–4	`.db 72,101,108,108,111`	bytes 72, 101, 108, 108, 111 and a pad 0	`0x6548 0x6C6C 0x006F`
5–6	`.dw var1, var2`	`.dw 0x0100, 0x0102`	`0x0100 0x0102`
7	`ldi temp, low(RAMEND)`	`ldi r16, 0xFF`	`0xEF0F`
8	`out SPL, temp`	`out 0x3D, r16`	`0xBF0D`
9	`ldi temp, high(RAMEND)`	`ldi r16, 0x08`	`0xE008`
10	`out SPH, temp`	`out 0x3E, r16`	`0xBF0E`
later	`ldi ZH, high(mesg*2)`	`ldi r31, 0x00`	`0xE0F0`
later	`ldi ZL, low(mesg*2)`	`ldi r30, 0x04`	`0xE0E4`
later	`lpm`	`lpm`	`0x95C8`

Points worth checking by hand:

- **`ldi r16, 0xFF`.** The form is `1110 KKKK dddd KKKK`, with `dddd` the register number minus 16. `K` = `1111 1111` and `dddd` = `0000`, so `1110 1111 0000 1111` = `0xEF0F`.
- **The `.db` words.** The first byte of each pair is the low byte: 72 and 101 make `0x6548`. The odd fifth byte, 111 = `0x6F`, shares its word with the pad.
- **`low()` and `high()`** split a 16-bit value into its bytes at assembly time. `RAMEND` = `0x08FF` gives `0xFF` and `0x08`: this is the stack pointer being set to the top of SRAM (procedures-and-the-stack[80]).
- **`mesg*2`.** `mesg` is a word address, 2. `lpm` reads one byte of program memory from the **byte** address in Z, and word 2 starts at byte 4. So Z = `0x0004`, and `lpm` loads `r0` with the byte there, 72, the `H` of `Hello`. The two kinds of address are drawn in memory-models[81].

[80] courses/csse2010/procedures-and-the-stack.pdf
[81] courses/csse2010/memory-models.pdf

- `table` holds the two variables' addresses, `0x0100` and `0x0102`: a table of pointers, in flash, into the data segment.

Common mistakes

- **Counting `jmp` as one word.** It is two, because it carries a full program address (avr-branches-and-jumps[82]). `rjmp` is one.
- **Forgetting the pad.** A `.db` line with an odd number of bytes takes a whole extra word, and each line is padded separately.
- **Counting `.db` bytes as words.** Two bytes share a word in program memory.
- **Letting `.dseg` lines move the code counter**, or the other way round. Each segment has its own counter, and switching segments only chooses which one is in use.
- **Giving size to lines that only name things.** `.equ`, `.set`, `.def`, `.cseg`, `.dseg`, `.include` and comments place nothing.
- **Expecting `.byte` to store a value.** `.byte 2` reserves two bytes. It does not put a 2 anywhere. Constants go in flash or EEPROM with `.db` or `.dw`.
- **`.def` for a constant, or `.equ` for a register.** `.def` is for registers only.
- **Redefining an `.equ`.** It cannot be changed; that is what `.set` is for.
- **Handing a code label straight to `lpm`.** A code label is a word address. `lpm` wants the byte address, the label times 2.
- **Ending statements with a semicolon** out of habit from another language. In AVR assembly it starts a comment.
- **Assuming another assembler reads the same way.** Directive names and meanings change between assemblers, even for the same chip.

See also

- avr-program-structure[83] — the skeleton: vector table, `RESET`, the main loop, and where segments and directives go in it
- memory-models[84] — word and byte addresses, cell sizes, and the AVR's separate address spaces
- avr-architecture[85] — flash, SRAM and EEPROM, the three memories behind the three segments · memory-technologies[86] — what each of the three is as a technology
- avr-instructions[87] · avr-branches-and-jumps[88] — instruction sizes, which decide how far the code counter moves

[82] courses/csse2010/avr-branches-and-jumps.pdf
[83] courses/csse2010/avr-program-structure.pdf
[84] courses/csse2010/memory-models.pdf
[85] courses/csse2010/avr-architecture.pdf
[86] courses/csse2010/memory-technologies.pdf
[87] courses/csse2010/avr-instructions.pdf
[88] courses/csse2010/avr-branches-and-jumps.pdf

- avr-interrupts[89] — `.org` placing each `jmp` in the vector table
- procedures-and-the-stack[90] — the stack pointer set-up the worked example starts with
- instruction-set-architecture[91] — assembly as a level, one line to one instruction
- compilation-and-linking[92] — the road to machine code from C
- 2026-10-08-assembly-process[93] — Lecture 20

AVR Instructions

The AVR instructions introduced so far, in assembly and in machine code. See instruction-set-architecture[94] for what an instruction set *is* and avr-architecture[95] for the hardware that runs these.

The full alphabetical instruction set — each mnemonic with its operation, description, **flags affected**, **clock count**, opcode pattern and an example — is in the AVR instruction set reference: Atmel's *AVR Instruction Set Manual*, and the instruction summary in the ATmega324A datasheet the course says to keep open. The notation key immediately below and the per-instruction flag table at the end are *unverified* against that reference — check them against your own copy before leaning on them.

Reading the instruction set reference

Operand notation — how an operand's allowed values map onto opcode bits:

Notation	Meaning	Values	Opcode pattern
`Rd`	destination register (may be a source too)	r0…r31	`ddddd`
		r16…r31	`dddd` (register number − 16)
`Rr`	source register	r0…r31	`rrrrr`
`Rh:Rl`	register pair as a 16-bit quantity	r25:r24 / X / Y / Z	`dd` (00 / 01 / 10 / 11)
`K`	constant data	0…255	`KKKKKKKK`
`k`	constant address	−64…63, −2048…2047, 0…65535	`kkkkkkk` … `kkkk kkkk kkkk kkkk`

[89] courses/csse2010/avr-interrupts.pdf
[90] courses/csse2010/procedures-and-the-stack.pdf
[91] courses/csse2010/instruction-set-architecture.pdf
[92] courses/csse2010/compilation-and-linking.pdf
[93] courses/csse2010/2026-10-08-assembly-process.pdf
[94] courses/csse2010/instruction-set-architecture.pdf
[95] courses/csse2010/avr-architecture.pdf

Notation	Meaning	Values	Opcode pattern
`b`	bit number in a register	0…7	`bbb`
`s`	bit in the status register	0…7	`sss`
`P`	I/O register number	0…63	`PPPPPP`
`q`	displacement for direct addressing	0…63	`qqqqqq`

The `Rd` row is the whole story behind `LDI`'s restriction — an instruction spending five bits on the register gets all 32; one spending four gets sixteen, and pays for it by starting at r16.

Operation notation — used in the *Operation* column:

`M[a]` / `PM[a]` / `IO[a]`	SRAM cell / program memory (flash) byte / I/O register at address `a`
`PC` / `SP` / `STACK`	program counter / stack pointer / `M[SP]`
←	assigned value
`reg(a)`	bit `a` of register `reg`, numbered 7…0 most to least significant

Flag names in the *Flags Affected* column are the `SREG` bits of avr-architecture[96].

Register conventions

Two operand names recur throughout the instruction set:

- **`rd`** — the **destination** register.
- **`rr`** — the **source** register.

AVR is a **two-operand** machine: the destination is also one of the sources, so `add rd, rr` means `rd` ← `rd` + `rr`. (Contrast a three-operand ISA such as ARM, where `add rd, r1, r2` leaves both sources untouched.) This matters when reading code — an AVR arithmetic instruction always overwrites one of its inputs.

[96] courses/csse2010/avr-architecture.pdf

Instructions by type

The instruction set can be cut two ways.

By what the instruction touches — the instruction-type list of instruction-set-architecture[97], filled in with the instructions so far:

Type	Does	AVR so far
Input/Output	communicate with I/O devices/ports	`IN`, `OUT`
Load/Store	move data from/to memory	`LD`, `LDS`, `ST`, `STS`
Move	copy data between registers	`MOV`
Arithmetic	addition, subtraction, …	`ADD`, `ADC`, `SUB`, `INC`, `NEG`
Logical	Boolean operations	`AND`, `ANDI`, `OR`, `ORI`, `EOR`, `COM`
Branching	deciding which instruction to perform next	`JMP`, `BREQ`, `BRNE`

By operand count, which splits the arithmetic and logical group instead:

		AVR examples
Dyadic	combine **2 operands** to produce a result	`add r5, r6` (arithmetic) · `and r5, r6` (logical, bitwise)
Monadic	**one operand** → one result	`ser r16` (set) · `clr r5` (clear) · `neg r5` (2's complement) · `com r5` (1's complement) · the shifts and rotates

The second cut is the one that predicts the syntax: a dyadic instruction has a comma in it, a monadic one does not.

The instructions so far

Instruction	Meaning	Example
`MOV rd, rr`	copy contents of `rr` into `rd` — `rd` ← `rr`	`mov r3, r14`
`ADD rd, rr`	add contents of `rr` to `rd` — `rd` ← `rd` + `rr`	`add r5, r6`

[97] courses/csse2010/instruction-set-architecture.pdf

Instruction	Meaning	Example
`ADC rd, rr`	add **with carry** — `rd` ← `rd` + `rr` + C	`adc r11, r5`
`AND rd, rr`	bitwise AND, result into `rd`	
`OR rd, rr`	bitwise OR, result into `rd`	
`EOR rd, rr`	bitwise exclusive-OR, result into `rd`	
`INC rd`	`rd` ← `rd` + 1	`inc r22`
`LDI rd, number`	load immediate — put the number into `rd`	`ldi r16, 73`
`SER rd`	set all bits — `rd` ← `$FF`. Same as `ldi rd, 0xFF`, so **r16–r31 only**	`ser r17`
`CLR rd`	clear all bits — `rd` ← 0. Same as `eor rd, rd`, so **any register**	`clr r3`
`SUB rd, rr`	`rd` ← `rd` − `rr`	`sub r7, r19`
`COM rd`	**one's complement** — flip all bits	`com r8`
`NEG rd`	**two's complement** — negate	`neg r9`
`ANDI rd, K`	bitwise AND with a constant — **r16–r31 only**	`andi r16, 0x01`
`ORI rd, K`	bitwise OR with a constant — **r16–r31 only**	`ori r16, 0xF0`

`MOV` is named misleadingly: it *copies*, it does not move. The source register is unchanged. It also affects **no flags**, which is what makes it safe to shuffle values between operations without disturbing the status register.

`SER` and `CLR` are short-hand ways of setting/clearing all bits of a register, and their restrictions differ because their expansions do:

	Exactly the same as	Reaches
`ser rd` (set register)	`ldi rd, 0xFF`	**r16–r31 only** — it inherits `LDI`'s four-bit register field
`clr rd` (clear register)	`eor rd, rd`	**any register**, r0–r31 — `EOR` is a two-register instruction with five bits for each

So `clr r3` is legal and `ser r3` is not, which looks arbitrary until you look at what each one

assembles to. `clr` is also the reason $X \oplus X = 0$ from bit-masking[98] is worth remembering: XORing a register with itself is how the machine zeroes one.

Compare instructions

These do the arithmetic, **discard the result, and keep only the flags** — which is exactly what a conditional branch needs.

Instruction	Does	Note
`CP rd, rr`	`rd` − `rr`	compares two registers
`CPI rd, K`	`rd` − K	compares against a constant — like `ldi`, **r16–r31 only**
`TST rd`	`rd` AND `rd`	examines one register; literally the same operation as `and rd, rd`

`tst` looks odd until you notice that ANDing a value with itself leaves it unchanged ($X \cdot X = X$) but still sets N and Z from it — so it is a zero/sign test that costs nothing and clobbers nothing. See bit-masking[99].

A comparison is **the same as subtraction**, with the **result not saved** — it exists only to **set status register flags (Z, N, C, V ...)**. And **the instruction reference specifies which status register bits are set by which instructions**: the flags are defined per instruction rather than falling out of one shared adder, which is the point of the table at the end of this note.

Branch instructions

Based on status register values — the flags a compare just set:

	Branches to `label` if
`BREQ label`	**Z = 1**
`BRNE label`	**Z = 0**

Together with `CP` that is a complete conditional: compare, which sets Z exactly when the two operands were equal, then branch on Z.

[98] courses/csse2010/bit-masking.pdf
[99] courses/csse2010/bit-masking.pdf

```
    cp   r5, r6          ; r5 - r6, result thrown away, flags kept
    breq same            ; Z = 1, i.e. r5 == r6
```

`JMP label` is the unconditional version — it tests nothing and always goes. Labels, and the program skeleton the branches live in, are in avr-program-structure[100].

`breq`/`brne` are two of ten: there is one branch per status flag per value, and the pairs that matter for *ordering* are `brsh`/`brlo` (unsigned, on C) and `brge`/`brlt` (signed, on S) — the full table, and why the signed pair reads S rather than N, are in avr-branches-and-jumps[101].

Control flow

The instructions that write the PC, and the two stack instructions that go with the calls (avr-branches-and-jumps[102], procedures-and-the-stack[103]):

Instruction	Does	Words	Cycles
`BRxx label`	branch if a flag has the wanted value; reach ±64 words	1	1 (not taken) / 2 (taken)
`JMP label`	PC ← address of `label`	2	3
`RJMP label`	PC ← PC + offset; reach ±2 K words	1	2
`IJMP`	PC ← Z	1	2
`CALL label`	push the return address, then as `JMP`	2	4
`RCALL label`	push the return address, then as `RJMP`	1	3
`ICALL`	push the return address, then as `IJMP`	1	3
`RET`	pop the return address into PC	1	4
`PUSH rr`	store `rr` at SP, SP ← SP − 1	1	2
`POP rd`	SP ← SP + 1, load `rd` from SP	1	2

[100] courses/csse2010/avr-program-structure.pdf
[101] courses/csse2010/avr-branches-and-jumps.pdf
[102] courses/csse2010/avr-branches-and-jumps.pdf
[103] courses/csse2010/procedures-and-the-stack.pdf

None of these touches the status register except by what the pushed and popped values happen to be; a `call` between a `cp` and its branch is safe only if the procedure preserves the flags, which in practice means it is not.

Three more belong to interrupts, and between them they are the only instructions that change the **I** bit (avr-interrupts[104]):

Instruction	Does	Words	Cycles
`SEI`	I ← 1: interrupts enabled	1	1
`CLI`	I ← 0: interrupts disabled	1	1
`RETI`	pop the return address into PC, and I ← 1	1	4

`RETI` is `RET` plus *set I*. A handler that ends in `RET` returns to the right place and leaves interrupts off for good.

Data movement

Data is usually copied, not moved — in every instruction below the source is left alone. What changes is *where* the two ends are:

Direction	Called	AVR
register → register	**move**	`mov`
memory → register	**load**	`lds`, `ld`
register → memory	**store**	`sts`, `st`
I/O register → register	**input**	`in`
register → I/O register	**output**	`out`

Three different places a value can live — a general purpose register, data memory, an I/O register — and a different instruction family for each. **The one thing that is genuinely out of reach is SRAM from `in`/`out`**: `P` only goes up to `0x3F`, so those two never leave the first 64 I/O registers.

Everything else is a question of **cost and numbering, not reachability**. Because the I/O registers are mapped into the data space (avr-io-ports[105]), `lds`/`sts` *can* reach one — `in r16, 0x09` and `lds r16, 0x29` both read `PIND`, the second for two instruction words instead of one. The bug to actually watch for is writing the I/O register number where the data address belongs, or the reverse.

[104] courses/csse2010/avr-interrupts.pdf
[105] courses/csse2010/avr-io-ports.pdf

Load and store

	Does	Addressing	Words	Cycles
`LD rd, X`	`rd` ← `Mem[X]`	**indirect** — address in the pointer register	1	2
`ST X, rr`	`Mem[X]` ← `rr`	indirect	1	2
`LDS rd, k`	`rd` ← `Mem[k]`	**direct** — 16-bit address in the instruction	**2** (32 bits)	2
`STS k, rr`	`Mem[k]` ← `rr`	direct	**2**	2

For example:

```
    ld   r5, X          ; r5 <- Mem[X]
    st   X, r5          ; Mem[X] <- r5
    lds  r5, 0x20       ; r5 <- Mem[0x20]
    sts  0x20, r5       ; Mem[0x20] <- r5
```

Both pairs reach the same data space — the difference is only where the address comes from (addressing-modes[106]), and `lds/sts` pay a **second instruction word** for spelling it out. X here is the pointer pair of avr-architecture[107].

In and out

	Does	Words	Cycles
`IN rd, P`	`rd` ← I/O register `P`	1	1
`OUT P, rr`	I/O register `P` ← `rr`	1	1

`rd/rr` may be **any** of r0–r31 — unlike `ldi`, these are not restricted to the top half — and `P` is an I/O register number 0–63 (`0x00`–`0x3F`) or a name from `m324Adef.inc`. Examples: `in r5, 0x26` (I/O register `0x26` is `TCNT0`) and `out PORTD, r6`. Full treatment in avr-io-ports[108].

One word and one cycle makes these the cheapest access on the chip — cheaper than reaching the very same register through `lds/sts`, which is the whole reason the first 64 I/O registers are special.

[106] courses/csse2010/addressing-modes.pdf
[107] courses/csse2010/avr-architecture.pdf
[108] courses/csse2010/avr-io-ports.pdf

Shifts and rotates

Monadic instructions that move every bit one place. They differ only in **what arrives at the vacated end** and **what happens to the bit that falls off** — and in all five, the bit that falls off lands in **C**.

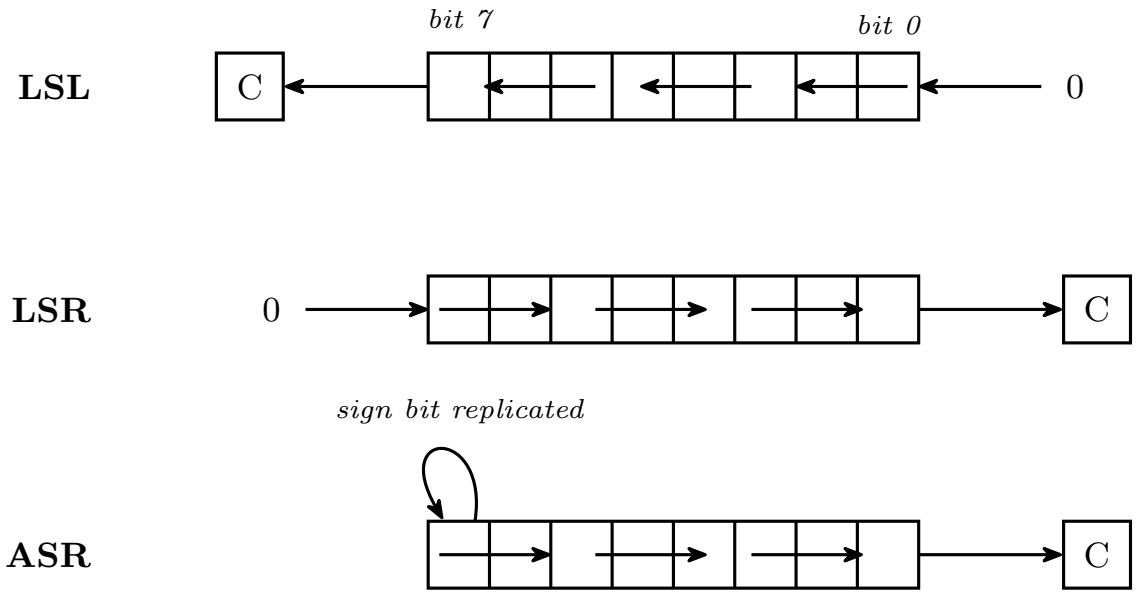


Figure 15: The three shift instructions. LSL and LSR feed a 0 into the vacated end; ASR feeds the sign bit back into itself, which is what makes it a signed division. In all three the bit shifted out lands in the carry flag.

	Name	Into the vacated end	Out
`LSL rd`	**L**ogical **S**hift **L**eft	`0` → bit 0	bit 7 → C
`LSR rd`	**L**ogical **S**hift **R**ight	`0` → bit 7	bit 0 → C
`ASR rd`	**A**rithmetic **S**hift **R**ight	bit 7 **replicated** into itself	bit 0 → C
`ROL rd`	**RO**tate **L**eft through carry	C → bit 0	bit 7 → C
`ROR rd`	**RO**tate **R**ight through carry	C → bit 7	bit 0 → C

What they compute

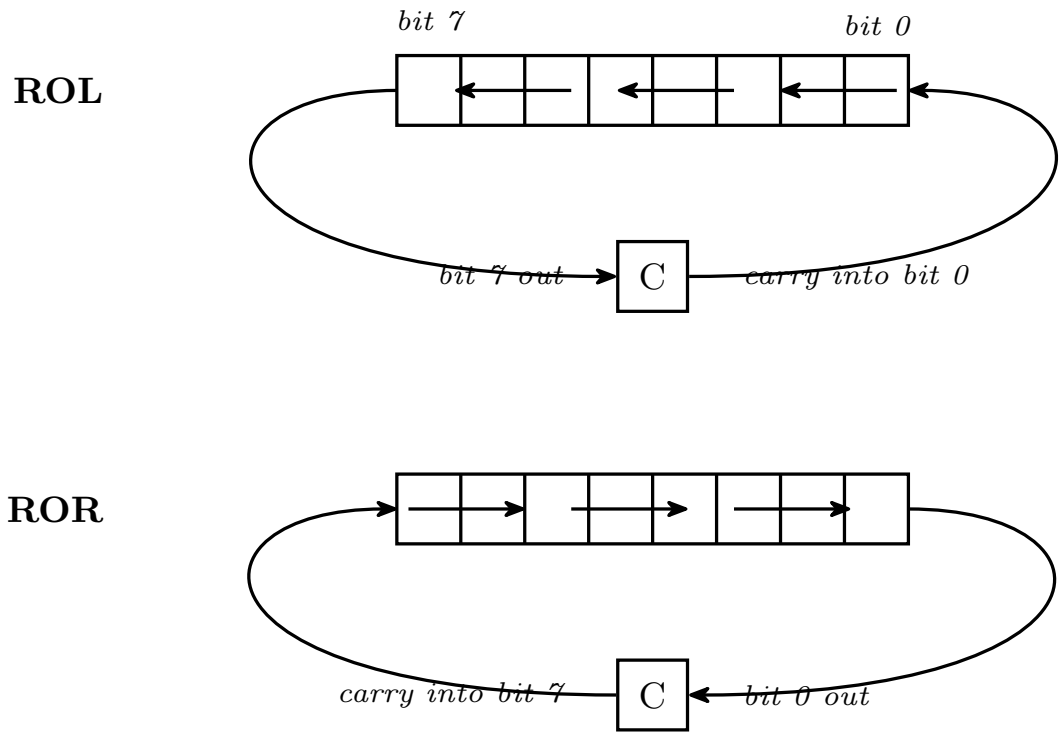


Figure 16: The two rotate instructions. A rotate is a shift whose vacated end is fed from the carry flag instead of a constant, so the register and C together behave as a nine-bit ring.

Shifting a binary number one place is multiplying or dividing it by two, which is the whole reason these exist. Using four bits for legibility:

		Worked example
`LSL`	**× 2**	`0010` (2) → `0100` (4) → `1000` (8)
`LSR`	**÷ 2, read unsigned**	`0110` (6) → `0011` (3); `1000` (8 unsigned) → `0100` (4)
`ASR`	**÷ 2, read signed**	`1000` (−8) → `1100` (−4)

LSR and ASR are the pair to keep straight. Given `1000`, LSR produces `0100` — right if the value meant unsigned 8, wrong if it meant −8. ASR replicates the sign bit into the vacated position and produces `1100` = −4. The same bits go in; **the instruction is where you declare which interpretation you meant** (binary-number-representations[109]).

There is no matching pair going the other way, because **left shifts need no such distinction**: a 0 always enters at the bottom, and the signed and unsigned readings of "× 2" agree — right up until the result stops fitting in eight bits.

Shifting a value wider than eight bits

Rotates exist to carry a bit from one byte into the next. To multiply the **16-bit unsigned** value in `r13:r14` by two (so `r13` is the high byte, `r14` the low):

```
    lsl  r14            ; low byte x2; bit 7 falls out into C
    rol  r13            ; high byte x2, pulling C into its bit 0
```

`LSL` on the low byte pushes its top bit into C; `ROL` on the high byte pulls exactly that bit back in at the bottom. The two instructions together shift a sixteen-bit quantity one place.

The wrong answers are worth naming, because each fails differently. `lsl r13` (and `asl r13`, which is the same operation under another name — shifting left, a 0 always enters) shifts the high byte but feeds it a **0**, discarding the bit the low byte just produced. `rol r14` rotates the **wrong byte**.

[109] courses/csse2010/binary-number-representations.pdf

Building wider operations from 8-bit ones

The AVR supports exactly **one data type in hardware — an 8-bit integer** (instruction-set-architecture[110]). Everything wider is a sequence of instructions, and **C is what stitches the pieces together**.

Add the 16-bit quantity `r5:r4` to `r11:r10`, result in `r11:r10` — high byte named first, so `r4` and `r10` are the low bytes:

```
    add  r10, r4          ; r10 <- r10 + r4        (low bytes; carry out lands in C)
    adc  r11, r5          ; r11 <- r11 + r5 + C    (high bytes, plus that carry)
```

`ADD` on the low bytes leaves any carry out of bit 7 in C, and `ADC` on the high bytes adds it back in — the same column-by-column carrying as long addition on paper, one byte per column.

The order is not negotiable, and neither is what sits between the two lines: `adc` has to run *after* the `add` that produced the carry, and any instruction in between that touches C ruins the result. That is the general shape of multi-byte arithmetic on an 8-bit machine — low end first, and nothing allowed to disturb the flag in the middle.

Constants

Constants do not have to be hexadecimal. All five of these load the same value into `r17`:

Written	Radix	
`ldi r17, 0x34`	hexadecimal	`0x` prefix
`ldi r17, $34`	hexadecimal	`$` prefix
`ldi r17, 52`	decimal	no prefix
`ldi r17, 064`	**octal**	a leading `0`
`ldi r17, 0b00110100`	binary	`0b` prefix

`0x34` = 52 = 64_8 = `0b00110100`. The octal form is the trap: **a leading zero changes the radix**, so `012` is 10, not 12.

[110] courses/csse2010/instruction-set-architecture.pdf

LDI's restrictions

`LDI` is the one with strings attached:

- **Registers r16 to r31 only.** Not r0–r15.
- **8-bit numbers**, `00000000` to `11111111`. Written in decimal the range is −128 to 255 — that span covers both the signed and the unsigned reading of eight bits.

A counter-example, and the legal version of it:

```
    ldi  r0, 5          ; NOT legal - r0 is below r16
    ldi  r16, 5         ; the same idea, in a register ldi can reach
```

The register restriction is not arbitrary: it falls straight out of the encoding below, which has only four bits for the register number.

Machine code

Each instruction is a 16-bit word made of an **opcode** and **operand** bit fields — see instruction-set-architecture[111]. In the AVR datasheet's notation, a format like `0000 11rd dddd rrrr` means the literal bits are fixed and each letter is one bit of an operand, gathered from wherever it appears.

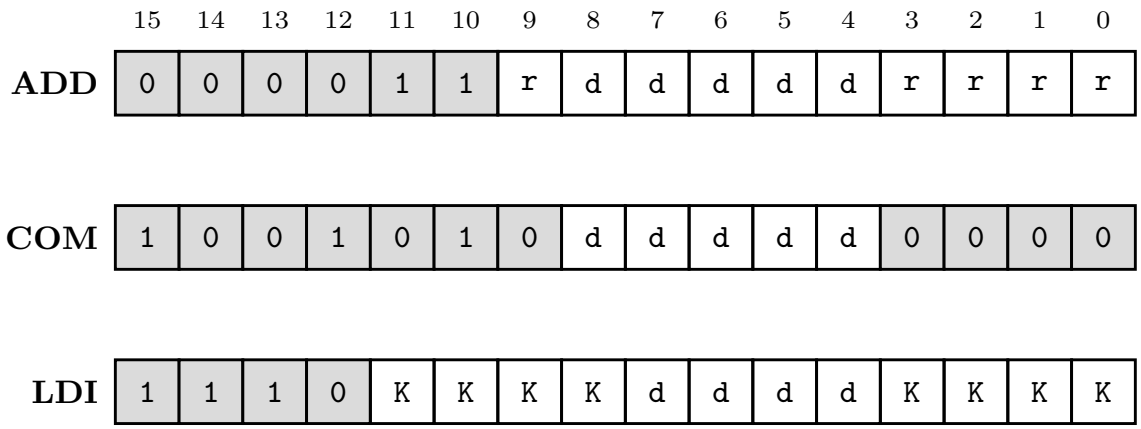


Figure 17: Three AVR instruction encodings, each 16 bits. Fixed opcode bits are shaded; letters are operand bits. In ADD and LDI the operand bits are not contiguous — the register number's top bit sits apart from the rest, and LDI's constant is split into two nibbles either side of the register field.

[111] courses/csse2010/instruction-set-architecture.pdf

ADD — `0000 11rd dddd rrrr`

The **opcode is bits 15–10**, `0000 11` — six fixed bits, all that is left once the two five-bit register numbers have taken ten of the sixteen. The destination register number is **`d dddd`** — five bits, so all 32 registers — and the source is **`r rrrr`**. Both have their top bit separated from the other four, and note the order in the middle nibble: the *source*'s top bit (`r`, bit 9) comes before the *destination*'s (`d`, bit 8), even though the mnemonic writes the destination first.

Worked example: `add r23, r7`.

	Register	In binary (5 bits)	Split as
`rd`	r23 = 16 + 4 + 2 + 1	`10111`	`d` = 1, `dddd` = `0111`
`rr`	r7 = 4 + 2 + 1	`00111`	`r` = 0, `rrrr` = `0111`

Dropping those into `0000 11rd dddd rrrr`:

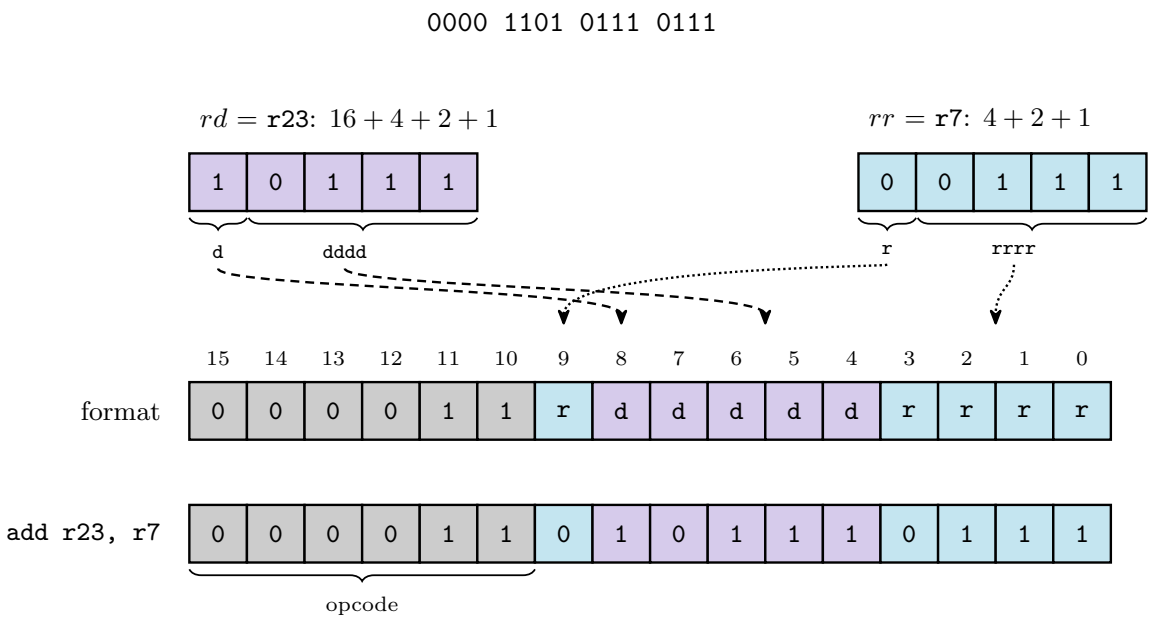


Figure 18: Assembling `add r23, r7`. Each register number is written as five bits, then torn in two: the top bit goes into the middle nibble and the low four into their own field. Shading, not position, is what tells the two operands apart here. The two guide lines cross because the destination's bit sits between the source's two pieces.

COM — `1001 010d dddd 0000`

Flip all bits — one's complement. One register operand, again as `d dddd`.

Worked example: `com r19`. r19 = 16 + 2 + 1 = `10011`, so `d` = 1 and `dddd` = `0011`:

```
1001 0101 0011 0000
```

LDI — `1110 KKKK dddd KKKK`

Two operand fields:

- `K` — the 8-bit constant, **split into two nibbles**: the high four bits in the first `KKKK` field, the low four in the last.
- `d` — the **register number minus 16**.

That subtraction is why `LDI` only reaches r16–r31: the field is four bits, so it can express $2^4 = 16$ distinct registers, and the ISA spends them on the top half of the register file.

Applying the rule to `ldi r16, 73`: $d = 16 - 16 = 0$, so `dddd` = `0000`; 73 = `0100 1001`, so the high nibble is `0100` and the low nibble `1001`:

```
1110 0100 0000 1001
```

Flags, by instruction

The reference's *Flags Affected* column is worth reading rather than assuming — the flags are **defined per instruction**, not derived from a single adder. For the instructions above:

Instruction	Z	C	N	V
`MOV`, `LDI`, `SER`	—	—	—	—
`AND`, `ANDI`, `OR`, `ORI`, `EOR`, `TST`	from result	**unchanged**	from result	**cleared to 0**
`ADD`, `ADC`, `SUB`, `CP`, `CPI`	from result	from carry/borrow	from result	signed overflow
`CLR`	**always 1**	**unchanged**	**always 0**	**cleared to 0**
`COM`	from result	**always set to 1**	from result	cleared to 0
`NEG`	from result	**set unless the result is `$00`**	from result	set only if the result is `$80`

Two of these will catch you out. The **logical operations leave C alone entirely** — whatever it held before an `and` it still holds after. And **`COM` sets C unconditionally**, which is not a

carry out of anything; it is a defined convention, chosen so that `com` followed by an increment behaves like a negation.

The `CLR` row is just the logical row with a result that is guaranteed to be `0x00` — `clr rd` *is* `eor rd, rd` — which makes it the one instruction whose flags you can predict without knowing the operand.

For the shift, rotate and branch instructions, look the flags up in the reference rather than guessing. The one part that is visible in the diagrams above is C: every shift and rotate sets it from the bit that fell out. The rest is defined per instruction, as the paragraph opening this section warns.

For `ADD` specifically, the manual gives the flags as Boolean formulas over the operand and result bits, writing R for the result (which equals `Rd` after the operation):

$$N = R_7 \qquad Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$$

$$V = Rd_7 \cdot Rr_7 \cdot \overline{R_7} + \overline{Rd_7} \cdot \overline{Rr_7} \cdot R_7 \qquad C = Rd_7 \cdot Rr_7 + Rr_7 \cdot \overline{R_7} + \overline{R_7} \cdot Rd_7$$

Z is a NOR over every result bit. V says overflow happened exactly when **both operands had the same sign and the result's differs** — two negatives giving a positive, or two positives giving a negative. And $S = N \oplus V$, for signed tests. See avr-architecture[112].

See also

- instruction-set-architecture[113] · avr-architecture[114] · bit-masking[115]
- avr-io-ports[116] — the I/O registers `in`/`out` reach
- addressing-modes[117] — the four ways an operand names its value
- avr-program-structure[118] — labels, `jmp` and the program skeleton
- avr-interrupts[119] — `sei`, `cli`, `reti`, and which flags an interrupt handler must save
- binary-number-representations[120] — one's and two's complement, and the −128 to 255 range
- 2026-08-27-avr-introduction[121] — Lecture 10
- 2026-09-03-instruction-set-architecture-part-2[122] — Lecture 12

[112] courses/csse2010/avr-architecture.pdf
[113] courses/csse2010/instruction-set-architecture.pdf
[114] courses/csse2010/avr-architecture.pdf
[115] courses/csse2010/bit-masking.pdf
[116] courses/csse2010/avr-io-ports.pdf
[117] courses/csse2010/addressing-modes.pdf
[118] courses/csse2010/avr-program-structure.pdf
[119] courses/csse2010/avr-interrupts.pdf
[120] courses/csse2010/binary-number-representations.pdf
[121] courses/csse2010/2026-08-27-avr-introduction.pdf
[122] courses/csse2010/2026-09-03-instruction-set-architecture-part-2.pdf

AVR I/O Ports

How an AVR program actually reads a switch and lights an LED: the I/O register file, the three registers behind every port, the two instructions that reach them, how to change one bit of a port without disturbing the others, what writing to `PINx` really does, and the pins' alternate uses by the serial interfaces. See avr-architecture[123] for the chip these live on, avr-instructions[124] for the instruction set, and io-board[125] for the devices that get wired to the pins.

I/O registers

The **ATmega324A has 224 I/O registers** — registers that control the peripherals and carry data to and from them:

- Timers and counters
- Analog to digital converters
- Serial input/output
- General purpose input/output ports

They are **not** the general purpose registers. Some instructions work with these, others with r0–r31, and **confusing the two is the standard beginner error** — `ldi r16, 0xFF` and `out DDRA, r16` name completely different register files with the same-looking numbers.

Some addresses are reserved (not actually available on this device). The full list is the *I/O register summary*, pages **636 to 639** of the datasheet.

Where they live, and how you get at them

I/O registers 0 to 223 sit in the **data** address space at **memory addresses 32 to 255** (`0x20` to `0xFF`) — see memory-models[126] for why the AVR has a separate data space at all. That gives two ways to name the same register, and two names for it:

	Range	Reached by
First 64 I/O registers (0 to 63)	data addresses `0x20`–`0x5F`	**`in` and `out`** *and* `ld`/`st`
Other 160 — the **"extended" I/O registers** (64 to 223)	data addresses `0x60`–`0xFF`	**`ld`/`st` only**

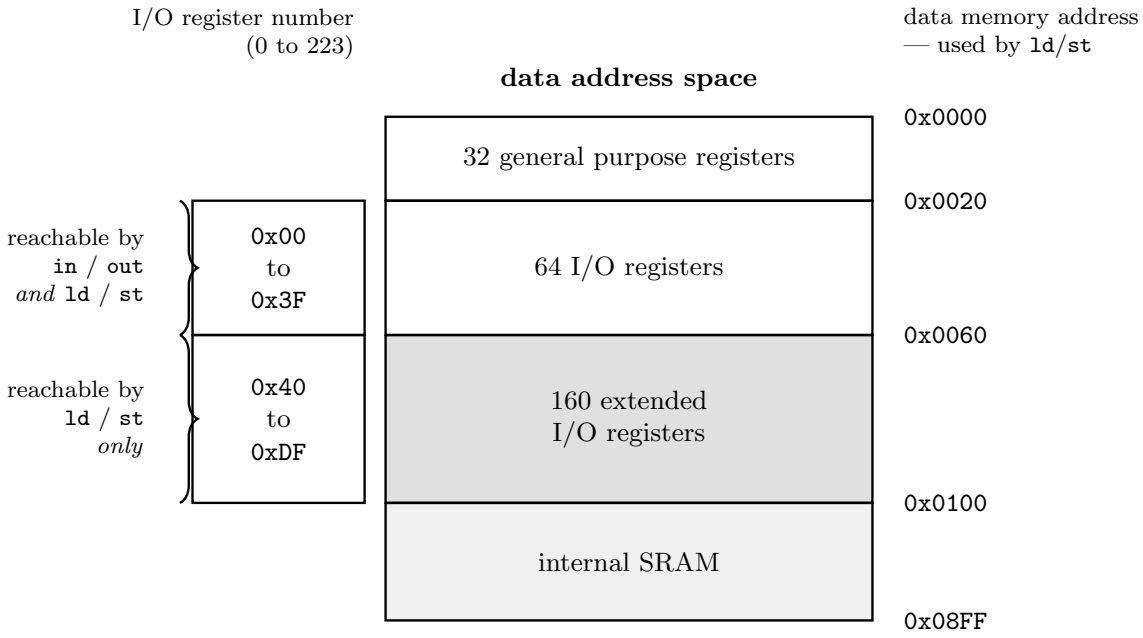
[123] courses/csse2010/avr-architecture.pdf
[124] courses/csse2010/avr-instructions.pdf
[125] courses/csse2010/io-board.pdf
[126] courses/csse2010/memory-models.pdf

so the two numbering systems differ by a constant:

$$\text{data memory address} = \text{I/O register number} + \texttt{0x20}$$

The offset is exactly the 32 general purpose registers, which occupy `0x0000`–`0x001F` at the bottom of the data space ahead of them.

Why the split exists at all: `in`/`out` encode the I/O register number in **six bits** of the instruction word, and $2^6 = 64$. Everything past register 63 has no room in the encoding, so it can only be reached the long way round, by its data address.



data address = I/O register number + `0x20` (the 32 registers in front of them)

Figure 19: The bottom of the ATmega324A data address space, and the two numbering systems that live on it. Every I/O register has **two** numbers — its I/O register number (what `in`/`out` take) and its data memory address (what `ld`/`st` take) — and they differ by a constant `0x20`, because the 32 general purpose registers sit in front of them in the same space. Only the first 64 I/O registers have an I/O register number small enough to fit the six operand bits of `in`/`out`; the 160 extended ones can be reached by their data address alone. The map follows the convention of csse2010-diagram-notation[127]: lowest address at the top, hex addresses outside.

[127] courses/csse2010/csse2010-diagram-notation.pdf

Extract from the I/O register list

The general purpose port registers are the first twelve entries, in the order PIN, DDR, PORT for each port in turn:

Data address	I/O reg number	Name
`0x2B`	`0x0B`	PORTD
`0x2A`	`0x0A`	DDRD
`0x29`	`0x09`	PIND
`0x28`	`0x08`	PORTC
`0x27`	`0x07`	DDRC
`0x26`	`0x06`	PINC
`0x25`	`0x05`	PORTB
`0x24`	`0x04`	DDRB
`0x23`	`0x03`	PINB
`0x22`	`0x02`	PORTA
`0x21`	`0x01`	DDRA
`0x20`	`0x00`	PINA

Further up the same list, to show the pattern continues and that reserved entries are real:

Data address	I/O reg number	Name
`0x38`	`0x18`	TIFR3
`0x37`	`0x17`	TIFR2
`0x36`	`0x16`	TIFR1
`0x35`	`0x15`	TIFR0
`0x34`	`0x14`	*Reserved*

Read the two columns carefully in an exam question. `out 0x02, r16` writes PORTA, but `sts 0x02, r16` writes general purpose register r2 — the same number, two address spaces, and only the instruction tells them apart.

GPIO ports

The ATmega324A has **four 8-bit general purpose I/O ports — PORT A, PORT B, PORT C, PORT D** — appearing on the datasheet block diagram as `PA7..0`, `PB7..0`, `PD7..0` around the edge of the chip, with port C drawn as `PC5..0` plus `TOSC1/PC6` and `TOSC2/PC7` broken out separately — those two pins double as the timer oscillator connections. They are still port C bits, and `DDRC/PORTC/PINC` still cover all eight.

Three registers are associated with each (x = A, B, C or D):

Register	Name	What it holds
DDRx	Data Direction Register	which pins are outputs
PORTx		values to output
PINx		values on the pins

Each is 8 bits — **one bit per pin**. `DDRA0`, `PORTA0` and `PINA0` are the three bits belonging to pin PA0.

What one bit of a port looks like

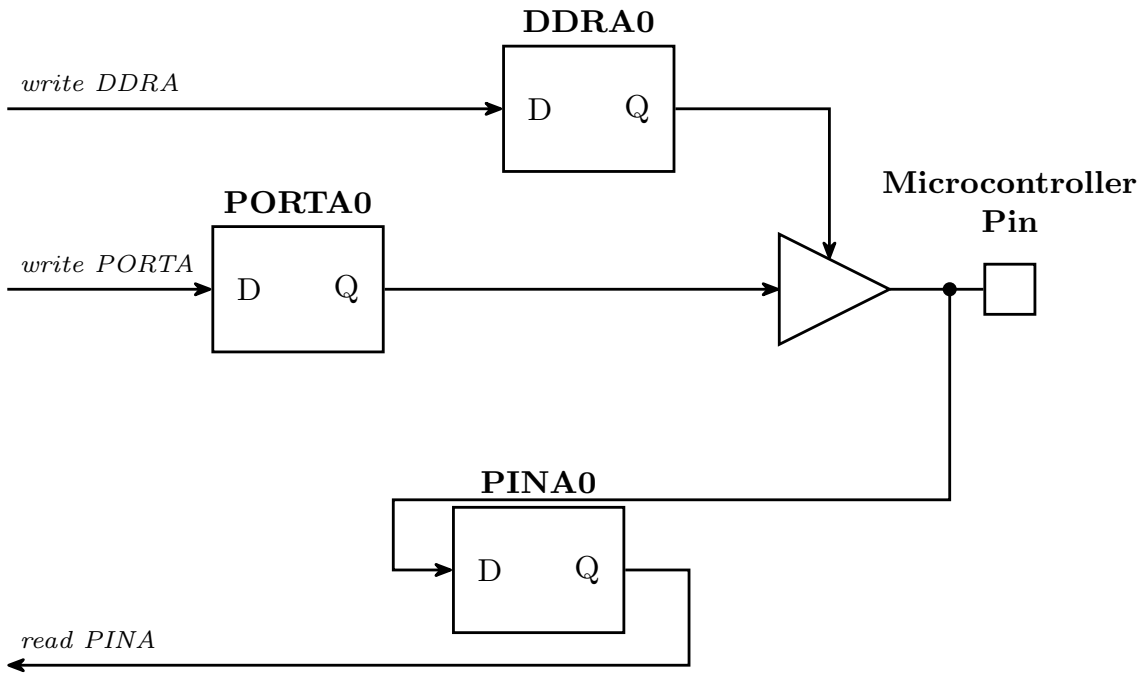


Figure 20: One bit of an AVR port — the three flip-flops behind pin PA0. DDRA0 enables the tri-state buffer, so it decides whether PORTA0's value is driven onto the pin or the pin is left free for something outside to drive. PINA0 samples whatever is on the pin either way. Hence: DDR is direction, PORT is what you write, PIN is what you read.

1 means output, 0 means input. Writing `0xFF` to DDRA makes every bit of port A an *output*, and writing `0x00` to DDRB makes every bit of port B an *input*.

Three consequences fall straight out of the circuit:

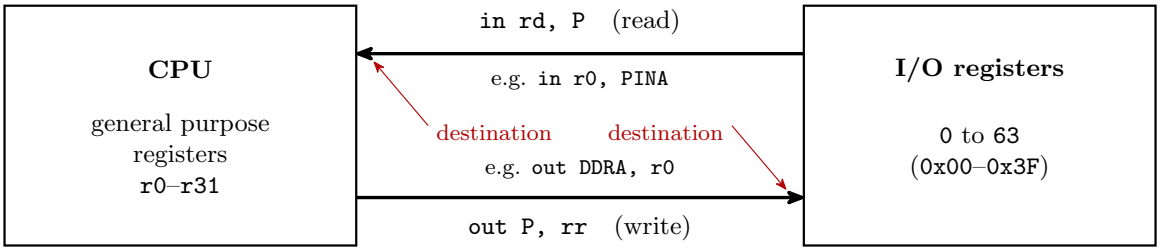
- **PORTx does nothing on a pin whose DDRx bit is 0.** The buffer is off; the value sits in the flip-flop unused. (On real AVR hardware it switches an internal pull-up instead.)
- **PINx is always live.** It samples the pin whether the pin is being driven from inside or outside, so reading PINx on an output pin reads back what you are driving.
- **Reading PORTx is not reading the pin.** It gives you the value you last wrote. To find out what is *there*, read PINx — which is why every input example uses `in rd, PINx` and never `in rd, PORTx`.

in and out

The two instructions that reach the first 64 I/O registers:

	Does	Operands
`in rd, P`	Load I/O register value into general purpose register	`rd`: r0 to r31. `P` = I/O register number (0 to 63, `0x00` to `0x3F`) or name
`out P, rr`	Store value from general purpose register into I/O register	`rr`: r0 to r31. `P` = 0 to 63 (`0x00` to `0x3F`) or name

The operand order flips between them, and it flips for a consistent reason: like every AVR instruction, the **destination is written first**. So the I/O register is the *second* operand of `in` and the *first* of `out`.



The arrowhead is the destination, and **the destination is always the first operand** — which is why the I/O register is the *second* operand of `in` and the *first* of `out`.

Figure 21: `in` and `out` between the two register files: the CPU's general purpose registers `r0`–`r31` and the first 64 I/O registers. `in` reads *from* an I/O register into a general purpose register, `out` writes the other way, and the head of each arrow, marked *destination*, is the operand written first. That is why the I/O register is the second operand of `in` and the first of `out`.

Answers of the form `in DDRB, r16` are the standard trap. What is wrong with them is the **mnemonic**, not the direction: `in`'s destination is always one of r0–r31, so it can never be `DDRB`. Moving a general purpose register into an I/O register is exactly what `out` does — which is why the right answer in that shape is `out DDRB, r16`.

`in` and `out` reach all 32 general purpose registers, r0–r31 — unlike `ldi`, they are not restricted to the top half.

Both are **one instruction word and one clock cycle**, which makes them the cheapest way to touch a peripheral.

Three equivalences, all of them just the number/name table above:

Written with a number	Same instruction, written with a name
`in r0, 0`	`in r0, PINA`
`out 1, r0`	`out DDRA, r0`
`in r5, 0x26`	`in r5, TCNT0` — I/O register `0x26` is timer/counter 0

Using port names

Numbers are how the machine sees it; names are how you should write it:

```
.include "m324Adef.inc"
```

The contents of this file are included directly in the program, and it defines the names for the I/O registers — so `DDRA`, `PORTA`, `PINA`, `TCNT0` and the rest can be written instead of `0x01`, `0x02`, `0x00`, `0x26`. Everything about the generated machine code is identical; the assembler substitutes the number.

Setting port A (all bits) to be an output, which is the shape of nearly every initialisation block:

```
    ldi  r16, 0xFF       ; every bit 1 = output
    out  DDRA, r16       ; ... into the direction register
```

or, using the shorthand of avr-instructions[128], `ser r16` in place of the `ldi`. Setting a port to be all *inputs* is the same two lines with `0x00` (or `clr`), and a mixed port is any other constant — DDRA is a bit pattern like everything else, so `ldi r16, 0x0F` gives four outputs and four inputs.

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

Changing one bit of a port

Writing a whole port register is right at initialisation. Afterwards a program usually wants to change **one pin and leave the other seven alone**, because the other pins belong to other parts of the program. In C that is a read-modify-write with a one-bit mask, the bit named rather than given as a number (bit-masking[129]):

```
DDRC  |= (1<<5);          /* PC5 becomes an output     */
PORTC |= (1<<5);          /* PC5 driven high           */
PORTC &= ~(1<<5);         /* PC5 driven low            */
PORTC ^= (1<<5);          /* PC5 flipped               */
if (PINC & (1<<2)) { }    /* is PC2 high?              */
```

In assembly there are two ways:

Register	Set or clear one bit	
I/O registers 0 to 31, which includes every `PINx`, `DDRx` and `PORTx`	`sbi PORTC, 5` and `cbi PORTC, 5`	one instruction, other bits untouched
any register `in`/`out` reach, 0 to 63	`in r16, TCCR0B`, then `ori` or `andi` with the mask, then `out TCCR0B, r16`	three instructions, and a spare register

`sbi` and `cbi` hold the register number in five bits of the instruction, so they reach only the first 32 I/O registers. `TCCR0B` is I/O register `0x25`, 37, which is past them: the long way is the only way. The extended registers past 63, such as `TIMSK2` and `TCCR1B`, need `lds` and `sts` in place of `in` and `out`.

Writing to `PINx`

`PINx` is the port's input register: reading it gives the levels on the pins. Writing to it is not storing a value. On the ATmega324A, **writing a 1 to a bit of `PINx` toggles the matching bit of `PORTx`**, whatever the pin's direction, and writing a 0 does nothing.

That makes `PINx` a quick way to flip a pin:

```
PINC = (1<<0);            /* toggle PC0; the 0s written to the other bits do nothing
 ↪  */
```

[129] courses/csse2010/bit-masking.pdf

and in assembly `sbi PINC, 0` does the same in one instruction.

It also makes `PINx` the **wrong register for a clear or a set**. `PINC &= ~(1<<5)` does not clear PC5. If the compiler turns it into a single `cbi PINC, 5`, it writes a 0 to bit 5 of `PINC` and nothing happens. If it does a read, an AND and a write, it writes 1s to every other bit that read high, and those pins flip:

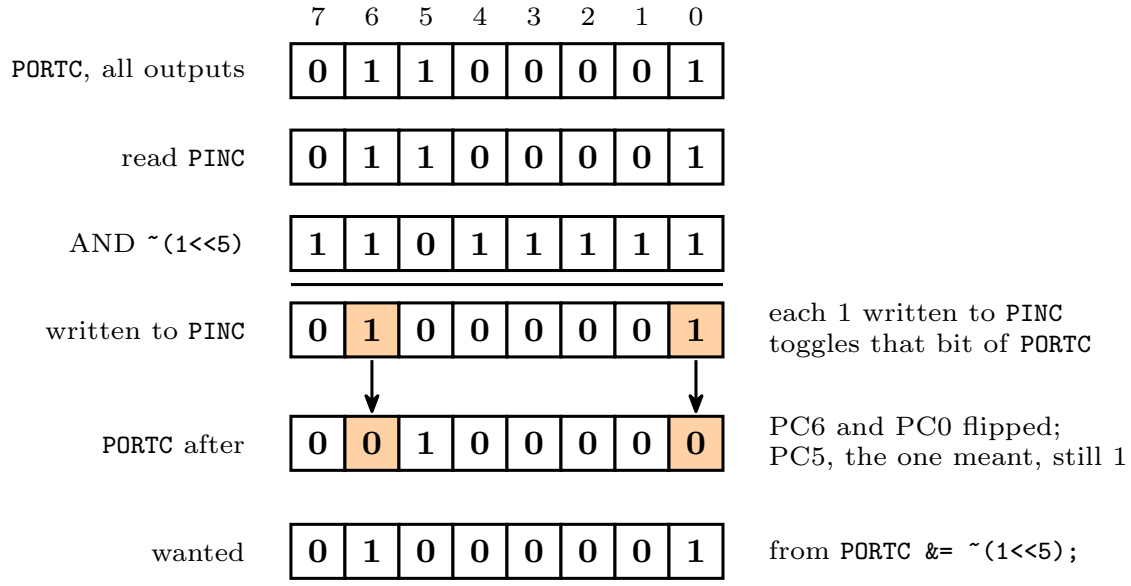


Figure 22: Why a read-modify-write on `PINC` goes wrong. Port C is all outputs, driving 0110 0001, so its pins read the same. `PINC &= ~(1<<5)` reads them, clears bit 5 of the reading and writes the result to `PINC`. A 1 written to a `PINx` bit is an order to toggle that bit of `PORTx`, not a value to keep, and a 0 does nothing. So the two other 1s, shaded, flip PC6 and PC0 low, and PC5, the bit meant to be cleared, stays high. The bottom row is what `PORTC &= ~(1<<5)` would have given.

So: **set and clear through `PORTx`, read through `PINx`, and write `PINx` only to toggle.**

Alternate pin functions

Most pins have a second job. When the peripheral that owns a pin is switched on, it takes the pin over from the port, and for some pins it overrides `DDRx` too. The serial interfaces use these:

Port pin	Package pin	Alternate use	Peripheral
PB0	1	XCK0	USART0's clock, used when it is an SPI master

Port pin	Package pin	Alternate use	Peripheral
PB4	5	$\overline{\text{SS}}$	SPI (avr-spi[130])
PB5	6	MOSI	SPI
PB6	7	MISO	SPI
PB7	8	SCK	SPI
PC0	22	SCL	TWI
PC1	23	SDA	TWI
PD0	14	RXD0	USART0 (avr-usart[131])
PD1	15	TXD0	USART0
PD2	16	RXD1	USART1
PD3	17	TXD1	USART1
PD4	18	XCK1	USART1's clock, used when it is an SPI master

So the SPI port is the upper half of port B, and the TWI is the bottom two bits of port C. Other pins carry timer outputs, the external interrupts (`INT0` and `INT1` share PD2 and PD3 with USART1) and, on port A, the analog inputs. A pin can do only one job at a time, so a program that uses the SPI cannot also use PB4 to PB7 as general purpose I/O. The whole pinout is drawn in 2026-10-05-more-io[132].

Common mistakes

- **`in DDRB, r16`.** `in` always writes a general purpose register. Writing an I/O register is `out DDRB, r16`.
- **Reading `PORTx` to find out what is on a pin.** That gives back what was last written. The pins are `PINx`.
- **Setting or clearing a bit through `PINx`.** A 1 written there toggles, so `PINC &= ~(1<<5)` flips other pins and leaves PC5 alone.
- **`PORTC = (1<<5)` to set one pin.** `=` writes all eight bits and drives the other seven low. Use `|=`.
- **`sbi` or `cbi` on a register past 31**, such as `TCCR0B`. They cannot reach it; use `in`, `ori`/`andi`, `out`.
- **Using a pin as general I/O while its peripheral is on.** The SPI, the TWI and the USARTs take their pins over.

[130] courses/csse2010/avr-spi.pdf
[131] courses/csse2010/avr-usart.pdf
[132] courses/csse2010/2026-10-05-more-io.pdf

See also

- avr-architecture[133] — the data address space these sit in, and the rest of the chip
- avr-instructions[134] · avr-program-structure[135] · bit-masking[136]
- memory-models[137] — why the I/O registers have two addresses
- io-board[138] · device-pinouts[139] — the switches, LEDs and buttons wired to the pins
- week6-lab-avr-io-ports[140] — Lab 10, where this gets wired up
- avr-usart[141] — PD0 and PD1 as a serial port · analog-to-digital-conversion[142] — port A as analog inputs
- avr-spi[143] — PB4 to PB7 as the SPI port · serial-communication[144] — the TWI on PC0 and PC1
- 2026-08-31-instruction-set-architecture-part-1[145] — Lecture 11 · 2026-10-05-more-io[146] — Lecture 19

Binary Number Representations

Bits, bytes, and words

- **Bit** = binary digit (0 or 1).
- **Nibble** = 4 bits. It is the unit that makes hexadecimal convenient (one hex digit = one nibble).
- **Byte** = 8 bits, e.g. `01010111`.
- Modern computers deal with **words**, usually a power-of-2 number of bytes: 1, 2, 4, or 8 bytes = 8, 16, 32, 64 bits. The **AVR** used in this course is at the small end — an **8-bit** (1-byte) word; **modern CPUs** sit at the other end, 32- or 64-bit.

[133] courses/csse2010/avr-architecture.pdf
[134] courses/csse2010/avr-instructions.pdf
[135] courses/csse2010/avr-program-structure.pdf
[136] courses/csse2010/bit-masking.pdf
[137] courses/csse2010/memory-models.pdf
[138] courses/csse2010/io-board.pdf
[139] courses/csse2010/device-pinouts.pdf
[140] courses/csse2010/week6-lab-avr-io-ports.pdf
[141] courses/csse2010/avr-usart.pdf
[142] courses/csse2010/analog-to-digital-conversion.pdf
[143] courses/csse2010/avr-spi.pdf
[144] courses/csse2010/serial-communication.pdf
[145] courses/csse2010/2026-08-31-instruction-set-architecture-part-1.pdf
[146] courses/csse2010/2026-10-05-more-io.pdf

Representing whole (unsigned) numbers

Each bit position has a value — a power of 2, increasing from right (least significant) to left (most significant). **Binary → decimal**: add the values of each position where the bit is 1.

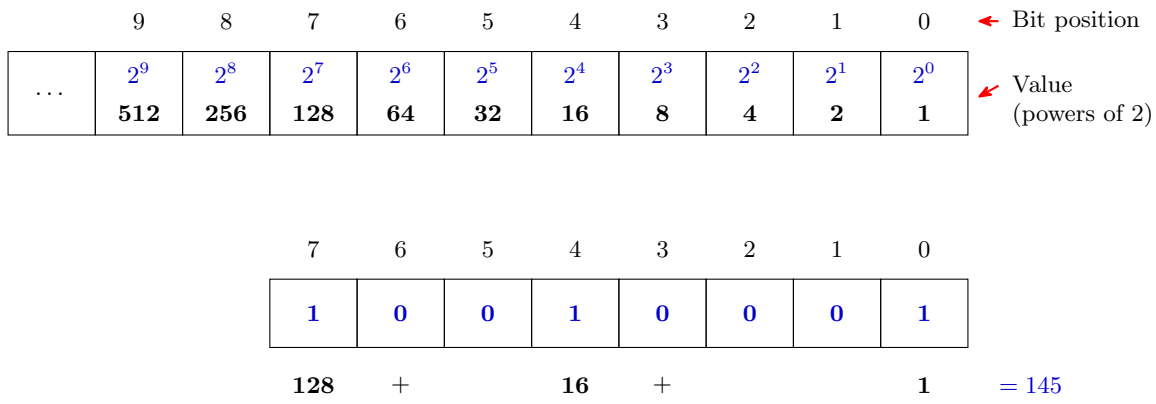


Figure 23: Place values for an unsigned binary number, and a worked conversion. Position i is worth 2^i, so reading a binary number is just adding the place values of the positions holding a 1: $10010001_2 = 128 + 16 + 1 = 145$.

Most and least significant bits

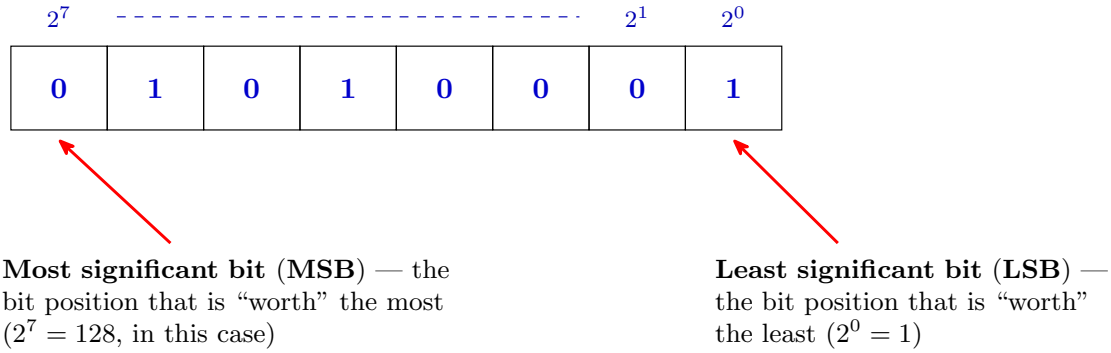


Figure 24: The two ends of a word get names, because almost every later convention (shifting, sign bits, bit numbering in AVR registers) is stated in terms of them. For an n-bit unsigned word the MSB is worth 2^{n-1} — here $2^7 = 128$ — and the LSB is always worth $2^0 = 1$.

Converting decimal to binary

Two equivalent methods, both worked on the same number, 53.

Method 1 — sum of powers of 2. Rewrite n as a sum of powers of 2, by repeatedly subtracting the largest power of 2 not greater than n ($53 - 32 = 21$, $21 - 16 = 5$, $5 - 4 = 1$, $1 - 1 = 0$). Then assemble the binary number from 1's in the bit positions corresponding to those powers of 2, and 0's everywhere else.

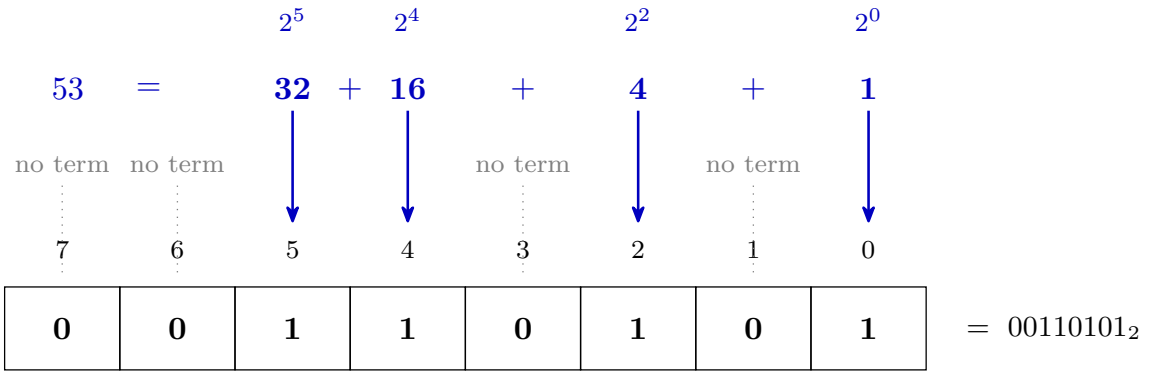


Figure 25: Method 1 on 53. Each power of two in the decomposition names one bit position — 2^5 is position 5, 2^4 is position 4, and so on — and every position with no term in the sum gets a 0. Reading the row off left to right gives 00110101_2.

Method 2 — repeated division. Build the bits up from the right (least significant): divide n by 2, the remainder (0 or 1) is the next bit, then repeat with $n =$ the quotient. Stop when the quotient reaches 0, and read the remainders **upwards**.

Number range (unsigned)

- Smallest representable value: all 0's $\rightarrow$ 0.
- Largest representable value: all 1's $\rightarrow$ for an n-bit word, $2^n - 1$ (e.g. 255 for 8 bits).

The -1 is the part worth understanding: an n-bit word has 2^n distinct patterns, but one of them is spent on zero.

Other radices

Radix = number system base. A radix-k number system has k distinct symbols for digits 0 to $k - 1$, and the value of each digit (from the right) is $k^0, k^1, k^2, \ldots$ — so a radix-k number $d_m \ldots d_1 d_0$ is worth $\sum_i d_i k^i$. E.g. $1\mathrm{A}_{16} = 1 \cdot 16^1 + 10 \cdot 16^0 = 16 + 10 = 26$.

- **Octal** (radix-8): symbols `0`–`7`. One octal digit corresponds to exactly 3 bits.

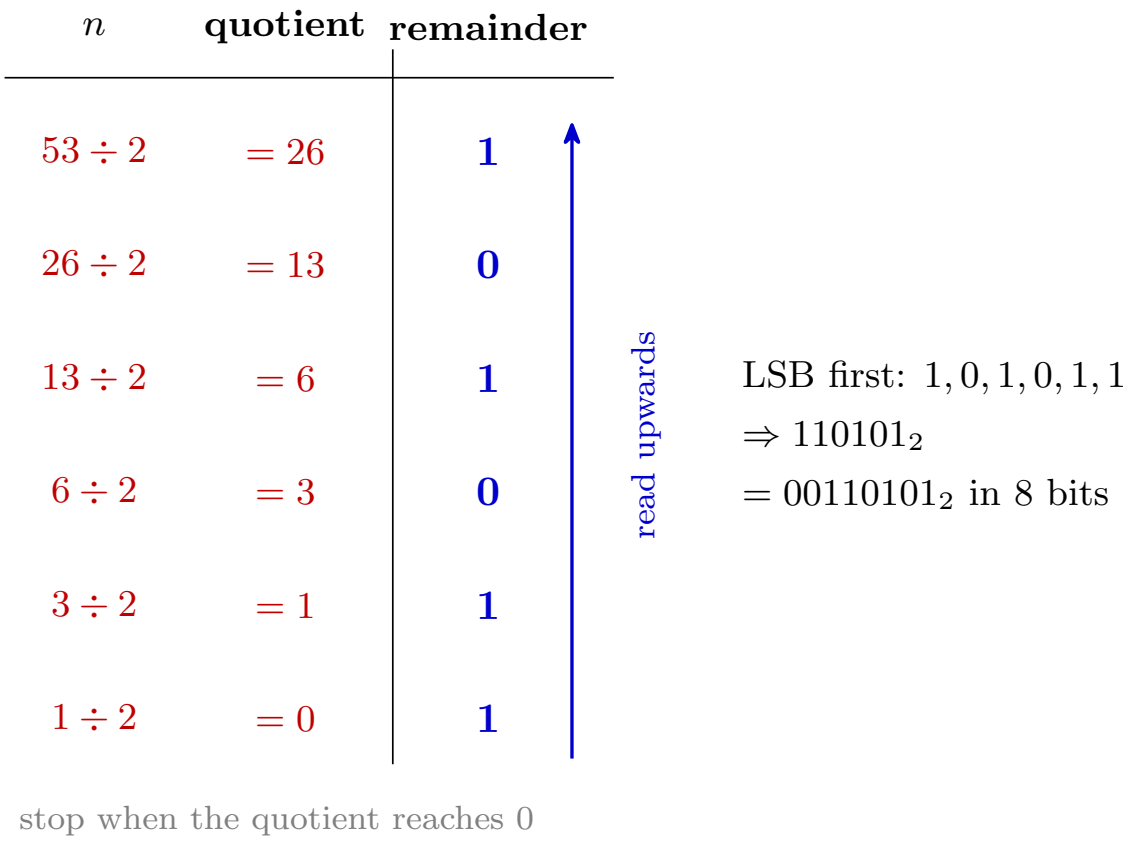


Figure 26: Method 2 on 53. The remainders come out LSB-first, which is why the answer has to be read back up the column: 110101_2, or 00110101_2 padded to a byte. The same procedure with a divisor of 16 or 8 converts to hexadecimal or octal instead.

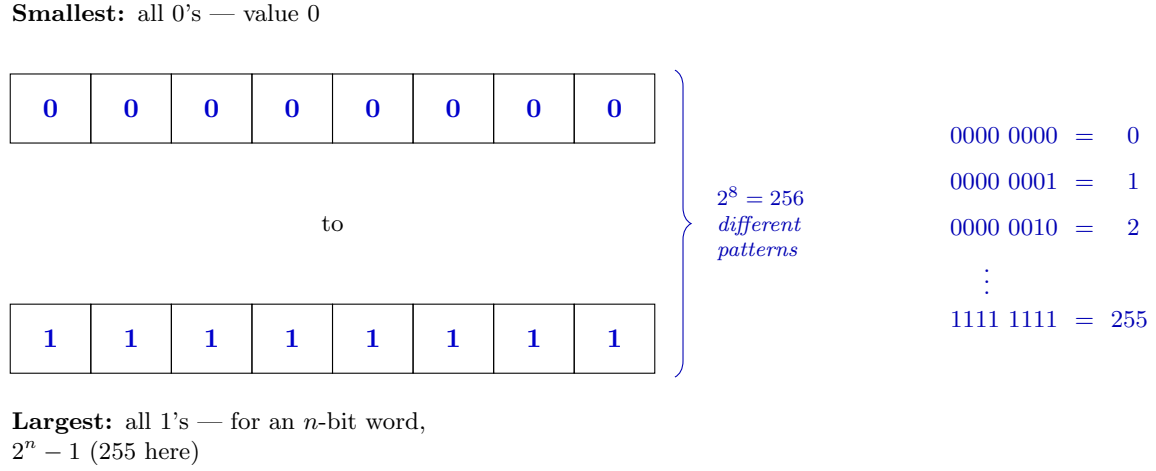


Figure 27: Why the largest unsigned value is $2^n - 1$ and not 2^n. Counting up from all-zeros steps through every pattern in turn — each extra bit doubles how many there are, so 8 bits give $2^8 = 256$ patterns — and because the count starts at 0, the last one is 255.

- **Hexadecimal** (radix-16): symbols `0`–`9`, `A`–`F`. One hex digit corresponds to exactly 4 bits — very convenient for grouping binary.

Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Oct	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17	20	21
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11

Converting binary ↔ hex/octal by grouping

Because $16 = 2^4$ and $8 = 2^3$, no arithmetic is needed: **group the bits from the right** — four at a time for hex, three at a time for octal — and translate each group with the table above. Pad the leftmost group with leading zeros.

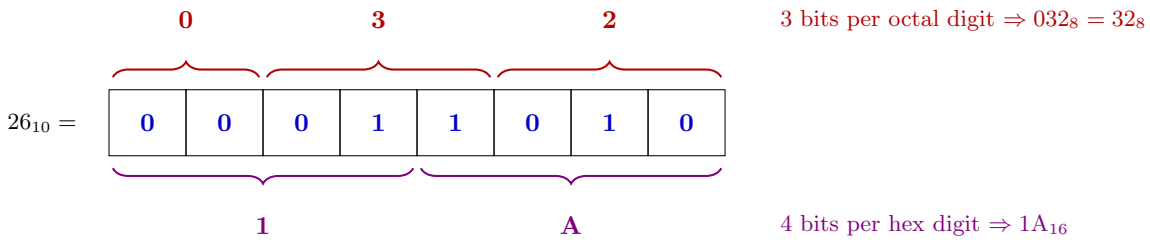


Figure 28: One bit pattern read two ways. Grouping 00011010_2 into nibbles from the right gives 1 and A, i.e. $1A_{16}$; grouping the same bits in threes gives 0, 3 and 2, i.e. 32_8. The grouping must start at the right — starting at the left gives the wrong answer whenever the width is not a multiple of 3 or 4.

The same trick read backwards is why hex is used for bit patterns at all: a full byte of 1's, `1111 1111`, is `F F` = `0xFF` = 255, and any byte can be written as two hex digits.

To convert **decimal → any radix**, use Method 2 with that radix as the divisor: $26 \div 16$ gives quotient 1 remainder 10 (= `A`), so $26_{10} = 1A_{16}$; $26 \div 8$ gives quotient 3 remainder 2, so $26_{10} = 32_8$. See 2026-07-27-course-intro-bits-bytes-binary[147] for that worked exercise.

Radix	Convention	Example	Where used
any	**subscript**	101_2, 101_{16}	**generic/mathematical**
Hex	**leading `0x`**	`0x101`	**C, Atmel AVR**
Hex	trailing `h`	`101h`	some assembly languages
Hex	**leading `$`**	`$747`	**Atmel AVR assembly**
Octal	**leading `0`**	`0101`	**C, Atmel AVR**
Octal	trailing `q`	`101q`	some assembly languages
Octal	leading `@`	`@747`	some assembly languages
Binary	**leading `0b`**	`0b101`	**Atmel AVR assembly, some C**
Binary	trailing `b`	`101b`	some assembly languages
Binary	leading `%`	`%101`	some assembly languages

The octal convention is the trap: in C, a leading `0` silently changes the radix, so `0101` is 65_{10}, not 101_{10}.

Representing signed numbers

Computers don't store + and − signs; the sign has to be encoded in the bits themselves. Four formats have been used, and **all four — plus unsigned — represent positive numbers identically.** Only the negatives differ.

Signed magnitude

The leftmost bit is a **sign bit** (0 positive, 1 negative) and **contributes nothing to the magnitude**. Negate by inverting the sign bit alone.

Ones' complement

The sign bit now also carries weight: its position value is $-(2^{N-1}-1)$. **Negate by inverting all the bits.**

Two's complement

The sign bit's position value is -2^{N-1}. **Negate by inverting all the bits and adding 1.** This is the one actually used in practice.

Compare the position values, 8-bit:

	bit 7	6	5	4	3	2	1	0
Signed magnitude	sign	64	32	16	8	4	2	1

	bit 7	6	5	4	3	2	1	0
Ones' complement	−127	64	32	16	8	4	2	1
Two's complement	−128	64	32	16	8	4	2	1

Excess-2^{N-1}

For 8 bits, "excess 128": the value stored is the true value **plus 128**. So −3 is stored as 125 and 3 as 131.

It is exactly **two's complement with the sign bit reversed** — a 0 in the MSB means bit value −128, a 1 means 0. Its practical use is that comparing two signed numbers becomes simple in hardware. The exponent of an IEEE floating-point number is stored in excess notation for exactly that reason, with a bias one smaller: excess-127 for 8 bits (floating-point-numbers[148]).

Ranges

Format	8-bit	n-bit
Unsigned	0 to 255	0 to $2^n - 1$
Signed magnitude	−127 to 127	$-(2^{n-1} - 1)$ to $2^{n-1} - 1$
Ones' complement	−127 to 127	$-(2^{n-1} - 1)$ to $2^{n-1} - 1$
Two's complement	−128 to 127	-2^{n-1} to $2^{n-1} - 1$
Excess-2^{n-1}	−128 to 127	-2^{n-1} to $2^{n-1} - 1$

Worked example — ±47 in 8 bits

+47 = `00101111` in every format.

Format	−47	How
Signed magnitude	`10101111`	flip the sign bit
Ones' complement	`11010000`	invert all bits
Two's complement	`11010001`	invert all bits, add 1
Excess-128	`01010001`	$-47 + 128 = 81$

And in 4 bits, −3 is `1100` in ones' complement and `1101` in two's complement.

[148] courses/csse2010/floating-point-numbers.pdf

Why "complement"?

A thing and its complement together make a whole — an angle and its complement make a right angle, a set and its complement a universal set.

- **Ones' complement**: a number plus its negative gives a string of ones (`11111111` in 8 bits).
- **Two's complement**: a number plus its negative gives a power of two, with overflow (`1 00000000`). It would more accurately be called *power-of-two's complement*.

Base 10 has the same pair — nines' and ten's complement — used on early mechanical adding machines to do subtraction.

Why two's complement wins

Two representations of zero, which wastes a bit pattern that could have extended the range:

- Signed magnitude: `00000000` and `10000000`
- Ones' complement: `00000000` and `11111111`

Asymmetric range: two's complement and excess-2^{n-1} (one more negative than positive).

Arithmetic: signed-magnitude addition requires taking cases; ones' complement addition requires an extra addition step (the end-around carry); sign extension and comparison are non-trivial in both. Two's complement has a single zero and its addition is *just* binary addition — no special cases.

Format matters

The same eight bits mean different things under different formats. `10010010` reads as:

Read as	Value
Unsigned	146
Two's complement	−110
Ones' complement	−109
Signed magnitude	−18
Excess-128	18

Or `01010101`: 85 unsigned, −43 in excess-128, `'U'` in ASCII, or 10.625 as fixed point with 5 bits before the binary point and 3 after.

The format is not encoded in the data. It has to be known separately — by a human, by a program, or by the hardware. See week1-lab-number-representations[149] for the full 4-bit table across all five formats, and 2026-08-04-binary-arithmetic[150] for what happens when you add these.

Fractions: fixed and floating point

Everything above is whole numbers. Place values carry on past a **binary point** as $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots$, and there are two ways to use them.

- **Fixed point**: agree where the point sits, as in the 5.3 reading of `01010101` above, and read the bits as before. Ordinary integer hardware does the arithmetic, but the format fixes the range and the precision once and for all. See fixed-point-numbers[151].
- **Floating point**: store the digits and the position of the point separately, as binary scientific notation: a sign, a biased exponent and a mantissa. This is IEEE 754, the `float` and `double` of C. See floating-point-numbers[152].

Bit Masking

Using a bitwise operation against a constant **mask** to touch only the bits you care about, leaving the rest alone. See boolean-algebra[153] for the underlying laws and avr-instructions[154] for the instructions used here. The same masks, written from the bit names in the AVR's header files, are how every I/O register is set up in C: see set, clear and toggle and writing register bits by name.

Sometimes you are interested in only a few bits of a register. A bitwise **AND** can **mask out** the bits that are not of interest — and more generally, masking can selectively **set**, **clear**, **toggle** or **extract** bits.

The reason any of this is possible is that a bitwise instruction is not one eight-bit operation. It is **eight independent one-bit operations running in parallel**, and the two operands' bit i decide result bit i with no reference to any other column.

Read Figure 29 column by column and the technique stops needing to be memorised: since the columns are independent, choosing a mask is just choosing, for each of the eight positions separately, whether that position gets *passed through* or *forced*.

[149] courses/csse2010/week1-lab-number-representations.pdf
[150] courses/csse2010/2026-08-04-binary-arithmetic.pdf
[151] courses/csse2010/fixed-point-numbers.pdf
[152] courses/csse2010/floating-point-numbers.pdf
[153] courses/csse2010/boolean-algebra.pdf
[154] courses/csse2010/avr-instructions.pdf

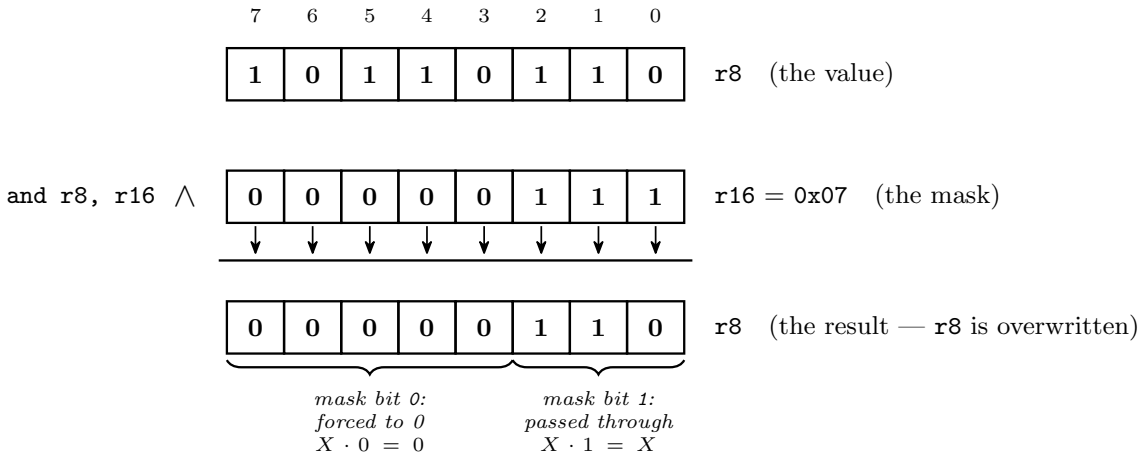


Figure 29: Why a mask works, one column at a time. Two rows of bits, an arrow down out of each column into a result row that overwrites the destination: here `and r8, r16` with the mask `0x07`, extracting the low three bits. The eight columns never interact — one mask bit decides one result bit, a `1` passing the value's bit through and a `0` forcing it to zero — and all eight happen at once in a single instruction. Choosing a mask is therefore nothing more than deciding, position by position, which bits to keep.

The three identities that make it work

Each of AND, OR and XOR has a value that **passes a bit through unchanged** and a value that **forces it**. That pairing is the whole technique.

	$X \cdot 0$	$X \cdot 1$	$X \cdot X$
AND	$= 0$	$= X$	$= X$

	$X + 0$	$X + 1$	$X + X$
OR	$= X$	$= 1$	$= X$

	$X \oplus 0$	$X \oplus 1$	$X \oplus X$
XOR	$= X$	$= \overline{X}$	$= 0$

Reading off the pass-through and force columns gives the operation to reach for:

To do this to the selected bits	Use	Mask bit = 1 on	Mask bit = 0 on
Clear them to 0	`AND`	the bits to **keep** — $X \cdot 1 = X$	the bits to **clear** — $X \cdot 0 = 0$
Set them to 1	`OR`	the bits to **set** — $X + 1 = 1$	everything else — $X + 0 = X$
Toggle them	`EOR`	the bits to **flip** — $X \oplus 1 = \overline{X}$	everything else — $X \oplus 0 = X$
Extract them (others zeroed)	`AND`	the bits to **keep** — $X \cdot 1 = X$	the bits to **discard** — $X \cdot 0 = 0$

Note the asymmetry: AND's mask is 1 on the bits you *keep*, whereas OR's and XOR's masks are 1 on the bits you *act on*. Getting that backwards is the usual bug — with AND you write the mask for what survives, with the other two for what changes.

$X \oplus X = 0$ is the other one worth remembering: XORing a register with itself is the cheapest way to zero it.

In AVR assembly

The masks are ordinary constants, so the bitwise instructions of avr-instructions[155] do the work. On AVR the immediate forms (`andi`, `ori`) **only reach r16–r31**; for a low register, load the mask into a high register and use the two-register form.

Extract the least significant three bits of `r7` into `r8`, other bits of `r8` zero.

```
    mov  r8, r7          ; copy the source
    ldi  r16, 0x07       ; mask 0000 0111 - keep the bottom three
    and  r8, r16         ; everything else forced to 0
```

Toggle the most significant four bits of `r9`, other bits unchanged.

```
    ldi  r16, 0xF0       ; 1111 0000 - flip the top four
    eor  r9, r16         ; bottom four XOR 0, so unchanged
```

Clear the least significant four bits of `r10`, other bits unchanged.

```
    ldi  r16, 0xF0       ; 1 on the bits to KEEP
    and  r10, r16        ; bottom four AND 0 -> 0
```

Set the most significant four bits of `r11` to 1, other bits unchanged.

```
    ldi  r16, 0xF0       ; 1 on the bits to set
    or   r11, r16        ; bottom four OR 0, so unchanged
```

Note how `0xF0` appears as the mask in three of these with three different meanings — *keep*, *flip*, *set* — decided entirely by the instruction it is paired with.

In C

The same four operations with C's bitwise operators (c-operators[156]), which is how every AVR C program touches its ports. The masks are the same; the immediate-register restriction disappears, since the compiler chooses the registers.

[155] courses/csse2010/avr-instructions.pdf
[156] courses/csse2010/c-operators.pdf

Operation	Operator	Example	Reads as
Clear particular bits	`&` with 0s on the bits to clear	`var = PINB & 0x0F;`	keep the low nibble, zero the high
Test particular bits	`&` with a 1 on the bit to test	`if (PINB & 0x02) …`	true if bit 1 is set
Set particular bits	`|` with 1s on the bits to set	`PORTC = PINB | 0x80;`	force bit 7 to 1
Invert particular bits	`^` with 1s on the bits to flip	`PORTC = PINB ^ 0x0F;`	flip the low nibble

Write the mask as a shift, not a hex constant: `(1<<5)` says *bit 5* where `0x20` has to be decoded, and the header files name every register bit as a bit number for exactly this use (c-preprocessor-and-avr-headers[157]):

```
DDRB  = (1<<5);                 /* bit 5 output, every other bit input: REPLACES the
 ↪  register */
DDRB |= (1<<4) | (1<<3);        /* bits 3 and 4 output, the rest unchanged:
 ↪  read-modify-write */
DDRB &= ~(1<<3);                /* bit 3 back to input, the rest unchanged
 ↪  */
PORTC ^= (1<<0);                /* toggle bit 0
 ↪  */
TCCR1B = (1<<WGM12) | (1<<CS11) | (1<<CS10);   /* named bits, ORed into one value
 ↪  */
```

The difference between the first two lines is the one that bites: `=` writes the whole register and silently sets the other seven bits to 0, `|=` ORs into what is already there. In assembly the second line is three instructions — `in`, `ori`, `out` — and the C compiler emits exactly those (week8-lab-avr-c-programming[158]). For a single-bit *clear* the mask is the complement of the set mask, `~(1<<n)`, which is why *clear bit 3* is `&= ~(1<<3)` and not `&= (1<<3)`.

The test idiom needs no comparison. `PINB & 0x02` is `0x02` when bit 1 is set and `0` when it is not, and C treats any non-zero value as true (c-control-flow[159]) — so `if (PINB & 0x02)` is right and `if ((PINB & 0x02) == 1)` is *always false*, because the masked value is 2, never 1.

[157] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[158] courses/csse2010/week8-lab-avr-c-programming.pdf
[159] courses/csse2010/c-control-flow.pdf

Set, clear and toggle on an I/O register

Changing one bit of an I/O register, and only that bit, is one of three lines. Each reads the register, combines it with a one-bit mask and writes it back:

To	Write	The mask	Why the other bits survive
set bit n	`REG |= (1<<n);`	a 1 on bit n	$X + 0 = X$
clear bit n	`REG &= ~(1<<n);`	a 0 on bit n, 1s elsewhere	$X \cdot 1 = X$
toggle bit n	`REG ^= (1<<n);`	a 1 on bit n	$X \oplus 0 = X$

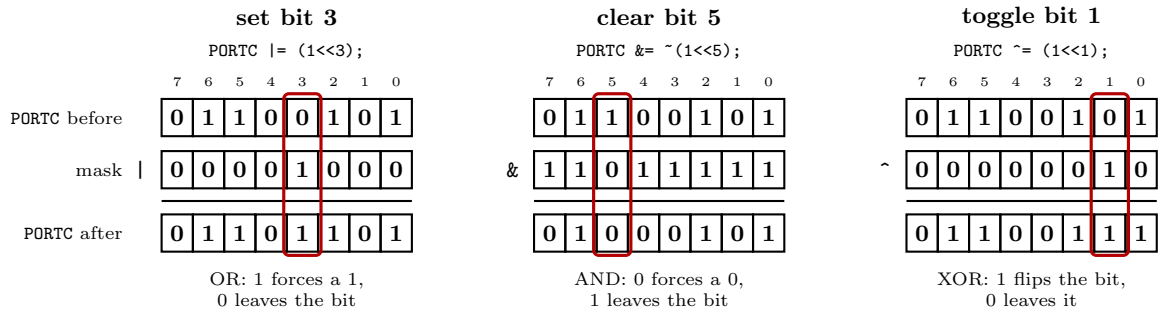


Figure 30: Set, clear and toggle on one register byte. Each panel starts from the same value of `PORTC`, 0110 0101, and changes one bit, outlined, with a single read-modify-write. The mask is a 1 shifted to the bit's position, complemented for a clear. In every other column the mask holds the value that leaves a bit alone: 0 for OR and XOR, 1 for AND. Set gives 0110 1101, clear gives 0100 0101 and toggle gives 0110 0111, and each time the other seven bits are untouched.

Toggle is not enable. `TIMSK2 ^= (1<<OCIE2A);` turns the timer 2 compare A interrupt on only if it was off, and a second run turns it off again. To enable whatever the bit was before, set it with `|=`; to disable, clear it with `&= ~`. Toggling is for things meant to alternate, like an LED blinking.

The register has to be one that stores what is written. `PORTx` and `DDRx` do. `PINx` does not: it reads the pins, and on the ATmega324A a 1 written to one of its bits toggles the matching `PORTx` bit, so a read-modify-write on `PINx` changes the wrong bits (avr-io-ports[160]).

[160] courses/csse2010/avr-io-ports.pdf

Writing register bits by name

When setting or clearing bits of an I/O register, **be explicit about which bits**. Three ways to write the same value, from worst to best:

```
TCCR1B = 10;                          /* poor:   a number                    */
TCCR1B = 0x0A;                        /* better: the pattern, 0000 1010      */
TCCR1B = (1<<WGM12) | (1<<CS11);      /* best:   the bits, by name           */
```

`WGM12` is bit 3 of `TCCR1B` and `CS11` is bit 1, so all three are `0000 1010`. Only the last says *which* bits, so a reader can check them against the datasheet without decoding anything: CTC mode, clock divided by 8 (avr-timers-and-counters[161]). It costs nothing, either. The compiler works out `(1<<WGM12) | (1<<CS11)` while compiling, and all three lines become the same instructions.

The names are bit numbers. The device header that `<avr/io.h>` includes defines every bit name as the bit's position in its register:

```
#define WGM02 3
```

A name is not a mask; `1<<` makes it one (c-preprocessor-and-avr-headers[162]). The names are the datasheet's, and the datasheet has errors in places, so the I/O register summary at the back is the one to check a name against.

Assembly uses the same names. In `ldi r16, (1<<WGM02)|(1<<CS01)` the assembler works out the constant, `0x0A`, and the instruction stored is `ldi r16, 0x0A`.

A name belongs to one register. `WGM12` and `WGM02` are both 3, so `(1<<WGM12)` written into `TCCR0B` sets `WGM02`, and the compiler cannot notice, because all it sees is the number 3. Pair each name with the register whose datasheet entry defines it. The same byte `0x0A` is CTC mode in `TCCR1B` and a reserved mode in `TCCR0B` (2026-10-05-more-io[163]).

Masking sets the status flags too

Status register flags will be set also — a mask operation is a real ALU operation, not a free one. `AND`, `OR` and `EOR` all clear V and set N and Z from the result (C is untouched); see avr-architecture[164].

That side effect is often the actual point. `andi r16, 0x01` keeps only bit 0, so:

[161] courses/csse2010/avr-timers-and-counters.pdf
[162] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[163] courses/csse2010/2026-10-05-more-io.pdf
[164] courses/csse2010/avr-architecture.pdf

$$Z = 1 \iff \text{bit 0 of } \texttt{r16} \text{ is } 0 \iff \texttt{r16} \text{ is even}$$

which makes a single masked AND an even/odd test. Any single-bit mask turns the Z flag into a test of that one bit — and `tst rd`, which is just `and rd, rd`, is the degenerate case that tests the whole register against zero.

Common mistakes

- **`&= (1<<n)` to clear bit *n*.** That keeps bit n and clears the other seven. A clear's mask is the complement, `~(1<<n)`.
- **`=` where `|=` was meant.** `DDRB = (1<<5)` makes PB5 an output and every other pin of port B an input.
- **A bit name without the shift.** `TCCR1B |= WGM12;` ORs in the number 3, `0000 0011`, which sets `CS11` and `CS10`, clock divided by 64, and leaves `WGM12` alone.
- **A bit name in the wrong register.** `(1<<WGM12)` in `TCCR0B` sets `WGM02`.
- **`^=` to enable.** A toggle enables only from off; run twice, it disables.
- **A read-modify-write on `PINx`.** Writing `PINx` toggles outputs, so it cannot clear or set a bit.
- **Getting AND's mask backwards.** With AND the 1s mark the bits to *keep*; with OR and XOR they mark the bits to *change*.
- **`if ((x & mask) == 1)`.** The masked value is the mask's own value, so the test is false unless the mask is bit 0. Test for non-zero.

See also

- boolean-algebra[165] · avr-instructions[166] · avr-architecture[167]
- avr-io-ports[168] — one bit of a port in C and assembly, and why `PINx` is different
- avr-timers-and-counters[169] — the `TCCRnB` fields named in the examples
- 2026-10-05-more-io[170] — Lecture 19: poor, better and best, `TIMSK2` toggled
- week5-lab-avr-assembly-one[171] — Lab 9, where these snippets are the exercise
- week8-lab-avr-c-programming[172] — Lab 11, the same operations in C · c-operators[173] · c-preprocessor-and-avr-headers[174]

[165] courses/csse2010/boolean-algebra.pdf
[166] courses/csse2010/avr-instructions.pdf
[167] courses/csse2010/avr-architecture.pdf
[168] courses/csse2010/avr-io-ports.pdf
[169] courses/csse2010/avr-timers-and-counters.pdf
[170] courses/csse2010/2026-10-05-more-io.pdf
[171] courses/csse2010/week5-lab-avr-assembly-one.pdf
[172] courses/csse2010/week8-lab-avr-c-programming.pdf
[173] courses/csse2010/c-operators.pdf
[174] courses/csse2010/c-preprocessor-and-avr-headers.pdf

Boolean Algebra

Logic functions can be expressed as expressions of **variables** (literals, e.g. A, B, X) and **functions** (the operators $+$, $\cdot$, $\oplus$, $\bar{}$) — i.e. an output written in terms of its inputs, $X = f(A, B)$. Variables and functions can only take the values 0 or 1. **Boolean algebra** is the set of rules governing how such expressions can be manipulated.

Notation conventions

- **Inversion**: overline, e.g. $\text{NOT}(A) = \bar{A}$ (pronounced "A bar").
- **AND**: dot, or implied by adjacency, e.g. $\text{AND}(A, B) = AB = A \cdot B$.
- **OR**: plus sign, e.g. $\text{OR}(A, B, C) = A + B + C$.
- Other examples: $\text{XOR}(A, B) = A \oplus B = \bar{A}B + A\bar{B}$; $\text{NAND}(A, B, C) = \overline{ABC}$; $\text{NOR}(A, B) = \overline{A + B}$.

$+$ and $\cdot$ are **OR** and **AND**, not arithmetic addition and multiplication — $1 + 1 = 1$ here.

Boolean identities

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\bar{A} = 0$	$A + \bar{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A}\bar{B}$

Two rows in particular do the heavy lifting when simplifying: the **inverse law** ($A\bar{A} = 0$, $A + \bar{A} = 1$) collapses whole terms to a constant, and the **absorption law** ($A + AB = A$) deletes a term outright.

The **distributive law in its AND form is not the arithmetic one** — $A + BC = (A + B)(A + C)$ has no algebraic analogue and is easy to misremember.

De Morgan's law and gate equivalents

De Morgan's law says AND and OR can be **interchanged if you invert both the inputs and the output**. Drawn as gates, that gives four equivalences — every gate has a second symbol, and pushing an inversion through a gate always flips its shape:

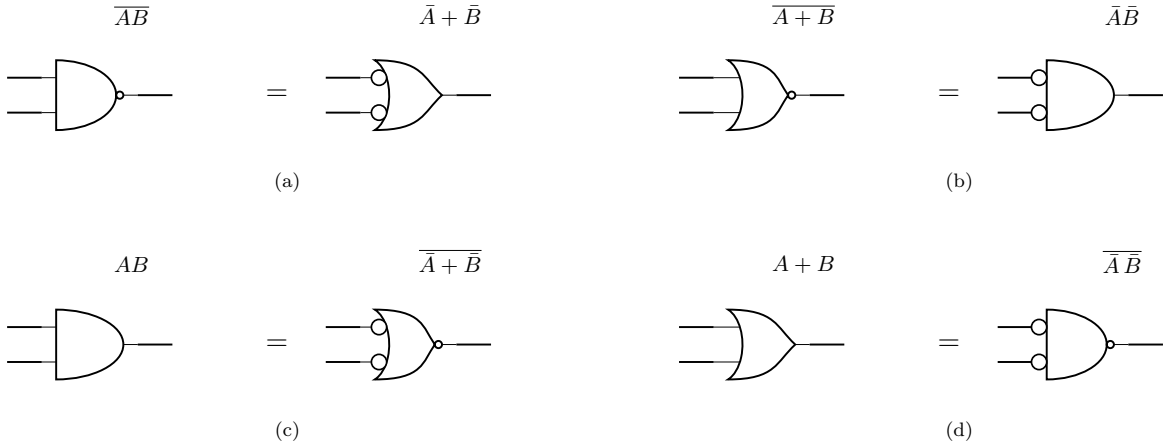


Figure 31: De Morgan's law drawn as gate symbols. (a) a NAND is an OR with inverted inputs; (b) a NOR is an AND with inverted inputs; (c) and (d) are the same two statements read the other way. Each pair is the same function, so either symbol may be used - pick whichever makes the surrounding bubbles cancel.

Exercise: **verify each of the four with a truth table.**

This is also why NAND-only and NOR-only circuits can implement any function (see logic-gates[175]) — an AND feeding an OR can always be redrawn as a NAND feeding a NAND.

Simplifying an expression

Worked example:

$$\text{CODE} = C + CD + \overline{(D + \bar{D})}$$

1. Apply **De Morgan's law** to the last term: $\overline{D + \bar{D}} = \bar{D} \cdot \bar{\bar{D}} = \bar{D} \cdot D = 0$ (the double overline cancels, then the **inverse law** $A\bar{A} = 0$ applies).
2. The **identity law** $0 + A = A$ drops that term: $\text{CODE} = C + CD$.
3. The **absorption law** $A + AB = A$ finishes it: $\text{CODE} = C$.

The whole expression is just C — neither D nor the CD term affects the output at all.

[175] courses/csse2010/logic-gates.pdf

Sum of products

Any logic function can be implemented as the **OR of AND combinations** of its inputs (a *sum of products*):

1. For each row where the truth table output is 1, write the AND term of the inputs (complementing wherever the input is 0) that produces that row.
2. OR all of those terms together.

Because every 1-row gets its own term, and each term is true for exactly that one row, the result is guaranteed to reproduce the table. Worked through in full, with the resulting circuit drawn out, in 2026-07-30-intro-to-logic-gates[176].

Sum of products always works, but does **not** necessarily give the minimum number of gates — the resulting expression can usually be simplified further with the identities above. E.g. $Z = AB + AC = A(B + C)$ (distributive law) needs two gates rather than three.

C Control Flow

The three shapes a C program's control can take — **sequential**, **selection** and **repetition** — with the statements the course uses for each: `if … else`, `while` and `for`. First met in 2026-09-07-introduction-to-embedded-c-programming[177] and 2026-09-10-more-c-programming-on-the-avr[178]; the assembly equivalents are in avr-program-structure[179].

The three shapes

	Shape	Statements
sequential	statements run one after another	the default — any list of statements
selection (decision)	a test picks one of two paths	`if … else`
repetition	a body runs again while a test holds	`for`, `while`

All three are drawn as flowcharts in 2026-09-07-introduction-to-embedded-c-programming[180]. Every program is built from nothing but these, nested.

[176] courses/csse2010/2026-07-30-intro-to-logic-gates.pdf
[177] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[178] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[179] courses/csse2010/avr-program-structure.pdf
[180] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf

Truth

There is no boolean type. **C interprets any 0 value as false, anything else as true.** So:

Written	Means
`if(a)`	`if(a != 0)`
`if(!a)`	`if(a == 0)`

Comparisons yield 1 or 0, so `if (a > b)` and `if ((a > b) != 0)` are the same test. A bitwise test like `PINC & 0x01` yields either `0x01` or `0x00`, which is why it can sit directly inside an `if` — anything non-zero is true, and it does not need to be exactly 1.

To the hardware a truth test is a compare against zero and a branch on **Z** — `tst`/`cp` then `breq`/`brne` in avr-instructions[181]. `if(a)` is the C spelling of *branch if not zero*.

Selection — `if`

```
if (expression)
    stmt
else
    stmt
```

- **The `else` clause is optional.**
- **`stmt` can be replaced by multiple statements enclosed in braces `{ }`.** Without the braces, only the *one* statement following the `if` (or `else`) is conditional — a second line at the same indentation runs unconditionally, whatever the indentation suggests. Always use the braces.

An `if` used as a guard:

```
if(digit < 10) {
    PORTA = seven_seg[digit];
} else {
    PORTA = 0;
}
```

Four input bits can hold 0–15, the array has ten entries, and the `if` is what keeps `seven_seg[digit]` inside it (c-pointers-and-arrays[182] — C does not check).

An `else if` chain is an `else` whose single statement is another `if`; nothing new is needed for it.

[181] courses/csse2010/avr-instructions.pdf
[182] courses/csse2010/c-pointers-and-arrays.pdf

Repetition — `while`

```
while (expression)
    stmt
```

Test first; if true, run the body, then test again; stop when the test is false. If the test is false to begin with, the body never runs.

`while(1)` — a condition that is always true — is the shape of every embedded program:

```
int main(void) {
    /* set-up, once */
    DDRA = 0xFF;
    while(1) {
        /* read inputs, compute, write outputs - forever */
        digit = PINC & 0x0F;
        …
    }
}
```

On the AVR `main` has nowhere to return to. The program is *set-up, then loop forever*, and the loop body is one pass of reading inputs and writing outputs. In assembly this was a label and an unconditional `rjmp` back to it (avr-program-structure[183]); `while(1)` is the same jump with the label hidden.

Repetition — `for`

```
for (init; test; step)
    stmt
```

is exactly

```
init;
while (test) {
    stmt
    step;
}
```

[183] courses/csse2010/avr-program-structure.pdf

— the `for` only gathers the three parts of a counted loop onto one line so they cannot drift apart. Two traversals of the array `a`:

```
for(index=0; index<10; index++) {     /* init: index=0; test: index<10; step: index++
 ↪  */
    a[index] = 0;
}

for(ptr=a; ptr < a+10; ptr++) {       /* same loop, counting in addresses */
    *ptr = 0;
}
```

The bound is the thing to get right. `index < 10` runs the body for 0 through 9, the ten legal indices of `a[10]`; `index <= 10` runs it once more, for `a[10]`, which does not exist. The pointer form has the same edge: `ptr < a+10` stops at the address one past the end without ever touching it.

Infinite loops, deliberately

The macro `#define EVER ;;` makes `for(EVER)` a spelling of `for(;;)` — a `for` with no init, no test and no step, which loops forever (c-preprocessor-and-avr-headers[184]). It is `while(1)` in a different costume; the course's programs use `while(1)`.

Other control statements

`switch`, `do … while`, `break` and `continue` are all real C. They are not in these notes yet.

See also

- c-language-basics[185] · c-operators[186] · c-pointers-and-arrays[187] · c-preprocessor-and-avr-headers[188]
- avr-program-structure[189] — labels and branches, the same shapes in assembly
- avr-instructions[190] — `cp`, `breq`, `brne`: what a test compiles to

[184] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[185] courses/csse2010/c-language-basics.pdf
[186] courses/csse2010/c-operators.pdf
[187] courses/csse2010/c-pointers-and-arrays.pdf
[188] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[189] courses/csse2010/avr-program-structure.pdf
[190] courses/csse2010/avr-instructions.pdf

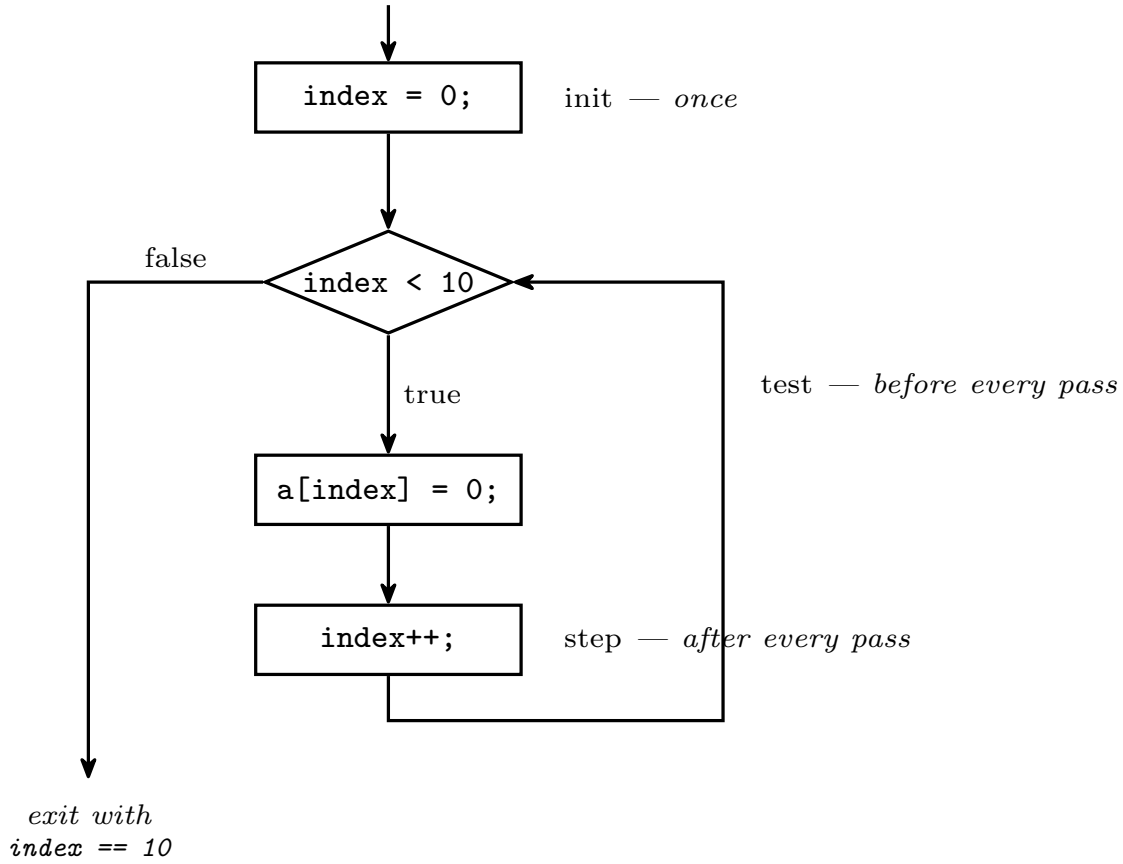


Figure 32: Where the three clauses of a `for` run, drawn on the repetition shape. *init* runs once, before anything; *test* runs before every pass, including the first; *step* runs after every pass, before the next test. The body of `for(index=0; index<10; index++)` therefore runs exactly ten times, with `index` taking 0 through 9 — and when the loop exits `index` is 10, the first value that failed the test. This is the `while` equivalence above, drawn.

- 2026-09-07-introduction-to-embedded-c-programming[191] — Lecture 13 · 2026-09-10-more-c-programming-on-the-avr[192] — Lecture 14

C Language Basics

What a C program is made of: the shape of a program, variables and their types, functions, and constants. Operators are in c-operators[193], the `if`/`while`/`for` statements in c-control-flow[194], arrays and pointers in c-pointers-and-arrays[195], and the preprocessor in c-preprocessor-and-avr-headers[196]. First met in 2026-09-07-introduction-to-embedded-c-programming[197].

Where C sits

C is the top layer of the course's three blocks — *high-level programming* over *low-level programming* over *digital logic design* — and the course's C is **embedded C for the ATmega324A**, the same chip the assembly targeted. The difference from assembly is one extra tool. An assembler turns each line into exactly one instruction; a **compiler** turns each C statement into however many instructions it needs, chooses the registers, and hands the result to the assembler. The chain from C statement down to control unit is drawn in 2026-09-07-introduction-to-embedded-c-programming[198], and the steps after the compiler, object files and the linker, are in compilation-and-linking[199].

What is expected: write small C programs and functions from scratch, understand the meaning of C programs, and write or modify C programs — practised in Labs 11–16, which all use C.

The shape of a program

- There must be a function **called `main`**; it is executed when the program starts.
- Blocks of code are enclosed by **braces `{ }`**.
- Statements **end with a semicolon `;`**.
- C is **case sensitive** — `variable` and `Variable` are different names.
- Comments are `/* … */` (any length, may span lines) or `//` to the end of the line.

[191] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[192] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[193] courses/csse2010/c-operators.pdf
[194] courses/csse2010/c-control-flow.pdf
[195] courses/csse2010/c-pointers-and-arrays.pdf
[196] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[197] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[198] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[199] courses/csse2010/compilation-and-linking.pdf

```c
/* the smallest useful embedded program: */
#include <avr/io.h>

int main(void) {
    DDRA = 0xFF;        /* a statement */
    while(1) {          /* a block */
        PORTA = 0x55;
    }
}
```

On the AVR `main` never returns — there is no operating system to return to — so every program ends in an infinite loop (c-control-flow[200]). `#include` is the preprocessor's (c-preprocessor-and-avr-headers[201]).

Variables and types

Declaration: `type-name variable-name, variable-name …;`

A declaration reserves room in data memory (memory-models[202]) and gives it a name and a type. The type decides two things: **how many cells** the variable takes, and **how the bits are read** — the signed/unsigned question of binary-number-representations[203], now answered once at the declaration instead of instruction by instruction.

Declaration	Size	Range	
`char c;`	1 byte	−128 … 127	two's complement
`unsigned char x;`	1 byte	0 … 255	
`int day;`	**machine dependent** — 16 bits on the AVR	−32768 … 32767	two's complement
`unsigned int count;`	as `int`	0 … 65535 on the AVR	
`float`	32 bits	about $\pm 3.4 \times 10^{38}$, 7 digits	IEEE single precision; no AVR hardware — done in software
`double`	64 bits in general; 32 on avr-gcc		likewise

On avr-gcc `float` and `double` are interchangeable: both are 32 bits (below).

[200] courses/csse2010/c-control-flow.pdf
[201] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[202] courses/csse2010/memory-models.pdf
[203] courses/csse2010/binary-number-representations.pdf

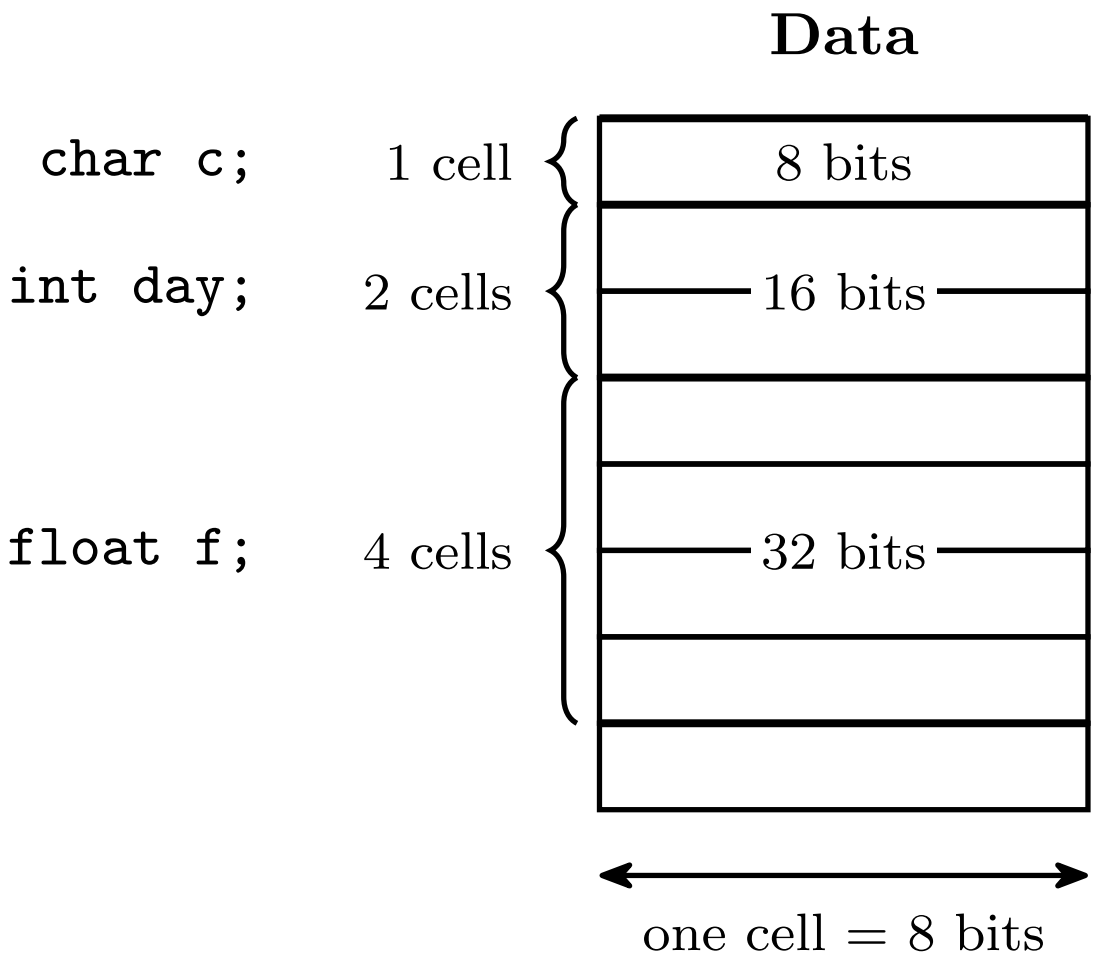


Figure 33: What three declarations reserve in the AVR's 8-bit data memory, drawn as a memory map. `char` is one cell. `int` is two, because `int` is 16 bits on the AVR — which is why a pointer to an `int` steps by two in the pointers note. `float` is four, because single precision is 32 bits. The compiler decides the actual addresses; the addresses here are schematic and the order is arbitrary.

Three consequences worth carrying around:

- **`char` is an integer.** It is exactly the 8-bit two's complement byte of the hardware block; the name only promises it can hold a character code. Arithmetic on it is byte arithmetic, and `'A' + 1` is `'B'` because `'A'` is 65.
- **`int` is not one size.** Write code that assumes 16 bits and it breaks on a desktop; assume 32 and it breaks on the AVR. The AVR headers provide `uint8_t`, `int16_t` and friends — types with the width in the name — and the course's example program uses `uint8_t` for exactly this reason.
- **Nothing is checked.** Storing 300 in an `unsigned char` does not fail; it stores 300 mod 256, i.e. 44. (For a signed `char` the C standard leaves the result implementation-defined; gcc also gives 44.) The type is a reading rule, not a guard.

Booleans. There is no boolean type in the course's C. **Any value can be treated as a boolean: zero means false, non-zero means true.** This is what lets `if(a)` mean `if(a != 0)` (c-control-flow[204]).

Functions

```
return-type function-name (arg-decl)
{
    variable-declarations;
    …
    statements;
}
```

Part	Meaning	Example
`return-type`	what type is returned, or `void` for nothing	`int`, `void`
`function-name`	the name; `main` for the entry point	`average`
`arg-decl`	argument (parameter) declarations, comma-separated, each with its type	`int a, float b`
`variable-declarations`	variables used **only** within the function	`int avge;`
`statements`	the code which actually does stuff	

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

An example:

```
/*
  Return the average of two integers
  (result will be rounded towards 0)
*/
int average(int a, int b) {
    int avge;

    avge = (a+b)/2;

    return avge;
}
```

- `avge = (a+b)/2;` is a **statement** built from **expressions** — `(a+b)`, then `(a+b)/2`, then the assignment; the outermost expression plus its semicolon is the statement.
- `return avge;` hands the value back and ends the function.
- `avge` is **local**: it exists only while `average` is running. Nothing outside can name it.
- **Integer division truncates toward zero.** `average(1, 2)` computes `3/2` and returns `1`; the `.5` never exists. The comment says *rounded towards 0*, and that is the whole of C's rule for `/` on integers — `-3/2` is `-1`, not `-2`.

A function with `void` in front returns nothing and is called for its effect. `int main(void)` is a function whose argument list is explicitly empty.

Where a variable lives

Where a declaration is written decides where the compiler stores the variable and how long it exists.

Declaration	Kind	Stored	Exists
outside every function: `int count;`	**global**	at a fixed address in data memory, from `0x0100` up	for the whole run
inside a function: `int avge;`	**local**	in the call's **stack frame**	only while that call runs
inside a function: `static int calls;`	**`static` local**	with the globals, at a fixed address	for the whole run, keeping its value between calls
outside every function: `static int a;`	**`static` global**	like any global	for the whole run; the name is private to its file

So `avge` in `average` above is made fresh on every call and is gone at the `return`, and a local that is never assigned starts out holding whatever was left on the stack. A `static` local is the way to have a function remember something from one call to the next. Memory obtained from `malloc()` comes from a third place, the heap, which grows from above the globals toward the stack. The full picture is in compilation-and-linking[205], and the frame itself is in procedures-and-the-stack[206].

Constants

Character constants

Single quotes: `'a'`, `'b'`, `'1'`. A character constant is **a number** — its ASCII code — so `'1'` is 49, not 1.

Range	Codes
`'A'` … `'Z'`	65 … 90
`'a'` … `'z'`	97 … 122
`'0'` … `'9'`	48 … 57

Which makes `c - '0'` the digit value of a digit character, and `c + ('a' - 'A')` the lower-case of an upper-case letter — both plain integer arithmetic.

Special characters are **backslash escaped**:

`'\n'`	newline
`'\''`	single quote
`'\t'`	tab
`'\\'`	backslash

String constants

Double quotes, escapes allowed inside: `"abc \n \" hello\t"`. A string is an array of `char` (c-pointers-and-arrays[207]).

Integer constants

[205] courses/csse2010/compilation-and-linking.pdf
[206] courses/csse2010/procedures-and-the-stack.pdf
[207] courses/csse2010/c-pointers-and-arrays.pdf

Base	Written	Examples
Decimal	as usual	`3`, `-27`, `65535`, `+5`
Hexadecimal	leading `0x`	`0x5F`, `0xFFFF`, `0xDEADBEEF`
Octal	**leading `0`**	`0377` = 255

The octal rule is a trap: **a leading zero changes the base**. `010` is eight. Do not zero-pad a decimal constant.

Hexadecimal is the natural notation for anything that is really a bit pattern — port values, masks — because each digit is exactly one nibble (binary-number-representations[208]). `0xFF` *is* `11111111` at a glance; `255` is not.

Floating point constants

Include a decimal point and/or an `e` exponent: `3.1416`, `-7.`, `6.02e23`, `-5.2e-2`. **`7` is an integer; `7.` is floating point.** The type of a constant is decided by how it is written, and it decides what arithmetic happens: `7/2` is `3`, `7./2` is `3.5`.

Floating point on the AVR

`float` and `double` are usually IEEE single and double precision, but the C standard only asks for minimums: about 6 significant decimal digits for `float` and 10 for `double`, both up to 10^{37}. **avr-gcc makes `double` the same 32-bit single precision as `float`**, which gives about 7 digits and so falls short of what the standard asks of `double`. Code written for a desktop that relies on `double` quietly loses half its digits on the AVR.

The AVR also has **no floating-point hardware**: its only hardware data type is the 8-bit integer (instruction-set-architecture[209]). Every `float` add, multiply or conversion is a call into a software library, hundreds of instructions with branches where an integer operation takes a few. So where integer arithmetic can give the same result, use it:

```
PORTB = PINA / 3;        /* integer: a handful of instructions, no branches */
PORTB = PINA * 0.334f;   /* float: the same results, 300+ instructions      */
```

Scaling to whole units (millivolts, tenths of a degree) or using fixed-point-numbers[210] keeps the fractions without the floats. The formats themselves, and why `0.1` cannot be stored exactly, are in floating-point-numbers[211].

[208] courses/csse2010/binary-number-representations.pdf
[209] courses/csse2010/instruction-set-architecture.pdf
[210] courses/csse2010/fixed-point-numbers.pdf
[211] courses/csse2010/floating-point-numbers.pdf

See also

- c-operators[212] · c-control-flow[213] · c-pointers-and-arrays[214] · c-preprocessor-and-avr-headers[215]
- binary-number-representations[216] — the signed/unsigned readings behind `char` and `unsigned char`
- floating-point-numbers[217] — what `float` and `double` hold, and their cost on the AVR · fixed-point-numbers[218] — fractions in integer arithmetic
- memory-models[219] — the 8-bit data cells a declaration reserves
- instruction-set-architecture[220] — the data types the AVR supports in hardware, and the ones it does not
- compilation-and-linking[221] — how the compiler maps globals, locals and statics to memory, and what happens after compiling · procedures-and-the-stack[222] — the stack frame a local lives in
- 2026-09-07-introduction-to-embedded-c-programming[223] — Lecture 13 · 2026-10-15-compilation-and-linking[224] — Lecture 22 · 2026-10-19-floating-point-numbers[225] — Lecture 23

C Operators

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[226]; the signed/unsigned shift rule is in 2026-09-10-more-c-programming-on-the-avr[227]. Types are in c-language-basics[228].

[212] courses/csse2010/c-operators.pdf
[213] courses/csse2010/c-control-flow.pdf
[214] courses/csse2010/c-pointers-and-arrays.pdf
[215] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[216] courses/csse2010/binary-number-representations.pdf
[217] courses/csse2010/floating-point-numbers.pdf
[218] courses/csse2010/fixed-point-numbers.pdf
[219] courses/csse2010/memory-models.pdf
[220] courses/csse2010/instruction-set-architecture.pdf
[221] courses/csse2010/compilation-and-linking.pdf
[222] courses/csse2010/procedures-and-the-stack.pdf
[223] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[224] courses/csse2010/2026-10-15-compilation-and-linking.pdf
[225] courses/csse2010/2026-10-19-floating-point-numbers.pdf
[226] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[227] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[228] courses/csse2010/c-language-basics.pdf

Binary operators

Group	Operator	Meaning	AVR instruction behind it
arithmetic	`+`	addition	`add`, `adc`
	`-`	subtraction	`sub`, `sbc`
	`*`	multiplication	`mul`
	`/`	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[229].

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`

[229] 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[230] and the `AND`/`OR`/`EOR`/`COM` of avr-instructions[231].
- **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[232].
- **`==` 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.

[230] courses/csse2010/bit-masking.pdf
[231] courses/csse2010/avr-instructions.pdf
[232] 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[233] — 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[234].

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[235]). 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[236]). Combined with the shorthands above:

[233] courses/csse2010/bit-masking.pdf
[234] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[235] courses/csse2010/2026-09-03-instruction-set-architecture-part-2.pdf
[236] courses/csse2010/c-preprocessor-and-avr-headers.pdf

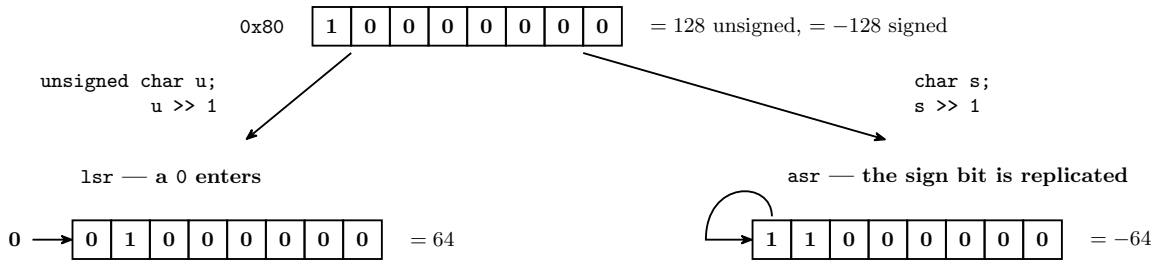


Figure 34: One operator, two instructions. The same bit pattern `10000000` is shifted right by one under two declarations. As `unsigned char` it means 128, the compiler emits `lsr`, a 0 enters at the top, and the result is 64. As `char` it means −128, the compiler emits `asr`, the sign bit is copied 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[237]).

Precedence

The one place precedence bites is the `SQUARE(1+2)` example in c-preprocessor-and-avr-headers[238], 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[239] · c-control-flow[240] · c-pointers-and-arrays[241] · c-preprocessor-and-avr-headers[242]
- bit-masking[243] — the set/clear/toggle/extract patterns the bitwise operators write
- avr-instructions[244] — the instructions each operator compiles to, including `lsl`/`lsr`/`asr`
- boolean-algebra[245] — the laws behind `&`, `|`, `^` and `~`
- floating-point-numbers[246] — integer `/` against floating point on the AVR: `PINA / 3` and `PINA * 0.334f`

[237] courses/csse2010/c-pointers-and-arrays.pdf
[238] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[239] courses/csse2010/c-language-basics.pdf
[240] courses/csse2010/c-control-flow.pdf
[241] courses/csse2010/c-pointers-and-arrays.pdf
[242] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[243] courses/csse2010/bit-masking.pdf
[244] courses/csse2010/avr-instructions.pdf
[245] courses/csse2010/boolean-algebra.pdf
[246] courses/csse2010/floating-point-numbers.pdf

- 2026-09-07-introduction-to-embedded-c-programming[247] — Lecture 13 · 2026-09-10-more-c-programming-on-the-avr[248] — Lecture 14

C Pointers and Arrays

Arrays, pointers, and the fact that connects them: **an array name is the address of its first element**, and **pointer arithmetic counts in elements, not cells**. Both are made concrete in the AVR's 8-bit data memory (memory-models[249]), where an `int` is two cells and so `a + 5` is ten cells past `a`. First met in 2026-09-10-more-c-programming-on-the-avr[250]; the language basics are in c-language-basics[251].

Arrays

Declaring: `type variable_name[size];`

```
char message[16];
int  values[10];
uint8_t seven_seg[10] = { 63,6,91,79,102,109,125,7,127,111 };   /* with initial
 ↪  values */
```

Accessing: `variable_name[index]`, with **index = 0 ... size−1** — **zero-based indexing**. A 365-element array has elements `[0]` through `[364]`; `[365]` is not one of them.

```
message[0] = 'c';
values[9] = values[8]++;        /* copies the OLD values[8], then increments it */
```

The declaration reserves `size` elements of the type **contiguously** — one after another with nothing between — which is what makes both the indexing and the pointer arithmetic below work. The elements are ordinary variables of the element type; `values[3]` is an `int` like any other.

C does not check the index. `values[10]` and `values[-1]` compile, and at run time they read or write whichever cell happens to sit past the end or before the start. The `seven_seg[digit]` lookup is guarded with `if(digit < 10)` for exactly this reason (c-control-flow[252]).

[247] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[248] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[249] courses/csse2010/memory-models.pdf
[250] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[251] courses/csse2010/c-language-basics.pdf
[252] courses/csse2010/c-control-flow.pdf

Pointers

A pointer variable stores a memory address. Declaration: `type *variable_name;` — *variable_name* is a pointer to something of the given *type*.

```
char a, b;
char *ptr;          /* or all on one line: char a,b,*ptr; */

ptr = &a;           /* & - address-of: ptr now holds a's address  */
b = *ptr;           /* * - indirection: the value at that address */
```

The two operators are inverses:

	From	To	
`&a`	a variable	its address	*address-of* — makes a pointer
`*ptr`	an address	the value stored there	*indirection* — follows a pointer

In memory, say `a` holds 8 at address 400, `b` is at 434, and `ptr` is at 576 holding **400** — an address, not a value (drawn in 2026-09-10-more-c-programming-on-the-avr[253]). `ptr` is a variable like the others, with its own cell; only its *contents* are special. On the AVR a data address is 16 bits (memory-models[254]), so a pointer is two cells whatever it points at.

`*ptr` can also be assigned to: `*ptr = 0;` writes through the pointer into whatever it points at. That is how the pointer traversal below clears an array.

The `*` in a declaration and the `*` in an expression are different things. `char *ptr;` declares `ptr` as a pointer; `*ptr` later *uses* it. The declaration form reads as *the thing `*ptr` is a `char`*.

The hardware view

A pointer is the **indirect addressing mode** of addressing-modes[255] as a language feature. `ptr` plays the part of the X, Y or Z register pair, `*ptr` is `ld rd, X`, and `*ptr = v` is `st X, rr`. `ptr++` is the post-increment form `ld rd, X+` that the instruction set provides precisely because walking a table is so common. The compiler chooses which of X, Y, Z it uses.

[253] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[254] courses/csse2010/memory-models.pdf
[255] courses/csse2010/addressing-modes.pdf

Arrays are pointers to their first element

An array name can be treated as a pointer to the first element — the address of the first element.

```
int a[10];
int *ptr;

/* the following are the same */
ptr = a;
ptr = &a[0];
```

So `a` on its own is an address, `a[0]` is the `int` there, and `*a` is the same `int`. In general **`a[i]` is `*(a + i)`** — the bracket notation is shorthand for pointer arithmetic, which is why the arithmetic has the rule it has.

Pointer arithmetic

Addition and subtraction on pointers work in multiples of the size of the object pointed to.

```
int a[10];
int *ptr;

/* the following are the same */
ptr = a+5;
ptr = &a[5];
```

If `a` starts at 300 and an `int` is 16 bits — two 8-bit cells — then `a + 5` is **310**, not 305: five elements on, two cells each. The sum is `300 + 2·5 = 310`, not `300 + 5 = 305`. The array is drawn in memory in 2026-09-10-more-c-programming-on-the-avr[256].

The consequences:

- **The same expression means different addresses for different types.** `a + 5` is 310 for `int a[]` and 305 for `char a[]`. The type is part of the arithmetic.
- **`ptr - a` counts elements**, not cells: for `ptr = &a[5]` it is 5.
- **Comparison works in addresses.** `ptr < a + 10` is true while `ptr` is inside the array.
- `int` is 16 bits *on the AVR*; on a 32-bit machine `a + 5` would be twenty cells on. The rule is stated in terms of *the size of the object* because the number is machine-dependent (c-language-basics[257]).

[256] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[257] courses/csse2010/c-language-basics.pdf

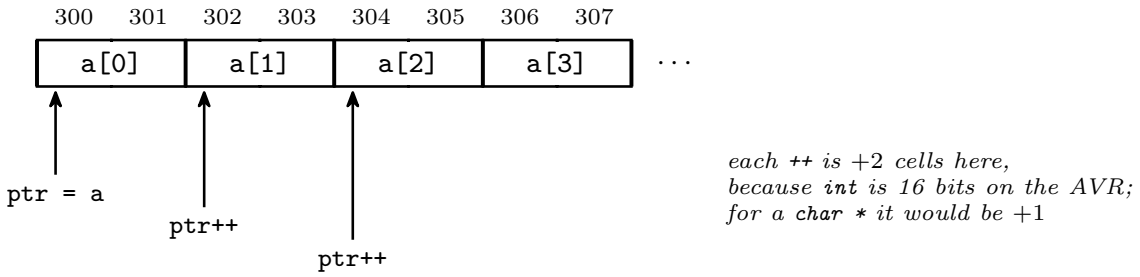


Figure 35: `ptr++` on a pointer to `int`, on the AVR. Each step moves the pointer by one *element*, which is two 8-bit cells, so the addresses it visits are 300, 302, 304, … and it lands on every element exactly once. The same `ptr++` on a `char` pointer would step by one cell. The compiler multiplies the step by the element size; the source says `++` either way. The pointer is the traversal loop's, drawn at three successive steps.

Traversing an array

Two ways to clear `int a[10]`:

```
/* using an index */
int a[10];
int index;
for(index=0; index<10; index++) {
    a[index] = 0;
}

/* using a pointer */
int a[10], *ptr;
for(ptr=a; ptr < a+10; ptr++) {
    *ptr = 0;
}
```

Adding one to an array pointer makes it point to the next element in the array. The two loops visit the same ten cells in the same order. The index loop computes `a + index` afresh each pass; the pointer loop keeps that address in `ptr` and steps it. The bound is the same in both — the address one past the last element, tested with `<` so it is never touched. See c-control-flow[258] for the `for` itself.

The pointer form is closer to what the compiler emits either way — a register pair holding an address, an `st X+, r` in the body — which is why C has it.

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

Strings, briefly

A string constant `"abc"` (c-language-basics[259]) is an array of `char` with a terminating zero byte after the last character, so `"abc"` occupies four cells. `char message[16];` therefore holds at most fifteen characters, though `message[16]` looks like room for sixteen. The terminator is standard C.

See also

- c-language-basics[260] · c-operators[261] · c-control-flow[262] · c-preprocessor-and-avr-headers[263]
- memory-models[264] — 8-bit data cells, 16-bit addresses, alignment
- addressing-modes[265] — indirect addressing, which a pointer is
- avr-instructions[266] — `ld`/`st` with X, Y, Z and their post-increment forms
- seven-segment-display[267] — the `seven_seg[10]` lookup table
- 2026-09-10-more-c-programming-on-the-avr[268] — Lecture 14

C Preprocessor and AVR Headers

The preprocessor — `#include` and `#define` — and the three lines of `avr/io.h` that let `DDRA = 0xFF;` be written as if a hardware register were a variable. First met in 2026-09-10-more-c-programming-on-the-avr[269]; the port registers themselves are in avr-io-ports[270].

What the preprocessor is

C programs are first parsed by a preprocessor, which **replaces preprocessor directives with substitute text**. Only then does the compiler see the program. Directives start with `#`:

[259] courses/csse2010/c-language-basics.pdf
[260] courses/csse2010/c-language-basics.pdf
[261] courses/csse2010/c-operators.pdf
[262] courses/csse2010/c-control-flow.pdf
[263] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[264] courses/csse2010/memory-models.pdf
[265] courses/csse2010/addressing-modes.pdf
[266] courses/csse2010/avr-instructions.pdf
[267] courses/csse2010/seven-segment-display.pdf
[268] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[269] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[270] courses/csse2010/avr-io-ports.pdf

Directive	Does
`#include …`	paste in a file
`#define …`	define a macro — a textual substitution
`#if`, `#ifdef`, `#else`, `#undef`, …	conditional inclusion, un-definition, and others

The preprocessor does not know C. It edits text. Whatever it produces is what the compiler compiles, and every surprise in this note comes from forgetting that.

`#include`

Form	Kind of header	Looks in
`#include <filename.h>`	system	the system include directories — where `avr/io.h` and `stdio.h` live
`#include "filename.h"`	user-defined	your own project directory first

The directive is replaced by the whole contents of the named file, at that point. After preprocessing there is one file, with the header pasted in at the top.

Header files

Header (`.h`) files are used for common macros, function prototypes etc — the C counterpart of `m324Adef.inc` in assembly (avr-io-ports[271]).

- `stdio.h` — prototypes of `printf()` etc. The AVR has a simplified version (no files); `printf()` only works if you say where the characters go, e.g. a serial port.
- `avr/io.h` — defines the AVR's useful constants and functions/macros. **Definitions depend on the actual device targeted**: it reads which chip is being compiled for and pulls in that chip's register map, so `DDRA` means the right address for the ATmega324A and a different one for another part. It also supplies `uint8_t` and the other fixed-width types.

Every program in the course starts with `#include <avr/io.h>`.

[271] courses/csse2010/avr-io-ports.pdf

Headers against libraries

A header file is not the library. A header

- **defines** useful preprocessor **macros**,
- **declares** the functions that are in the library, as **function prototypes**, and
- **declares** variables that are available externally, with `extern`.

The library is the collection of object files that implements the functions (compilation-and-linking[272]).

A prototype such as `int printf(const char *fmt, ...);` tells the compiler the name, the argument types and the return type. That is enough for the compiler to check a call and generate it. There is no code in it, so in `main.o` the call to `printf` is left as an **external reference**. The code is in the library, an object file `printf.o` inside it, and the **linker** joins it on and fills in the address. An `extern` declaration does the same for a variable: `extern int count;` says that `count` exists somewhere else and reserves nothing.

So the header goes to the compiler, through the preprocessor, and the library goes to the linker. The two kinds of mistake show up at different steps:

Missing	Shows up when	Typical message
the `#include`	compiling	implicit declaration of function `printf`
the library, or the `.c` file that defines the function	linking	undefined reference to `printf`

`#define` — macros

Macros are textual substitutions.

Macro	Effect
`#define pi 3.1415`	every later `pi` becomes `3.1415` — `A = pi * r²;` becomes `A = 3.1415 * r²;` (the superscript is shorthand; in real C the square would be `r * r`)
`#define EVER ;;`	`for(EVER)` becomes `for(;;)`, an infinite loop
`#define sum(a,b) (a+b)`	a macro **with parameters**: `sum(x, 2)` becomes `(x+2)`

[272] courses/csse2010/compilation-and-linking.pdf

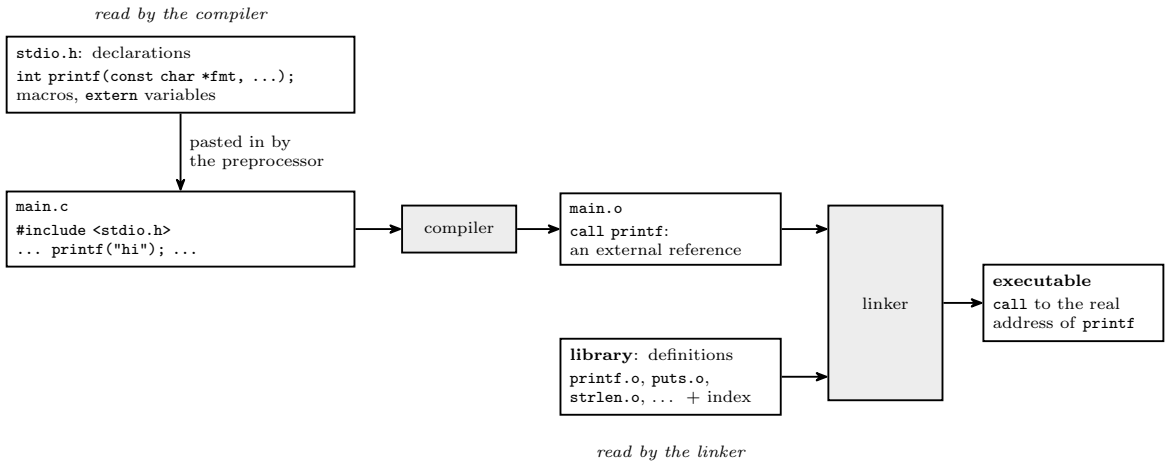


Figure 36: Where a header and a library each come into play, for a program that calls printf. The header holds only declarations. The preprocessor pastes it into main.c so that the compiler can check and generate the call, and in main.o that call is still an external reference. The library holds the definitions, as already compiled object files with an index. The linker finds printf.o through the index, links it in, and fills in the call. The header is used by the compiler and the library by the linker.

A macro with parameters looks like a function call and is not one: no call happens, the arguments are pasted into the body as text, and the result is compiled in place.

The `SQUARE` trap

Given `#define SQUARE(x) (x*x)`, what is `SQUARE(1+2)`?

5, not 9. Paste the text: `(1+2*1+2)`. Multiplication binds tighter than addition, so it is $1+2+2$. The argument was never evaluated first, because there is no function to evaluate it.

The fix is **parentheses around every parameter, and around the whole body**:

```
#define SQUARE(x) ((x)*(x))        /* SQUARE(1+2) -> ((1+2)*(1+2)) = 9 */
```

The outer pair matters as much as the inner: without it, `1 / SQUARE(a)` becomes `1 / (a)*(a)`, which is a, not $1/a^2$. **Write every macro this way.** The AVR headers below do — note the brackets around `(mem_addr)` and `(io_addr)`.

A second pitfall: an argument with a side effect is pasted **as many times as the parameter appears**. `SQUARE(i++)` becomes `((i++)*(i++))` and increments `i` twice.

The `avr/io.h` register macros

The definitions behind the port names, from `avr/io.h` and the device header it includes:

```
#define _MMIO_BYTE(mem_addr) \
                (*(volatile uint8_t *)(mem_addr))
#define _SFR_IO8(io_addr) \
                _MMIO_BYTE((io_addr) + 0x20)

/* Input Pins, Port A */
#define PINA      _SFR_IO8(0x00)

/* Data Direction Register, Port A */
#define DDRA      _SFR_IO8(0x01)

/* Data Register, Port A */
#define PORTA     _SFR_IO8(0x02)
```

(\ at the end of a line continues a directive onto the next line.)

Upshot: I/O register names can be used as variables.

```
DDRA = 0xFF;
PORTA = PINC | 0xF0;
```

Unrolling `DDRA`

Step	Text
as written	`DDRA = 0xFF;`
`#define DDRA`	`_SFR_IO8(0x01) = 0xFF;`
`#define _SFR_IO8`	`_MMIO_BYTE((0x01) + 0x20) = 0xFF;`
`#define _MMIO_BYTE`	`(*(volatile uint8_t *)((0x01) + 0x20)) = 0xFF;`

Reading the final line from the inside out:

1. `(0x01) + 0x20` is `0x21` — a number.
2. `(volatile uint8_t *)` casts it: *treat this number as the address of an 8-bit unsigned byte.* Now it is a pointer (c-pointers-and-arrays[273]).
3. The leading `*` dereferences it: *the byte at that address.*
4. `= 0xFF` stores into that byte.

[273] courses/csse2010/c-pointers-and-arrays.pdf

So `DDRA = 0xFF;` is a store to data address `0x21`, and `x = PINC;` is a load from data address `0x26`. There is no variable called `DDRA` anywhere; the name dissolves before the compiler sees it.

Why `+ 0x20`

The AVR's 64 I/O registers have **two addresses each** (avr-io-ports[274]): an *I/O-space* number 0–63, which is what `in` and `out` use, and a *data-space* address `0x20` higher, where `lds`/`sts` — and any C pointer — find the same register. `DDRA` is I/O `0x01` and data `0x21`. The macro takes the I/O number the datasheet lists and adds the offset, because a C pointer can only name data-space addresses. (The compiler, seeing a constant address in the I/O range, will usually emit `in`/`out` anyway.)

Why `volatile`

`volatile` tells the compiler **the value at this address can change without the program changing it**, and that **every read and write matters**. Without it the compiler is entitled to assume that a byte it wrote a moment ago still holds what it wrote, and that two reads with nothing between them give the same answer — both false for a port: `PINC` changes whenever the pins do, and `PORTA` is written for its effect on the pins, not to be read back. `volatile` forbids the optimisations that would break that. It is the one piece of the macro that is about hardware rather than addressing.

The same keyword belongs on any ordinary variable that an **interrupt handler** writes. To the code in `main`, a handler's write is as invisible as a pin changing, and for the same reason: nothing in `main`'s own flow of control explains it. See avr-interrupts[275].

Naming bits

The headers also define **bit numbers** for every flag and pin — for example `FLAG_NAME == 5` — so a mask is written from a name rather than a constant:

```
1 << FLAG_NAME              /* 00100000 when FLAG_NAME is 5 */
PORTA |= (1 << 5);          /* set one bit                   */
PORTA &= ~(1 << 5);         /* clear one bit                 */
```

[274] courses/csse2010/avr-io-ports.pdf
[275] courses/csse2010/avr-interrupts.pdf

The shift and the `|=`/`&=` shorthands are in c-operators[276]; the mask patterns are in bit-masking[277]. `FLAG_NAME` is a placeholder name; the actual names for the ATmega324A's pins and flags come from the device header, as `#define WGM02 3` does in 2026-10-05-more-io[278].

See also

- avr-io-ports[279] — the port registers, and the I/O-space versus data-space address columns
- c-pointers-and-arrays[280] — what `*(uint8_t *)addr` means
- c-operators[281] — `1 << n`, `|=`, `&=`
- bit-masking[282] · memory-models[283]
- avr-interrupts[284] — `volatile` on variables shared with an interrupt handler
- compilation-and-linking[285] — what happens after the preprocessor: object files, libraries and the linker · 2026-10-15-compilation-and-linking[286] — Lecture 22
- 2026-09-10-more-c-programming-on-the-avr[287] — Lecture 14

Compilation and Linking

How a C program turns into the machine code in the AVR's flash. This note covers compiling against interpreting, where the compiler puts global, local and `static` variables, object files and their six sections, the linker's two jobs (relocating and resolving), libraries, how header files differ from libraries, and static against dynamic linking. The assembler's part, with its two passes and symbol table, is in assembly-process[288]. First met in 2026-10-15-compilation-and-linking[289].

Compiled or interpreted

A processor runs machine code and nothing else. The fetch-decode-execute cycle (fetch-decode-execute-cycle[290]) reads instruction words from memory, and nothing in it knows the program

[276] courses/csse2010/c-operators.pdf
[277] courses/csse2010/bit-masking.pdf
[278] courses/csse2010/2026-10-05-more-io.pdf
[279] courses/csse2010/avr-io-ports.pdf
[280] courses/csse2010/c-pointers-and-arrays.pdf
[281] courses/csse2010/c-operators.pdf
[282] courses/csse2010/bit-masking.pdf
[283] courses/csse2010/memory-models.pdf
[284] courses/csse2010/avr-interrupts.pdf
[285] courses/csse2010/compilation-and-linking.pdf
[286] courses/csse2010/2026-10-15-compilation-and-linking.pdf
[287] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[288] courses/csse2010/assembly-process.pdf
[289] courses/csse2010/2026-10-15-compilation-and-linking.pdf
[290] courses/csse2010/fetch-decode-execute-cycle.pdf

was once written in C. A program in a high-level language reaches the processor in one of two ways.

- **Compilation** translates the whole program into machine code before it runs, often by way of assembly language. The machine code then runs directly on the processor. C works this way.
- **Interpretation** never turns the program into machine code. Another program, the **interpreter**, reads the source and carries out each statement when it gets to it. The interpreter is itself machine code, and it is the interpreter that the processor runs. Python, Java and MATLAB work this way.

	Compiled	Interpreted
What the processor runs	the program's own machine code	the interpreter's machine code
When translation happens	once, before the program runs	every time a statement is executed
Speed	fast	**slow**
Portability	the machine code suits one processor only	**runs anywhere** an interpreter exists

Why interpreting is slow. Every time a statement is executed, the interpreter has to work out again what the statement means and then carry it out. That costs many of its own instructions per statement, and inside a loop it is repeated on every pass. A compiler does that work once, ahead of time.

Why it is portable. The program never becomes machine code, so nothing in it is tied to one instruction set. Moving it to a new processor only needs an interpreter built for that processor.

Bytecode and just-in-time compilation

Java and Python take a route in between. The source is first compiled, but not into machine code. It becomes **bytecode**, the instruction set of an imaginary machine. Bytecode is the same on every platform, so a compiled Java program is portable across all of them.

The bytecode is then run by a **virtual machine**, which is an interpreter for bytecode. The Java Virtual Machine is the one for Java. Running bytecode is faster than interpreting source text, because the parsing has already been done, but it is still interpretation.

To win back the speed, the virtual machine watches which parts of the program run most often and compiles them into real machine code while the program is running. This is **just-in-time (JIT) compilation**. Once a part has been JIT-compiled, it runs directly on the processor like

compiled code. Standard Python interprets its bytecode, and a JIT compiler such as Numba can compile chosen Python functions to machine code in the same way.

The toolchain

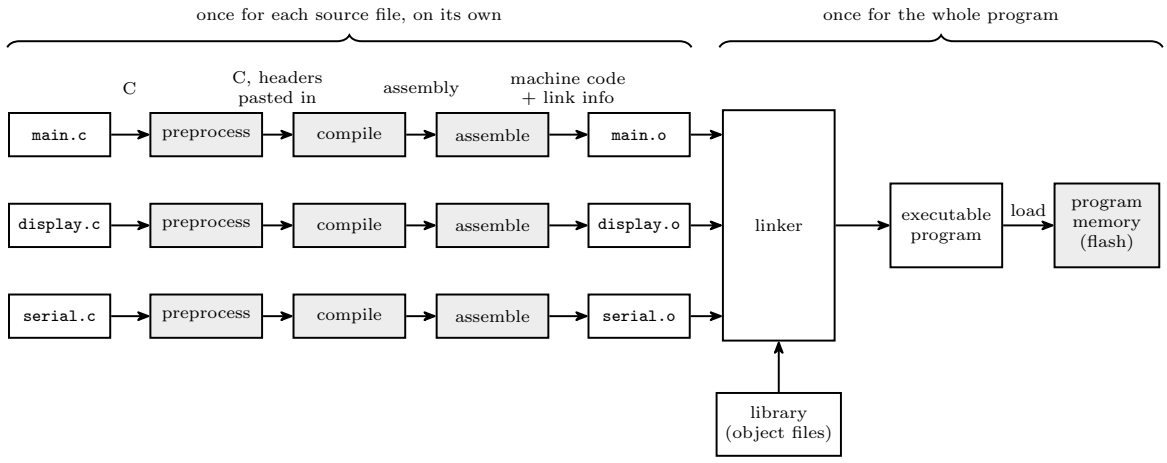


Figure 37: How a three-file C program becomes a running program. Each source file goes through the first three steps by itself. The preprocessor pastes in headers and expands macros, the compiler turns C into assembly, and the assembler turns assembly into an object file, which is machine code plus the information needed to link it. Only the linker sees the whole program. It joins the object files to each other and to the library members they use, fixes up every address, and writes one executable. That executable is then loaded into memory, which on the AVR means programming it into flash. Changing one source file means repeating the first three steps for that file alone, then linking again.

Step	Takes	Produces	Notes
Preprocess	a `.c` file	C with every `#include` pasted in and every macro expanded	c-preprocessor-and-avr-headers[291]
Compile	that C	assembly language	maps variables to memory, and translates each statement into instructions

[291] courses/csse2010/c-preprocessor-and-avr-headers.pdf

Step	Takes	Produces	Notes
Assemble	the assembly	an **object file**, `.o`	assembly-process[292]
Link	all the object files, plus the library	one **executable**	relocates and resolves addresses
Load	the executable	the program in memory, ready to run	on the AVR, the executable is programmed into flash

Some compilers write machine code directly and never produce assembly text. The object file that comes out is the same either way, so the three middle steps are often just called "compiling". In the simplified view the compiler itself does two things, **mapping variables to memory** and **translating C into assembler instructions**. Turning those instructions into machine code is the assembler's work.

With the AVR GCC toolchain the object files and the executable are ELF files. The `.hex` file that gets programmed into flash is extracted from the ELF executable.

How a compiler maps variables to memory

The compiler decides where each variable goes by **how long it has to exist**.

Declared	Lives	Stored	Address
outside every function: a **global**	for the whole run	in the fixed data area, which starts at the bottom of SRAM	fixed before the program runs
inside a function: a **local**	only while that call is running	in the call's **stack frame**	depends on how deep the stack is at the call
inside a function, marked `static`	for the whole run, and it keeps its value between calls	with the globals	fixed
nowhere: obtained from `malloc()`	until it is given back	on the **heap**	handed out while the program runs

Globals. A global can be used at any moment from any function, so it needs one address that never changes. The compiler reserves as many bytes as the type needs (c-language-basics[293]), one variable after another, from the start of SRAM. On the ATmega324A that is `0x0100` = 256,

[292] courses/csse2010/assembly-process.pdf
[293] courses/csse2010/c-language-basics.pdf

because addresses 0 to 255 are taken by the registers and the I/O registers (memory-models[294]). It is the same thing an assembly programmer does with `.dseg` and `.byte`.

Locals. A local exists only while its function is running, and a recursive function can have several calls running at once, each needing its own copy. So the compiler cannot give it a fixed address. Instead, each call builds a **stack frame**, a block on the stack holding that call's arguments, its return address and its local variables. The return takes the frame away again (procedures-and-the-stack[295]). The same local therefore sits at different addresses on different calls.

`static` locals. `static` inside a function keeps the name private to the function but gives the variable a global's lifetime. It is stored with the globals, at a fixed address, and is never part of a frame. That is why it still holds its value the next time the function is called.

`static` on a global means something else. The variable is stored exactly like any global, but its name is private to its own source file. It is left out of the object file's entry point table (below), so no other file can link to it.

The heap and the stack grow toward each other. The heap starts above the globals and grows toward higher addresses. The stack starts at the top of SRAM (`RAMEND`) and grows down. The free memory between them shrinks from both ends, and if the two meet, that is a **stack overflow**. Nothing on the chip detects it. The first sign is usually a global or heap variable changing for no reason.

Object files

A program of any size is split into several source files, and it pays to **compile or assemble each one separately and then link them together**. Change one file and only that file has to be compiled again. The others' object files are reused as they are.

The price is that **when a file is compiled, nobody knows where its code will end up**. The file is translated as if it started at address 0. Every address inside it is relative to its own start, and any symbol defined in another file has no address at all yet. Once the linker has decided where everything goes, these addresses have to be fixed. A file that carries enough information for that fixing is a **relocatable object module**. "Object module" and "object file" mean the same thing.

So an object file is **machine code plus information for the linker**. There are many formats for it: `a.out`, COFF and ELF are object file formats, and DWARF is a format for debugging information, usually carried inside ELF files.

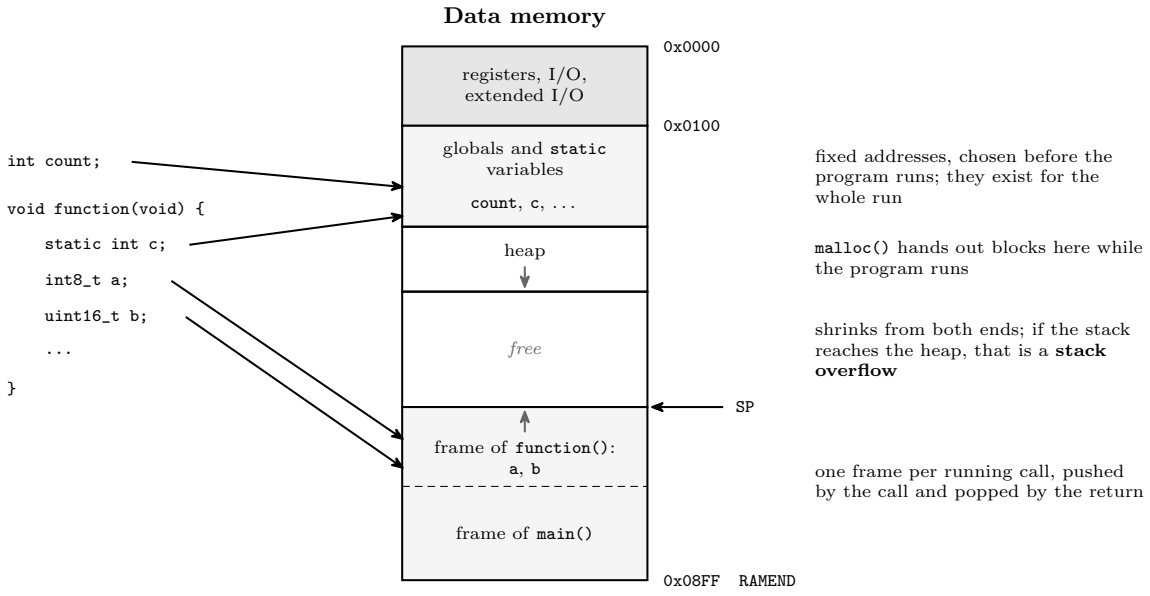


Figure 38: Where each kind of C variable ends up in the ATmega324A's data memory, with the lowest address at the top. Globals and static variables are given fixed addresses from 0x0100, the first SRAM byte, so a static local such as c sits among the globals even though only its own function can name it. The heap, which malloc() hands out while the program runs, grows down the page toward the stack. The stack grows up the page from RAMEND. Each running call has a frame on it, and the frame of function() holds its locals a and b. When function() returns, that frame is gone. If the stack and the heap meet, the stack has overflowed.

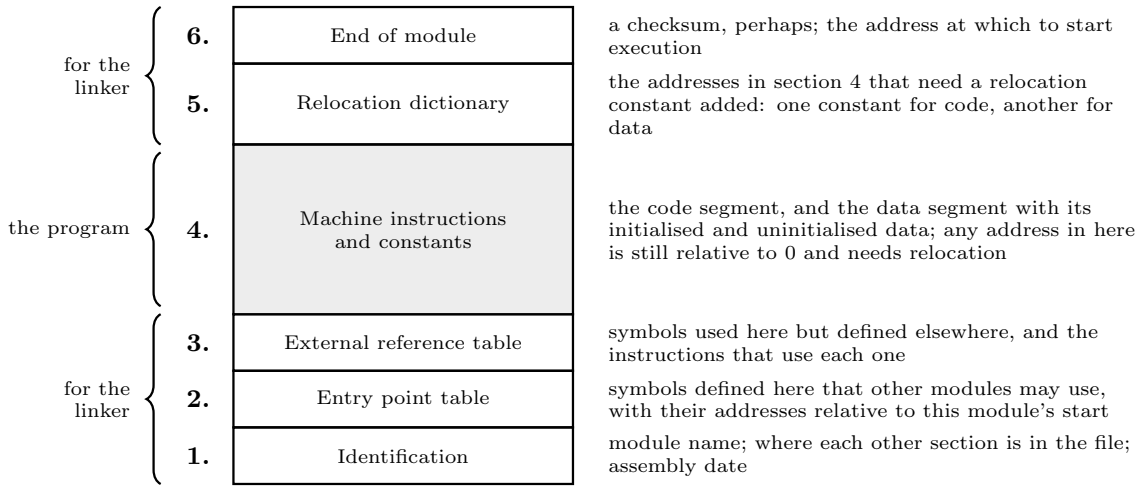


Figure 39: The six sections of a relocatable object module, numbered from the bottom. Only section 4 is program: the machine instructions and constants, assembled as if the module began at address 0. The other five are information for the linker. They say what the module is called and where its parts are in the file (1), which of its symbols other modules may use (2), which symbols it uses but does not define (3), which addresses in section 4 have to be moved when the module is moved (5), and where execution starts (6). Linking reads 1, 2, 3, 5 and 6 to decide how to rewrite 4.

The six sections

1. Identification. The name of the module, where each of the other sections is in the file, and the date it was assembled.

2. Entry point table. The symbols **defined in this module that other modules may refer to**, with their values. A value is an address: a procedure's entry point, or a variable's location. It is **relative to the beginning of this object module**. A file that defines a procedure `abs_value` at its address 10 has the entry `abs_value 10`. Anything not listed, such as a `static` global or a label used only inside the module, cannot be reached from another file.

3. External reference table. The symbols **used in this module but not defined in it**, both procedure names and variables, and for each one **the list of instructions that use it**. Those instructions are the places the linker will have to fill in once it knows the address.

One module's entry point is another module's external reference. If module A defines `SUM` and module B calls it, `SUM` appears in A's entry point table and in B's external reference table.

4. Machine instructions and constants. The assembled code and constants: the **code segment**, and the **data segment** holding the initialised and the uninitialised data. An initialised global such as `int n = 5;` needs its starting value stored in the file. An uninitialised one needs only its size. **The addresses used in this code are not yet correct.** They were worked out as if the module started at 0, so they need relocation.

5. Relocation dictionary. The list of addresses in section 4 that need to be relocated. When the module is placed, a **relocation constant** is added to the instruction or data at each of those addresses. There is **one constant for code and a different one for data**, because a module's code goes in program memory and its data in data memory, and the two start in different places.

6. End of module. Possibly a checksum, to catch a corrupted file, and the address at which execution starts.

The linker

The **linker** combines object modules, the program's own and any it needs from libraries, into a single **executable binary program** of machine code.

Its problem is the assembler's problem again, one level up. That is the **external reference problem**: symbols are used before their values are known. Module A can call a procedure in module B before the linker has even read B. So the linker, like the assembler, works in **two passes** (assembly-process[296]).

[294] courses/csse2010/memory-models.pdf
[295] courses/csse2010/procedures-and-the-stack.pdf
[296] courses/csse2010/assembly-process.pdf

Pass 1: lay the program out.

1. Read each module's sizes and give it a place, in link order. Each module's code starts where the previous module's code ended, and the same goes for data. These starting addresses are the module's two **relocation constants**.
2. Read every entry point table. Add each symbol to one **global symbol table**, with its final address: its value relative to its module, plus that module's starting address.

Pass 2: copy and patch.

1. Copy each module's section 4 to its place.
2. **Relocate.** For each address in the module's relocation dictionary, add the module's constant, the code constant for a code address and the data constant for a data address.
3. **Resolve.** For each external reference, look the symbol up in the global table and write its final address into every instruction listed against it. A symbol that no module defines cannot be resolved, so linking fails with an "undefined reference" error.

The two words to keep straight:

> **Relocating** = adding a constant: fixing an address that points into the module's own code or data, which moved when the module moved. **Resolving** = finding the value of an external address: filling in an address that points into another module.

Where a module starts follows from the order alone. **A module's code starts at the program start plus the code sizes of every module linked before it**, and its data starts at the data start plus the data sizes of those same modules. Code and data are counted separately, in their own units: on the AVR, words of program memory and bytes of data memory (memory-models[297]).

Worked: linking four modules

Four object modules, each assembled from address 0. Their lengths are A = 400, B = 600, C = 500, D = 300. Each holds a branch within itself, a data move, and (apart from D) a call to the next module.

Module	At	Instruction
A	0	`BRANCH TO 200`
A	200	`MOVE P TO X`
A	300	`CALL B`
B	0	`BRANCH TO 300`
B	300	`MOVE Q TO X`
B	500	`CALL C`

[297] courses/csse2010/memory-models.pdf

Module	At	Instruction
C	0	`BRANCH TO 200`
C	200	`MOVE R TO X`
C	400	`CALL D`
D	0	`BRANCH TO 200`
D	200	`MOVE S TO X`

They are linked in the order A, B, C, D. The first 100 addresses hold start-up code, the code that runs before `main` (on the AVR it sets the stack pointer and prepares the data), so A starts at 100.

Pass 1: place the modules. Each starts where the previous one ends.

Module	Length	Starts at	Ends at	Relocation constant
A	400	100	499	+100
B	600	100 + 400 = **500**	1099	+500
C	500	500 + 600 = **1100**	1599	+1100
D	300	1100 + 500 = **1600**	1899	+1600

The global symbol table holds the module names as entry points: A = 100, B = 500, C = 1100, D = 1600. The program ends at 1899, and 1900 is the first free address.

Pass 2: patch each instruction. An instruction moves to its offset plus its module's start. A branch inside a module is **relocated**: the target moves by the same constant as the instruction, so it still lands on the same `MOVE`. A call to another module is **resolved** from the symbol table.

New address	Was	Becomes	Because
100	A: `BRANCH TO 200`	`BRANCH TO 300`	relocate: 200 + 100
300	A: `MOVE P TO X`	`MOVE P TO X`	no code address in it
400	A: `CALL B`	`CALL 500`	resolve: B starts at 500
500	B: `BRANCH TO 300`	`BRANCH TO 800`	relocate: 300 + 500
800	B: `MOVE Q TO X`	`MOVE Q TO X`	no code address in it
1000	B: `CALL C`	`CALL 1100`	resolve: C starts at 1100
1100	C: `BRANCH TO 200`	`BRANCH TO 1300`	relocate: 200 + 1100
1300	C: `MOVE R TO X`	`MOVE R TO X`	no code address in it
1500	C: `CALL D`	`CALL 1600`	resolve: D starts at 1600
1600	D: `BRANCH TO 200`	`BRANCH TO 1800`	relocate: 200 + 1600
1800	D: `MOVE S TO X`	`MOVE S TO X`	no code address in it

B's constant, 500, is the start-up code plus all of A: 100 + 400. Every branch ends up pointing at the instruction it pointed at before: A's branch at 100 reaches A's `MOVE` at 300, and B's at 500 reaches B's `MOVE` at 800. The `MOVE`s name data, `P`, `Q`, `R`, `S` and `X`. Those addresses are fixed in the same way with each module's data constant, which this example leaves out so that only the code is followed.

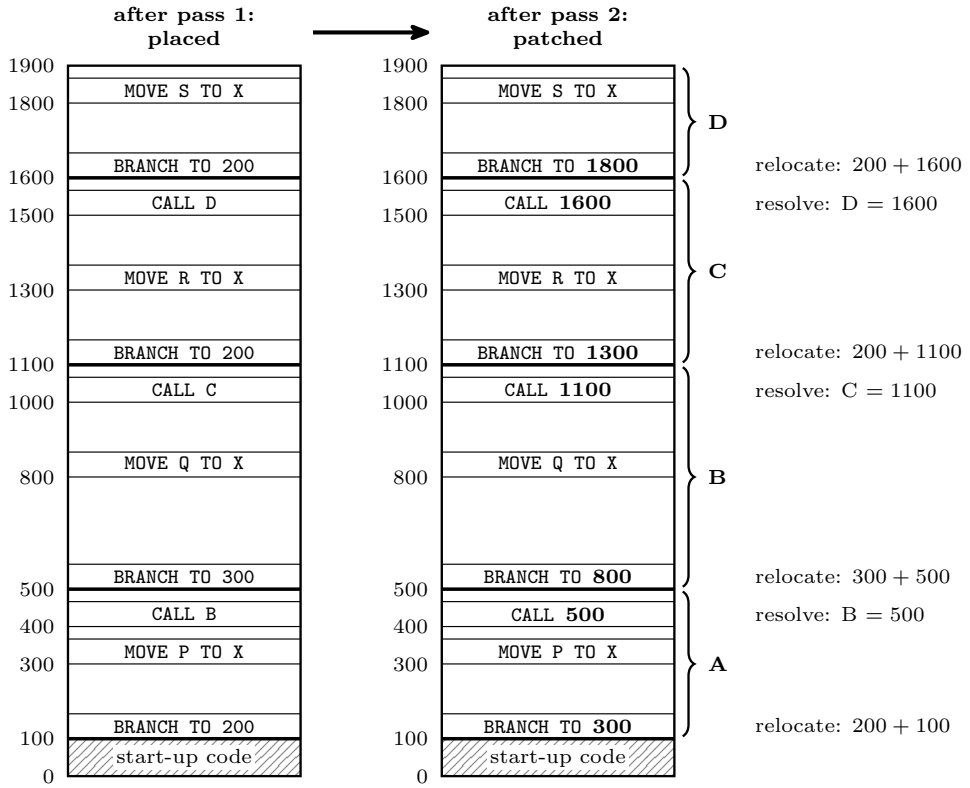


Figure 40: Linking four object modules in the order A, B, C, D, with addresses increasing up the page and start-up code in 0 to 99. Left: after the first pass. Every module has its place, each starting where the one before it ends, but the instructions are still exactly as assembled. A's branch at 100 still names 200, its target counted from A's own 0, and 200 is now in the middle of A, not at its MOVE. Each call still names a module, not an address. Right: after the second pass. Each branch within a module has been relocated by its module's starting address, so it lands on that module's MOVE again, and each call to another module has been resolved to that module's start. The bold operands are the ones the linker rewrote.

Worked: where each module starts

Three modules have these sizes. Code is measured in words, because AVR program memory holds 16-bit words, and data in bytes.

Module	Code size (words)	Data size (bytes)
1	45	17
2	67	5
3	100	25

They are linked in the order **2, 1, 3**, starting at program address 62 and data address 256. Where does each module's code and data start, and where do two sample references inside module 1 end up?

Lay them out in link order, keeping a running total for each memory:

Module	Code	Data	Code constant	Data constant
2	62 to 62 + 67 − 1 = 128	256 to 256 + 5 − 1 = 260	+62	+256
1	129 to 129 + 45 − 1 = 173	261 to 261 + 17 − 1 = 277	+129	+261
3	174 to 174 + 100 − 1 = 273	278 to 278 + 25 − 1 = 302	+174	+278

The next free code word is 274 and the next free data byte is 303. Both totals come out the same as for any other order, because the same three sizes are added up. Only where each module falls changes.

Relocate inside module 1. Suppose module 1 has, at its own word 20, a branch to its own word 40, and at its own word 30, a load from its own data byte 4.

- The branch moves to 20 + 129 = **149**, and its target is relocated by the same code constant, 40 + 129 = **169**.
- The load moves to 30 + 129 = 159, and its operand is a data address, relocated by the **data** constant: 4 + 261 = **265**.

Adding the code constant to the data address would give 133, and the load would read the wrong byte. This is why the relocation dictionary has to say whether each entry is a code address or a data address.

Why 62 and 256? On the ATmega324A, program words 0 to 61 (`0x0000` to `0x003D`) are the interrupt vector table (avr-interrupts[298]), so `0x003E` = 62 is the first free word. Data addresses 0 to 255 are registers and I/O, so `0x0100` = 256 is the first byte of SRAM.

Libraries

A **library** is a **collection of object files with an additional index**. It provides useful functions and procedures that have already been written, tested and compiled. The rule is not to reinvent the wheel: use library code where it exists.

The index maps each symbol to the member object file that defines it. When an external reference is still unresolved, say a call to `printf`, the linker looks the name up in the index, takes that one member, and links it in like any other module. The rest of the library stays out of the program.

The **standard C library** is the one every C program can count on. It covers:

- standard input and output (`stdio.h`: `printf`, `getchar`, …)
- string handling (`string.h`: `strlen`, `strcpy`, …)
- date and time functions (`time.h`)
- general utilities (`stdlib.h`: `malloc`, `abs`, `atoi`, …)

On the AVR it is provided by avr-libc, which also supplies the `avr/io.h` and `avr/interrupt.h` headers.

Header files are not the library

The two are easy to mix up, because `#include <stdio.h>` is what lets a program call `printf`.

A **header file** (`.h`)

- defines useful **preprocessor macros**,
- **declares** the functions that are in the library, as **function prototypes**, and
- **declares** variables that are available externally, with **`extern`**.

The **library** is the collection of **object files that implement** those functions.

A declaration says what a name is and what type it has, so the compiler can check every use and generate the call. It contains no code, so the call still has nowhere to go. The header is therefore read by the **compiler**, and the library is read by the **linker**. Each kind of mistake shows up at its own step:

[298] courses/csse2010/avr-interrupts.pdf

Missing	When it shows	Typical message
the header	compiling	implicit declaration of function `printf`
the library, or the file that defines the function	linking	undefined reference to `printf`

How a header gets pasted in is covered in c-preprocessor-and-avr-headers[299].

Static and dynamic linking

Everything so far is **static linking**. All the object code, the library's and the main program's, is joined into one binary before the program ever runs.

In **dynamic linking**, linking happens **at run time** instead: a library procedure is linked **when it is first called**. The executable holds only a reference to it, and the library stays in a separate file that is loaded when the program needs it. **DLLs** (dynamic-link libraries) on Windows work this way, and so do shared libraries (`.so` files) on Linux.

	Static	Dynamic
Library code is joined in	when the program is linked	when the procedure is first called
Library code is stored	inside every executable that uses it	once, in its own file, shared
Executable size	larger	smaller
Updating the library	means relinking each program	takes effect in every program at once

Its advantages:

- **It saves space.** Libraries are not linked into the executable, so ten programs that use the same library share one copy of it.
- **The library can be updated independently of the programs that use it.** A bug fixed in the DLL is fixed in all of them, without relinking anything.

The second advantage is also its weakness. A program tested with one version of a library can break when another installation replaces that library with a version that behaves differently. On earlier versions of Windows this was common enough to be called **"DLL hell"**.

The ATmega324A has no operating system to load a library at run time, so every AVR program is linked statically. Everything it will ever call is already in flash.

[299] courses/csse2010/c-preprocessor-and-avr-headers.pdf

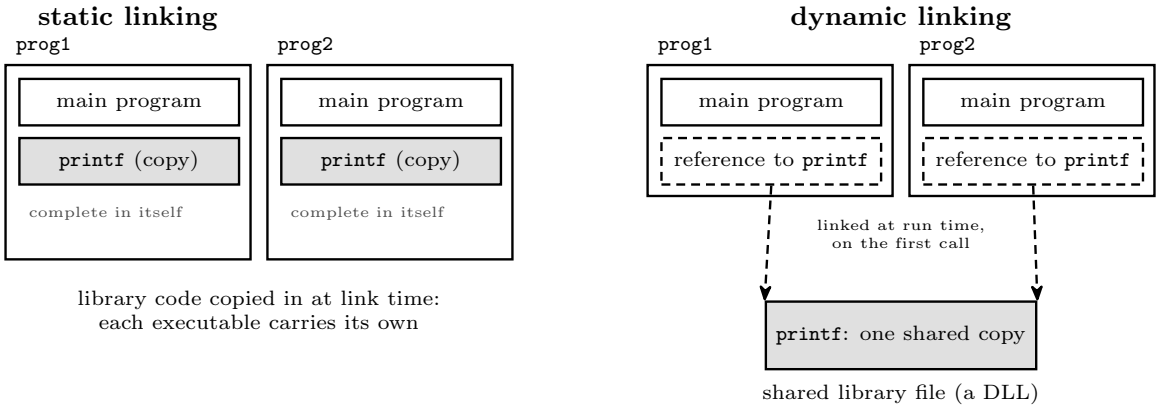


Figure 41: The same two programs, both calling printf, linked statically and dynamically. Static linking copies printf's object code into each executable when it is linked, so each program carries its own copy and is complete in itself. Dynamic linking leaves a reference in each executable and keeps a single copy of printf in a shared library file. The reference is linked to that copy at run time, when printf is first called. That saves space, and updating the shared file changes the printf both programs get, for better or worse.

Common mistakes

- **Treating the header as the library.** `#include <math.h>` makes `sqrt` compile. It does not supply `sqrt`. Without the library the link fails with an undefined reference.
- **Counting from the wrong module.** A module's start depends on the modules linked **before** it in link order, not on its number. In the order 3, 2, 1, module 2 starts after module 3.
- **Mixing code and data sizes.** The code start adds up code sizes only, and the data start adds up data sizes only. On the AVR they are not even in the same unit: words against bytes.
- **Off by one at a boundary.** A module of size n starting at s occupies s to $s + n - 1$, and the next module starts at $s + n$.
- **Relocating an external reference.** A call into another module is not fixed by adding *this* module's constant. Its address comes from the symbol table, which holds the *other* module's start plus the symbol's offset in it.
- **Relocating a data address with the code constant**, or the other way round.
- **Expecting a local variable to keep its value between calls.** The frame is gone when the function returns. Use `static` if the value must persist.
- **Returning a pointer to a local.** The address points into a frame that no longer exists, and the next call will overwrite it.
- **Thinking "interpreted" means "never compiled".** Java is compiled, to bytecode,

and a JIT compiles parts of it again to machine code. What makes it interpreted is that the virtual machine, and not the processor, runs the bytecode.

See also

- assembly-process[300] — the assembler's two passes and symbol table, the same problem one level down
- c-preprocessor-and-avr-headers[301] — `#include`, macros, and the header against the library
- procedures-and-the-stack[302] — stack frames, `call` and `ret`, and where locals live
- c-language-basics[303] — declarations and how much memory each type takes
- memory-models[304] · avr-architecture[305] — the AVR's word-addressed program memory and byte-addressed data memory
- avr-interrupts[306] — the vector table that the program code follows
- instruction-set-architecture[307] · 2026-09-07-introduction-to-embedded-c-programming[308] — compiling one-to-many against assembling one-to-one
- fetch-decode-execute-cycle[309] — what "runs on the processor" means
- 2026-10-15-compilation-and-linking[310] — Lecture 22

Fetch-Decode-Execute Cycle

What the control unit actually does: a continuous cycle over the datapath, issuing control signals in the right order for whatever instruction it just read. See cpu-organisation[311] for the blocks involved and arithmetic-logic-unit[312] for the one being driven.

The cycle

The control unit operates in a **continuous cycle** on the datapath:

[300] courses/csse2010/assembly-process.pdf
[301] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[302] courses/csse2010/procedures-and-the-stack.pdf
[303] courses/csse2010/c-language-basics.pdf
[304] courses/csse2010/memory-models.pdf
[305] courses/csse2010/avr-architecture.pdf
[306] courses/csse2010/avr-interrupts.pdf
[307] courses/csse2010/instruction-set-architecture.pdf
[308] courses/csse2010/2026-09-07-introduction-to-embedded-c-programming.pdf
[309] courses/csse2010/fetch-decode-execute-cycle.pdf
[310] courses/csse2010/2026-10-15-compilation-and-linking.pdf
[311] courses/csse2010/cpu-organisation.pdf
[312] courses/csse2010/arithmetic-logic-unit.pdf

Stage	What happens
Fetch	Fetch an instruction from memory and bring it inside the CPU. Two special purpose registers involved — **PC** and **IR**
Decode	Resolve the operation (**opcode**) and operands (variables)
Execute	Control signals are issued and the operation takes place in the datapath

Stated as steps:

1. **Fetch** the next instruction from memory into the Instruction Register (**IR**).
2. Change the Program Counter (**PC**) to point to the next instruction.
3. Determine the type of (**decode**) the instruction just fetched — i.e. look at the opcode.
4. **Execute** the instruction.
5. Return to step 1.

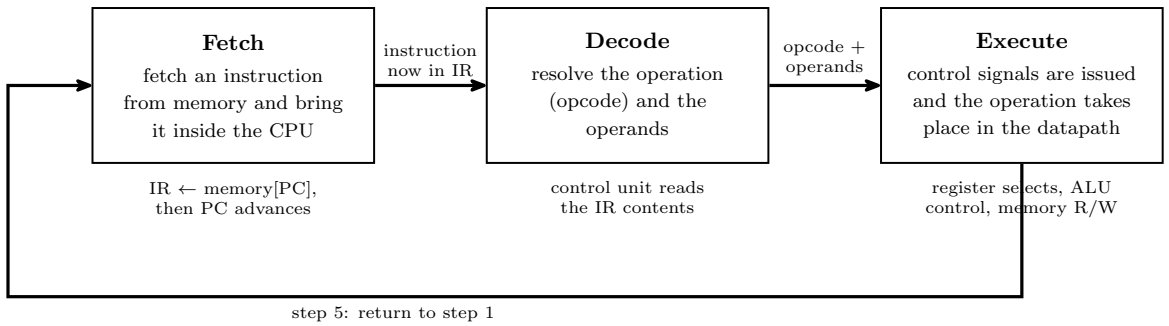


Figure 42: The cycle as a loop: a three-stage pipeline. Each box holds a one-line description of its stage, each arrow carries what the stage leaves behind for the next one, and the notes under the boxes summarise steps 1–4 above.

Step 2 advances the PC *before* the instruction executes, which is why a jump instruction works by overwriting the PC rather than by suppressing the increment.

The datapath

The question this raises — and the one the control unit exists to answer — is **how do we control this?** Specifically:

- How do we tell the ALU what to do?
- How do we choose what register values are placed on the A and B buses?

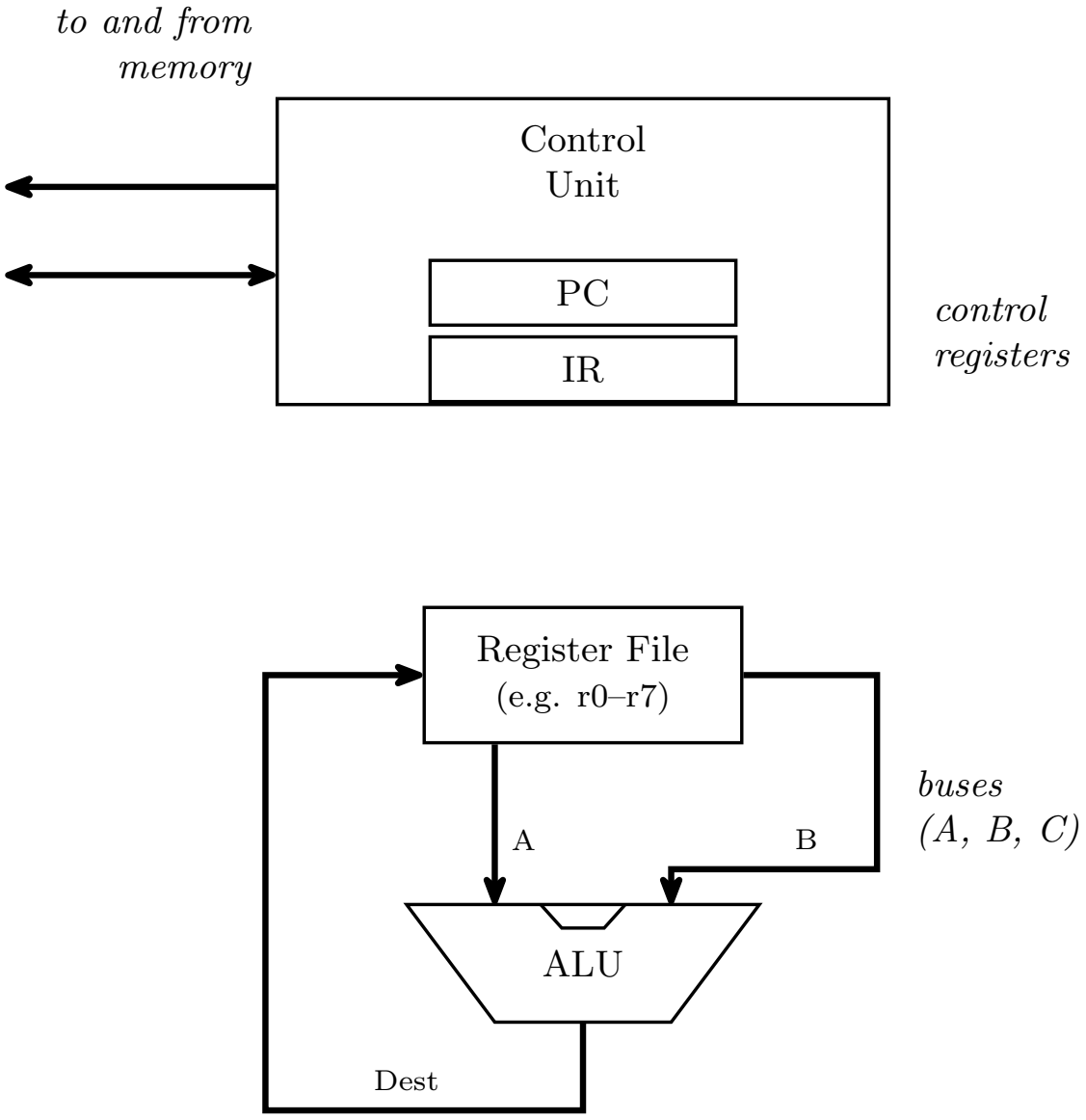


Figure 43: The datapath the cycle runs on. The control unit holds PC and IR and talks to memory; below it, the register file puts two values onto the A and B buses, the ALU combines them, and the result returns on the destination bus to be written back into a register.

- How do we choose which register receives the result from the destination bus?
- How do we know when to read or write memory?

Control signals

Each of those questions is answered by a bundle of wires out of the control unit. For a machine with 8 registers and the ALU of arithmetic-logic-unit[313]:

Signal	Width	Answers
A register select	3	which register drives the A bus ($2^3 = 8$ registers)
B register select	3	which register drives the B bus
Destination select	3	which register the result is written back into
ALU control	6	what the ALU does — F_1, F_0, ENA, INVA, ENB, INC
Memory read/write	1	whether this is a read or a write

Plus the address and data connections to memory itself. **The widths are not arbitrary**: three bits because there are eight registers, and six bits because the ALU has exactly six control inputs.

Control signals come from the control unit, which must generate them **in the right order** for a given instruction — and which instruction that is, is determined by the contents of the **Instruction Register**.

So a line of high-level code like `r5 = r0 + r2` becomes: A select = 0, B select = 2, destination select = 5, ALU control = the $A + B$ row of the combinations table, memory read/write idle.

Data influencing control

The IR is not enough. **The control unit can't operate without knowing something about the data** — for example, if a for-loop variable is supposed to count from 10 down to 0, how does the control unit know when it gets there?

The answer is the **status register**: determined by ALU operations, it stores the status of the **last** ALU operation and is available to the control unit. This closes the loop — the ALU's results feed back into the control unit and change what it does next, which is what makes

[313] courses/csse2010/arithmetic-logic-unit.pdf

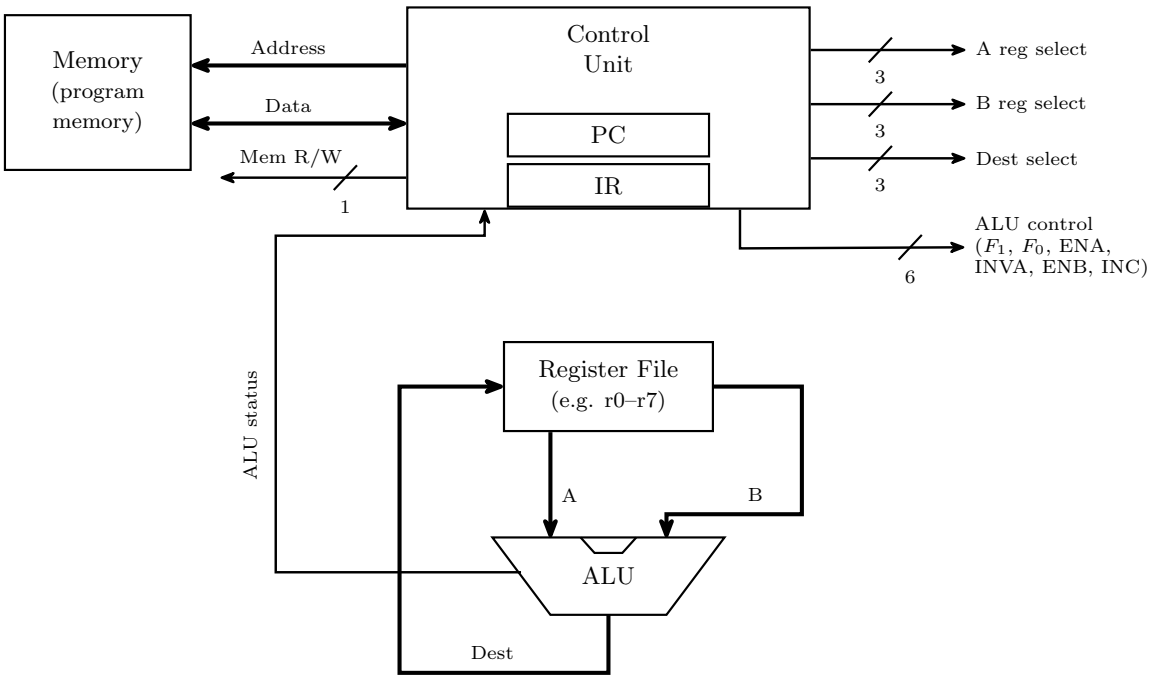


Figure 44: The same datapath with every control signal named. Added to the previous figure: the memory block, the *Address*, *Data* and *Mem R/W* labels, the five bundles with their widths, and the ALU status path back into the control unit. The bundles are drawn as labelled stubs: A/B/Dest select drive the register file, ALU control drives the ALU. Widths are marked with a slash and a number; the course also letters the width beside a plain arrow, and neither is fixed as the standard. Everything except the memory block sits inside the CPU.

conditional branching possible at all. It is the return path drawn in Figure 44, and it is what a source-level `if (x < 0)` compiles down to.

ALU status flags

The ALU emits, alongside its result, a summary of *what happened during the last operation.* Typically stored in a special purpose register called the **STATUS register**:

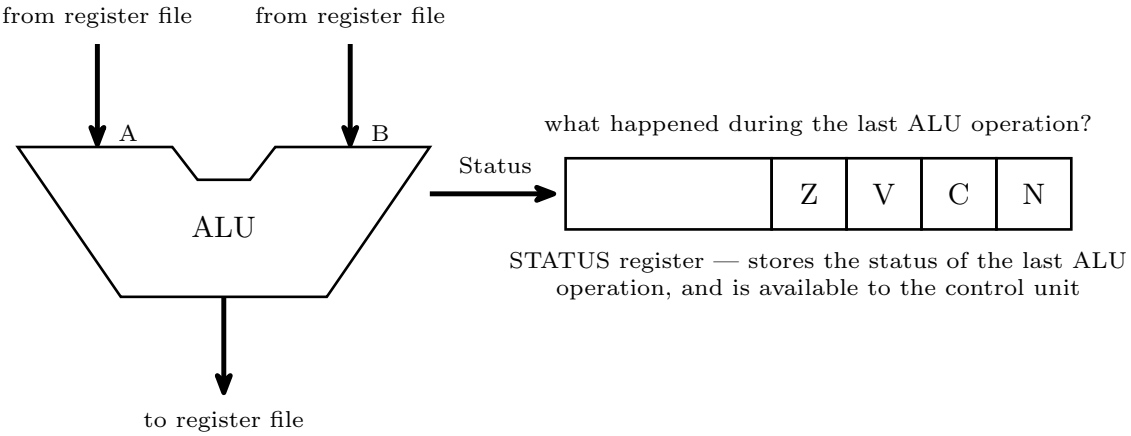


Figure 45: The ALU's second output. The result goes back to the register file; alongside it the ALU reports what happened, and those flags land in the STATUS register where the control unit can read them. The ALU is drawn as a notched trapezoid here; the block diagram in the ALU note draws the same block as a plain rectangle, and the course uses both shapes. The register is drawn as four named cells at the right-hand end of a wider register. That is shorthand, not a bit-position claim — the AVR's own SREG orders and numbers its bits differently, see the AVR architecture note.

Flag	Name	Asks
Z	zero	was the last result 0?
V	overflow	did the last addition/subtraction overflow, assuming it was a two's complement operation?
C	carry	was a carry out generated?
N	negative	was the result negative if considered as two's complement — i.e. what is the sign bit?

V and N both presuppose a two's complement interpretation while **C does not** — the hardware computes all four regardless, and it is the program's business to read the ones that mean something for the operation it just did. See binary-number-representations[314], and 2026-08-04-binary-arithmetic[315] for the overflow rules V implements.

Worked example

What values are in the status register flags after the operation $8_{10} - 9_{10}$ on a 5-bit ALU?

First, -9 in 5-bit two's complement: $9 = 01001$, invert to 10110, add one to get 10111. Then add, tracking the carry into and out of the sign column:

$$\begin{array}{rr} \text{carry into MSB} \rightarrow & 0 \\ & 01000 \\ + & 10111 \\ \hline \text{carry out} = 0 \rightarrow & 11111 \end{array}$$

The result is 11111, with no carry out of the top bit. So:

Flag	Value	Why
Z	0	the result 11111 is not zero
C	0	no carry out of the most significant bit
V	0	the carry into the sign column and the carry out of it agree, both 0 — and indeed $8 + (-9) = -1$ is comfortably in range for 5-bit two's complement
N	1	the sign bit is 1 — the result is negative

And indeed 11111 read as two's complement is -1, which is $8 - 9$.

[314] courses/csse2010/binary-number-representations.pdf
[315] courses/csse2010/2026-08-04-binary-arithmetic.pdf

See also

- cpu-organisation[316] · arithmetic-logic-unit[317]
- binary-number-representations[318] — two's complement, overflow and the sign bit
- 2026-08-04-binary-arithmetic[319] — the $C_{in} \oplus C_{out}$ overflow test the V flag implements
- avr-architecture[320] — the AVR's own status register, bit by bit
- state-machines[321] — the control unit is itself a state machine, sequencing signals for each instruction
- 2026-08-24-alu-and-control-unit[322] — Lecture 9

Floating-Point Numbers

How a computer holds numbers as large as 6×10^{23} and as small as 3.6×10^{-4} in the same few bytes: store the digits and the position of the point separately, as scientific notation in binary. This note covers why a fixed point is not enough, scientific notation and normalisation, where the word "mantissa" comes from, the IEEE 754 single and double precision formats (sign, excess-127 or excess-1023 exponent, mantissa with an implicit leading 1, and why the fields are in that order), converting in both directions, the special values (zero, subnormal numbers, infinity and NaN), what finite precision does to arithmetic, the family of formats from quarter to octuple precision with every quarter-precision value listed, and floating point in C and on the AVR. Fixed point, the alternative this improves on, is in fixed-point-numbers[323]. The excess notation behind the exponent is in binary-number-representations[324]. First met in 2026-10-19-floating-point-numbers[325].

Why the point has to float

A fixed-point format spends its bits once and for all between range and precision (fixed-point-numbers[326]). Real quantities do not cooperate.

- **Avogadro's number** is $6.02214076 \times 10^{23}$, written out `602214076000000000000000`. As a whole number it needs **79 bits** ($\log_2 \approx 78.99$), and only its first nine digits carry any information.

[316] courses/csse2010/cpu-organisation.pdf
[317] courses/csse2010/arithmetic-logic-unit.pdf
[318] courses/csse2010/binary-number-representations.pdf
[319] courses/csse2010/2026-08-04-binary-arithmetic.pdf
[320] courses/csse2010/avr-architecture.pdf
[321] courses/csse2010/state-machines.pdf
[322] courses/csse2010/2026-08-24-alu-and-control-unit.pdf
[323] courses/csse2010/fixed-point-numbers.pdf
[324] courses/csse2010/binary-number-representations.pdf
[325] courses/csse2010/2026-10-19-floating-point-numbers.pdf
[326] courses/csse2010/fixed-point-numbers.pdf

- **The volume of a cup** is about 0.00036 m^3. In fixed point its first 1 is in the 12th place after the point ($2^{-12} \approx 0.000244$), so it needs at least 12 fractional bits just to be non-zero, and more for any precision.

To hold both in one format, a fixed-point number would need about 79 integer bits *and* a few dozen fractional bits, nearly all of them zero in any given value. The way out is the one people already use: write the significant digits once and say separately **where the point goes**.

Scientific notation

Everyday language already lets the point move. **SI prefixes** do it in words: a kilometre is 10^3 m, a milligram 10^{-3} g, and the scale runs out to femto (10^{-15}), yocto (10^{-24}) and beyond. The number in front stays a convenient size and the prefix carries the scale.

Scientific notation does it in symbols:

$$\text{value} = \text{significant digits} \times \text{base}^{\text{exponent}}$$

so Avogadro's number is 6.02×10^{23} and the cup is 3.6×10^{-4} m^3. The digits say *what* the number is and the exponent says *how big*. Moving the point one place left and adding 1 to the exponent leaves the value unchanged, which is exactly the freedom fixed point lacks.

Normalisation

That freedom means one value has many spellings: $602 \times 10^{21} = 60.2 \times 10^{22} = 6.02 \times 10^{23} = 0.602 \times 10^{24}$. A **normalised** number picks one of them: **exactly one non-zero digit before the point**. In decimal that puts the digits in the range $[1, 10)$, and 6.02×10^{23} is the normalised form.

In binary the same rule gives the range $[1, 2)$**.** The only non-zero binary digit is 1, so a normalised binary number always looks like $1.xxx_2 \times 2^e$. Something like 3.985×2^{77} is not normalised in binary, even though 3.985 would be fine in front of a power of 10: it should be 1.993×2^{78}. Halving the digits and adding one to the exponent, or doubling and subtracting one, moves between the spellings.

Normalising gives every value a single representation, so no bit patterns are wasted on duplicates, and it guarantees no precision is spent on leading zeros. In binary it does one more thing, used below: the first digit is always 1, so it need not be stored.

Why "mantissa"

The digits part of a floating-point number is called the **mantissa** (the IEEE standard's own term is *significand*). The word comes from **logarithm tables**.

Before calculators, multiplication was done with $\log(a \times b) = \log a + \log b$: look up two logarithms, add them, and look up the antilog. A printed table only needs the logarithms of 1.00 to 9.99, because scaling by 10 only adds 1 to the logarithm:

x	$\log_{10} x$	Integer part	Fractional part
0.123	−0.9101	−1	0.0899
1.23	0.0899	0	0.0899
12.3	1.0899	1	0.0899
123	2.0899	2	0.0899

The fractional part, the same 0.0899 for all four, depends only on the significant digits 1, 2, 3, and it was called the **mantissa**. The integer part, which only says where the point goes, was the **characteristic**. A four-figure table lists the mantissas: the row for 12 and the column for 3 give `0899`, meaning $\log_{10} 1.23 = 0.0899$. (For 0.123 the log is $0.0899 - 1 = -0.9101$; tables wrote it $\bar{1}.0899$ to keep the mantissa visible.)

A floating-point number splits the same way: digits that say *what*, an exponent that says *where the point goes*. The digits inherited the name, although strictly the old mantissa was a logarithm and the new one is the digits themselves.

IEEE 754

Almost every computer stores floating point in the formats of the **IEEE 754** standard. A number is stored as three fields, **sign**, **exponent** and **mantissa**, and stands for

$$(-1)^S \times 1.M_2 \times 2^{E-\text{bias}}$$

Format	Bits	Sign	Exponent	Mantissa	Bias
Single precision	32	1	8	23	127
Double precision	64	1	11	52	1023

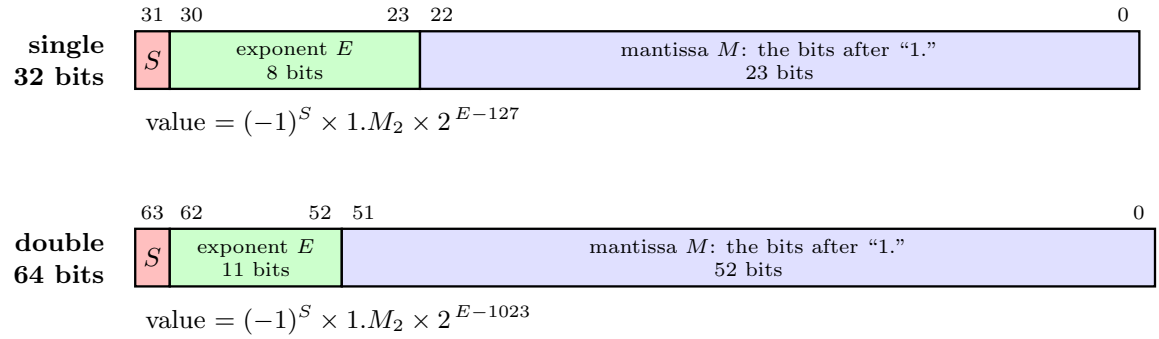


Figure 46: The IEEE 754 single and double precision layouts, MSB on the left, with the bit number at each field boundary. The sign bit is 0 for plus and 1 for minus. The exponent field E holds the true exponent plus the bias, 127 or 1023. The mantissa field M holds only the bits after the point, because a normalised binary mantissa always starts with a 1 that does not need storing. Reading a pattern means putting those three back together as (-1) to the S, times 1.M, times 2 to the E minus the bias. The exponent and mantissa are drawn to scale within each row, double precision at half the scale of single; the sign bit is drawn wider than one bit so it can be labelled.

The biased exponent

The exponent can be negative, so it needs a signed format. IEEE uses **excess** (biased) notation: the field holds the true exponent plus a fixed **bias**,

$$E = e + \text{bias}, \qquad \text{bias} = 2^{k-1} - 1 \text{ for a } k\text{-bit exponent.}$$

So single precision is **excess-127** and double precision **excess-1023**. A true exponent of 4 is stored as $4 + 127 = 131 = 10000011_2$; a true exponent of -2 as $125 = 01111101_2$.

This is the excess notation of binary-number-representations[327], with one difference to watch: the bias there is 2^{k-1} (excess-128 for 8 bits), while IEEE uses one less, 127. The stored values 0 and 255 are kept for the special values, so the normal true exponents run from $1 - 127 = -126$ to $254 - 127 = +127$.

The implicit leading 1

A normalised binary mantissa is always $1.xxxx_2$ (normalisation). A bit that is always 1 carries no information, so **IEEE does not store it**. The mantissa field holds only the bits after the point, and the 1 is put back when the number is read. Single precision therefore has **24**

[327] courses/csse2010/binary-number-representations.pdf

significant bits for the price of 23. The one exception is the subnormal numbers, where the hidden bit is 0.

Why sign, exponent, mantissa

The order is chosen so that **floating-point numbers sort like integers**. Compare 1×2^{100} with 3×2^{20}: the first is far larger, even though its mantissa is smaller, because the exponent decides the scale and the mantissa only refines it. Putting the exponent in the high bits, above the mantissa, means that comparing two patterns as plain integers compares exponents first and mantissas only when the exponents are equal, which is exactly numeric order.

The biased exponent is what makes this work. In excess form a larger exponent is always a larger unsigned field, negative exponents included, so no two's complement sign gets in the way. The result: for two non-negative floats, the one whose bit pattern is the larger unsigned integer is the larger number, and the integer comparison hardware can sort floats. The quarter-precision table shows it: reading the patterns in counting order reads the values in increasing order. The sign bit comes first so that sign alone splits positive from negative; negative numbers are sign and magnitude, so among them the order reverses.

Encoding: decimal to IEEE

1. **Convert to binary scientific notation**, normalised: $\pm 1.xxxx_2 \times 2^e$ (fixed-point-numbers[328] for the fraction bits).
2. **Store the sign**: $+$ gives 0, $-$ gives 1.
3. **Store the biased exponent**: $e + 127$ in 8 bits (single) or $e + 1023$ in 11 bits (double).
4. **Store the mantissa**: drop the leading 1 and **pad with zeros on the right** to 23 (or 52) bits. Right, because these are fraction bits: zeros after the last 1 do not change the value.
5. **Group into nibbles** and write the 32 bits as 8 hex digits.

And **beware of special values**: zero, infinity and NaN do not follow these steps (below).

-23.25 worked through these steps to `0xC1BA0000` is in 2026-10-19-floating-point-numbers[329].

Worked: 5.75 to single precision

> Write 5.75 as an IEEE single-precision number in hexadecimal.

1. $5.75 = 101.11_2 = 1.0111_2 \times 2^2$.
2. Positive: $S = 0$.
3. $E = 2 + 127 = 129 = 10000001_2$.

[328] courses/csse2010/fixed-point-numbers.pdf
[329] courses/csse2010/2026-10-19-floating-point-numbers.pdf

4. M = `0111` followed by 19 zeros.
5. `0 10000001 01110000000000000000000` = `0100 0000 1011 1000 0000 0000 0000 0000` = `0x40B80000`.

Worked: -0.375 to single precision

Write -0.375 as an IEEE single-precision number in hexadecimal.

1. $0.375 = 0.011_2 = 1.1_2 \times 2^{-2}$. The point moves *right*, so the exponent is negative.
2. Negative: $S = 1$.
3. $E = -2 + 127 = 125 = 01111101_2$.
4. M = `1` followed by 22 zeros.
5. `1 01111101 10000000000000000000000` = `1011 1110 1100 0000 0000 …` = `0xBEC00000`.

Worked: 0.1 to single precision

Write 0.1 as an IEEE single-precision number. Is it exact?

$0.1 = 0.0\,0011\,0011\,0011\ldots_2$ repeating (fixed-point-numbers[330]), which normalises to $1.1001\,1001\,1001\ldots_2 \times 2^{-4}$. So $S = 0$ and $E = -4 + 127 = 123 = 01111011_2$. The mantissa should be `1001` repeated for ever. Only 23 bits fit:

`10011001100110011001100` | `1100…` (the bits that do not fit)

What is cut off starts with a 1 followed by more 1s, so it is more than half a unit in the last place, and the stored mantissa **rounds up** to `10011001100110011001101`. The pattern is `0 01111011 10011001100110011001101` = `0x3DCCCCCD`, and it stands for 0.100000001490116 ..., not 0.1. **0.1 has no exact binary floating-point form**, for the same reason $\frac{1}{3}$ has no exact decimal one.

Decoding: IEEE to decimal

1. Write the hex as 32 bits and **split 1 / 8 / 23**.
2. **Sign**: 0 is +, 1 is −.
3. **Exponent**: check for all 0s or all 1s (special values); otherwise $e = E - 127$.
4. **Mantissa**: put back the hidden 1, giving $1.M_2$.
5. Shift the point e places (right if $e > 0$) and convert.

`0x422F0000` decoded this way to 43.75 is in 2026-10-19-floating-point-numbers[331].

[330] courses/csse2010/fixed-point-numbers.pdf
[331] courses/csse2010/2026-10-19-floating-point-numbers.pdf

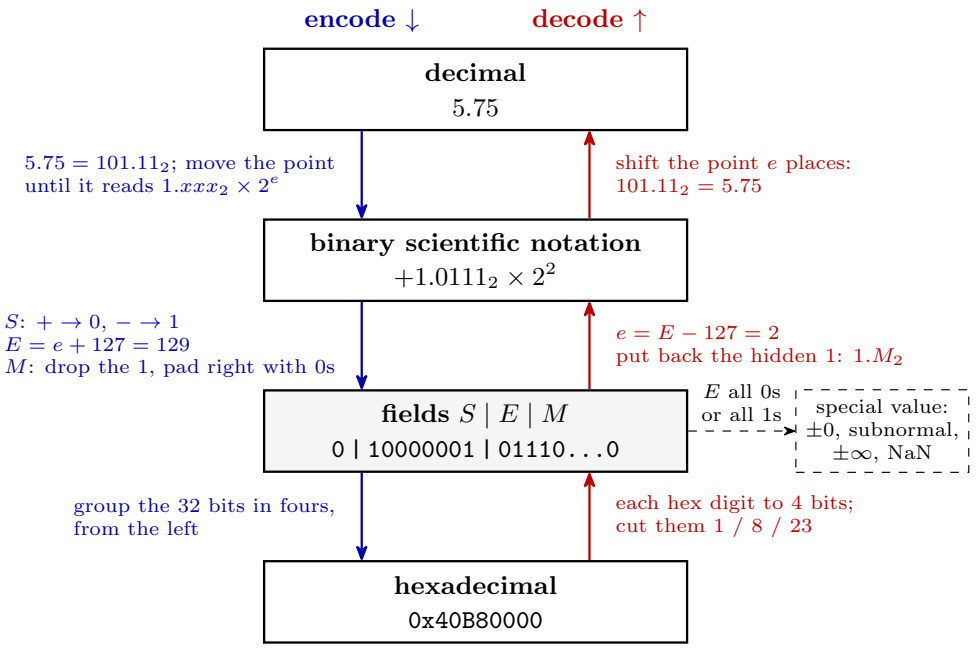


Figure 47: Encoding and decoding are the same four stages walked in opposite directions, shown with 5.75. Encoding (down the left) writes the value in binary and moves the point until one 1 is left in front of it; the sign becomes S, the power of 2 plus 127 becomes E, and the bits after the leading 1 become M, padded with zeros on the right; the 32 bits are then grouped in fours for hex. Decoding (up the right) undoes each step: expand every hex digit to four bits, cut the bits 1 / 8 / 23, take 127 off E and put the hidden 1 back, and shift the point. The one extra step is on the way up: an exponent field of all 0s or all 1s is not a normal number and has to be read as a special value instead.

Worked: `0x41200000`

What decimal value does the single-precision pattern `0x41200000` hold?

`0100 0001 0010 0000 …` splits as $S = 0$, $E = 10000010_2 = 130$, $M = 0100\ldots$. So the value is $+1.01_2 \times 2^{130-127} = 1.01_2 \times 2^3 = 1010_2 = \mathbf{10}$.

Worked: two special patterns

What do `0x80000000` and `0x7F800000` hold?

- `0x80000000` is `1 00000000 000…0`: exponent all 0s and mantissa 0, with the sign set. It is -0, which compares equal to $+0$.
- `0x7F800000` is `0 11111111 000…0`: exponent all 1s and mantissa 0, sign clear. It is $+\infty$. Following steps 4 and 5 blindly would give 1.0×2^{128}, which is the mistake the special-value check is there to catch.

Special values

The smallest and largest exponent fields are reserved. What a pattern means then depends on whether the mantissa field is zero. The exponent here is the raw field as stored, with no bias taken off, so the reserved values are 0 (all 0s) and the maximum (all 1s, 255 in single precision):

Exponent field (as stored)	Mantissa = 0	Mantissa ≠ 0
0 → all 0s	±0	subnormal numbers
anything else	normal numbers	normal numbers
maximum → all 1s (255 for single, 2047 for double)	± infinity	± NaN (not a number)

This raw field is often called the *unbiased* exponent, in the sense that the bias has not yet been subtracted; the true exponent of a normal number is the field minus 127 (or 1023).

Zero

Zero cannot be normalised (it has no leading 1), so it gets its own pattern: **every exponent and mantissa bit 0**. Because the sign bit is separate there are two of them, $+0$ (`0x00000000`) and -0 (`0x80000000`). They compare equal; the sign survives only in things like $1/-0 = -\infty$.

stored exponent	**mantissa** $= 0$	**mantissa** $\neq 0$
all 0s 00000000	± 0	subnormal $\pm 0.M_2 \times 2^{-126}$
anything else 1 to 254	normal: $\pm 1.M_2 \times 2^{E-127}$	
all 1s 11111111	$\pm\infty$	$\pm$ NaN

single precision shown; double uses 11-bit fields, 2047 and 2^{E-1023}, 2^{-1022}

Figure 48: What a floating-point pattern means, decided by the stored exponent field (rows) and whether the mantissa field is zero (columns). Every exponent except all 0s and all 1s gives an ordinary normalised number, with the hidden 1 in front of the mantissa. All 0s gives zero, or a subnormal number with a hidden 0 instead of a 1 and the smallest normal exponent, which is 2 to the minus 126 in single precision. All 1s gives infinity, or NaN when any mantissa bit is set. The sign bit applies in every cell, so there are two zeros, two infinities, and NaNs of either sign.

Subnormal numbers

With an all-0s exponent and a non-zero mantissa, the number is **subnormal** (also called denormal):

$$(-1)^S \times 0.M_2 \times 2^{-126} \quad \text{(single precision)}$$

Two rules differ from normal numbers.

- **The hidden bit is 0, not 1.** This breaks the scientific-notation rule of one non-zero digit before the point, on purpose: it lets values smaller than the smallest normal number be held, with fewer significant bits as they shrink.
- **The exponent is 2^{-126}, the same as the smallest normal numbers**, not $2^{0-127} = 2^{-127}$ as the excess rule would give. With 2^{-126} the subnormals continue the smallest normal binade at exactly the same spacing, 2^{-149}, all the way down to 0, with no gap and no overlap. With 2^{-127} they would be spaced twice as finely and leave a gap below 2^{-126}.

The smallest normal single is $2^{-126} \approx 1.18 \times 10^{-38}$, and the smallest subnormal is $2^{-23} \times 2^{-126} = 2^{-149} \approx 1.4 \times 10^{-45}$. The effect is easiest to see in quarter precision (figure below).

Infinity

All-1s exponent, zero mantissa: $\pm\infty$. It is the result when

- a number is **too big to store**: anything past the largest finite value (about 3.4×10^{38} for single) **overflows** to ∞;
- a calculation is mathematically infinite: $1 \div 0 = +\infty$, $-\log 0 = +\infty$ ($\log 0 = -\infty$), $\tan(\pi/2)$.

On the last: in practice `tan(M_PI/2)` returns a large finite number, about 1.6×10^{16} in double, because $\pi/2$ itself cannot be stored exactly and the tangent of the stored value is merely huge.

Infinity behaves sensibly in further arithmetic: $\infty + 1 = \infty$, $1/\infty = 0$, $-\infty < \$$ every finite number.

NaN

All-1s exponent, non-zero mantissa: **NaN, not a number**. It is the result of an operation with **no meaningful answer**:

- $0 \div 0$, $0 \times \infty$, $\infty - \infty$;
- $\sqrt{-1}$, $\log(-1)$, $\sin^{-1} 20$ (no real angle has a sine of 20);
- 0^0, as an indeterminate form. IEEE 754's `powr(0, 0)` gives NaN; C's `pow(0, 0)` returns 1 by convention.

A NaN propagates: any arithmetic with a NaN operand gives NaN, so one bad step poisons everything computed from it, and the NaN at the end shows something went wrong. There are a few exceptions, functions whose answer does not depend on the missing value: `pow(NaN, 0)` is 1, and `fmax(NaN, 3)` is 3. Comparisons do not produce NaN at all; they return false. In particular **`x == x` is false when `x` is NaN**, which is the standard test for one (or `isnan(x)` in C). With 23 mantissa bits there are $2^{23} - 1$ different NaN patterns per sign, all meaning NaN.

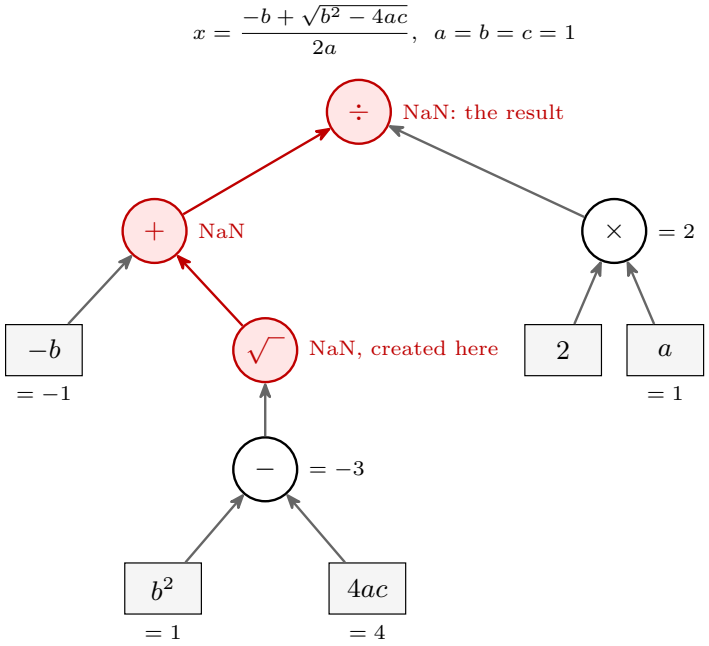


Figure 49: How one NaN poisons a calculation, in the quadratic formula with a = b = c = 1, which has no real root. Arrows show which results feed which operation. The discriminant b squared minus 4ac is -3, an ordinary number. Its square root has no real answer, so that operation creates a NaN. Every operation that then takes the NaN as an operand gives NaN too, so the sum and the final division are NaN. The branch that never touches it, 2a, is still 2. The NaN that comes out at the top is the signal that something went wrong further down. Comparisons are the exception: they do not give NaN but false, so result == result is false for a NaN.

Finite precision

A float has a fixed number of mantissa bits, so most real numbers **cannot be represented exactly** and are **rounded** to the nearest value that can (ties go to the neighbour whose

last mantissa bit is 0, "round half to even"). The error is relative: single precision keeps 24 significant bits, about 7 decimal digits, wherever the point is; double keeps 53, about 16.

Rounding happens after **every** operation, and that changes the arithmetic rules.

Division and multiplication need not undo each other. $10 \div 3 = 3.333\ldots = 11.010101\ldots_2$, which repeats for ever and has to be cut short. In 3-digit decimal the same thing gives $10 \div 3 = 3.33$ and $3.33 \times 3 = 9.99 \neq 10$. Whether the binary version lands back on 10 depends on how the roundings fall: for $10 \div 3 \times 3$ they happen to cancel in single and double, but $(1 \div 41) \times 41 = 0.99999994$ in single, and $(1 \div 49) \times 49 = 0.9999999999999999$ in double. So $(x \div y) \times y$ **may not equal** x, and code should never test floats for exact equality after arithmetic; it should test whether they are within a tolerance.

The order of operations matters. This is often put loosely as floating point being *occasionally non-commutative*. Strictly, addition is still commutative ($a + b = b + a$ always); what fails is **associativity**: $(A + B) + C$ **need not equal** $A + (B + C)$, because the two orders round different intermediate results. In double, $(0.1 + 0.2) + 0.3 = 0.6000000000000001$ while $0.1 + (0.2 + 0.3) = 0.6$. The order of a long sum changes its answer.

Absorption

If $X \gg \delta$, then $(X + \delta) - X$ can be 0. Near X the representable values are spaced by about $X \times 2^{-23}$ (single). If δ is less than half that gap (or exactly half, when the tie goes to X), $X + \delta$ rounds straight back to X, and subtracting X leaves nothing: δ has been **absorbed**.

- Single precision: $2^{24} = 16\,777\,216$, and the gap there is 2. So $2^{24} + 1$ is a tie between $16\,777\,216$ and $16\,777\,218$, rounds to the even one, and $(2^{24} + 1) - 2^{24} = 0$. The same happens with $10^8 + 1 - 10^8$.
- It also breaks associativity: $(2^{24} + 1) + 1 = 2^{24}$, since each 1 is absorbed in turn, but $2^{24} + (1 + 1) = 2^{24} + 2$.

So adding many small numbers to a large total can lose them all; adding the small ones together first keeps them.

The formats compared

IEEE 754 defines a family of binary formats that differ only in field widths. Each follows the same rules: bias $2^{k-1} - 1$ for a k-bit exponent, hidden 1, the same special values. With k exponent bits and m mantissa bits:

- the **next value after 1** is $1 + 2^{-m}$, the precision near 1;
- the **largest finite value** is $(2 - 2^{-m}) \times 2^{\text{bias}}$: all-1s mantissa, the exponent field one below all 1s.

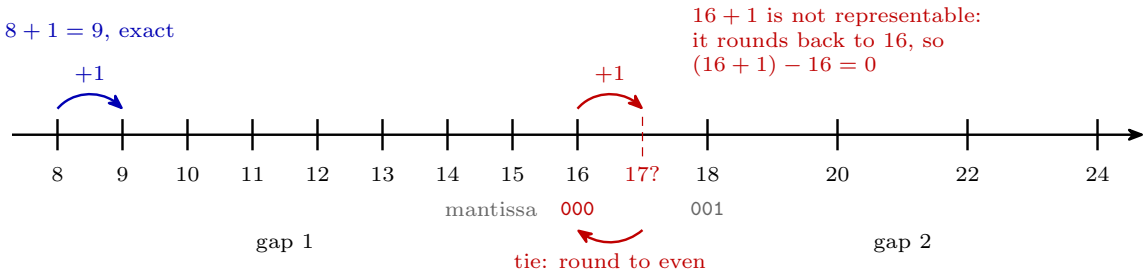


Figure 50: Absorption, shown on the quarter-precision values between 8 and 24. Between 8 and 16 neighbouring values are 1 apart, so $8 + 1 = 9$ is exact. Between 16 and 32 they are 2 apart, and $16 + 1 = 17$ falls exactly halfway between 16 and 18. The tie goes to the neighbour whose last mantissa bit is 0: 16 has mantissa field 000 and 18 has 001, so $16 + 1$ rounds back to 16 and (16 + 1) - 16 = 0. The same thing happens in any format whenever the number added is less than half the gap at the size of the larger number, or exactly half with the tie going back down.

	Quarter	Half	Single	Double	Quadruple	Octuple
Total bits	8	16	32	64	128	256
Sign bits	1	1	1	1	1	1
Exponent bits	4	5	8	11	15	19
Mantissa bits	3	10	23	52	112	236
Bias	7	15	127	1023	16383	262143
Next after 1	$1 + \frac{1}{8} = 1.125$	$1 + \frac{1}{1024} \approx 1.001$	$1 + 2^{-23} \approx 1 + 1.2 \times 10^{-7}$	$1 + 2^{-52} \approx 1 + 2.2 \times 10^{-16}$	$1 + 2^{-112} \approx 1 + 1.9 \times 10^{-34}$	$1 + 2^{-236} \approx 1 + 9.1 \times 10^{-72}$
Maximum	240	65504	$\approx 3.4 \times 10^{38}$	$\approx 1.8 \times 10^{308}$	$\approx 1.2 \times 10^{4932}$	$\approx 1.6 \times 10^{78913}$
Use	ultra rare	rare	common	common	rare	ultra rare

Quarter precision is not one of the standard's formats. It is an 8-bit format built by the same rules (1 sign, 4 exponent, 3 mantissa bits, bias 7), small enough to list every value (below). Half precision appears in graphics and machine learning, where memory matters more than digits; single and double are what ordinary code uses.

The maximum grows with the **exponent** width and the precision with the **mantissa** width. Each extra exponent bit doubles the bias and so roughly squares the maximum: $2^8 \to 2^{16} \to 2^{128} \to 2^{1024}$ as the exponent goes 4, 5, 8, 11 bits. The quarter maximum is $1.875 \times 2^7 = 240$ and the half maximum $(2 - 2^{-10}) \times 2^{15} = 65504$.

Every quarter-precision value

Quarter precision has only 256 patterns, so every value can be listed. Each row is a sign bit (0 here) and a 4-bit exponent field; each column is the 3-bit mantissa field. A normal row with stored exponent E holds $1.M_2 \times 2^{E-7}$. The first row (in italics) is the subnormals, $0.M_2 \times 2^{-6}$, and the last row is infinity and NaN. The negative values are the same table with the sign bit set.

Sign, exponent	...000	...001	...010	...011	...100	...101	...110	...111

Sign, exponent	…000	…001	…010	…011	…100	…101	…110	…111
0 0110	0.5	0.5625	0.625	0.6875	0.75	0.8125	0.875	0.9375
0 0111	1	1.125	1.25	1.375	1.5	1.625	1.75	1.875
0 1000	2	2.25	2.5	2.75	3	3.25	3.5	3.75
0 1001	4	4.5	5	5.5	6	6.5	7	7.5
0 1010	8	9	10	11	12	13	14	15
0 1011	16	18	20	22	24	26	28	30
0 1100	32	36	40	44	48	52	56	60
0 1101	64	72	80	88	96	104	112	120
0 1110	128	144	160	176	192	208	224	240
0 1111	Inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN

What the table shows:

- **Each row is a binade**, the values from one power of 2 up to the next, and it holds **eight equally spaced values**. The spacing doubles from each row to the next: $\frac{1}{8}$ between 1 and 2, $\frac{1}{4}$ between 2 and 4, and 16 between 128 and 240. The relative precision is the same in every row, which is the whole point of floating point.
- **Exponent field 0111 is the row of 1**, because 7 is the bias: $2^{7-7} = 1$.
- **Counting up through the patterns counts up through the values**, row by row (why this order).
- **The subnormal row has the same spacing as the row after it**, $\frac{1}{512}$, and reaches down to 0. That is the effect of giving subnormals the exponent 2^{-6} rather than 2^{-7}.
- **The largest finite value is 240**, and the next pattern up is infinity. Anything at or above 248, halfway from 240 to the next step, overflows.
- **Not every whole number is here.** 17, 19 and 33 are missing; between 16 and 32 only the even numbers can be held.

Worked: rounding in quarter precision

In quarter precision, what are $16 + 1$, $13 \div 3$, and then that quotient times 3?

- $16 + 1$**.** Between 16 and 32 the values are 2 apart. 17 is exactly halfway between 16 (mantissa 000) and 18 (mantissa 001); the tie goes to the even mantissa, **16**. The 1 has been absorbed.
- $13 \div 3 = 4.333\ldots$ Between 4 and 8 the values are 0.5 apart, and 4.333 is nearer 4.5 than 4. So the quotient is **4.5**.
- $4.5 \times 3 = 13.5$**.** Between 8 and 16 the values are 1 apart, and 13.5 is a tie between 13 (101) and 14 (110); it goes to the even mantissa, **14**. So $(13 \div 3) \times 3 = 14$, not 13.

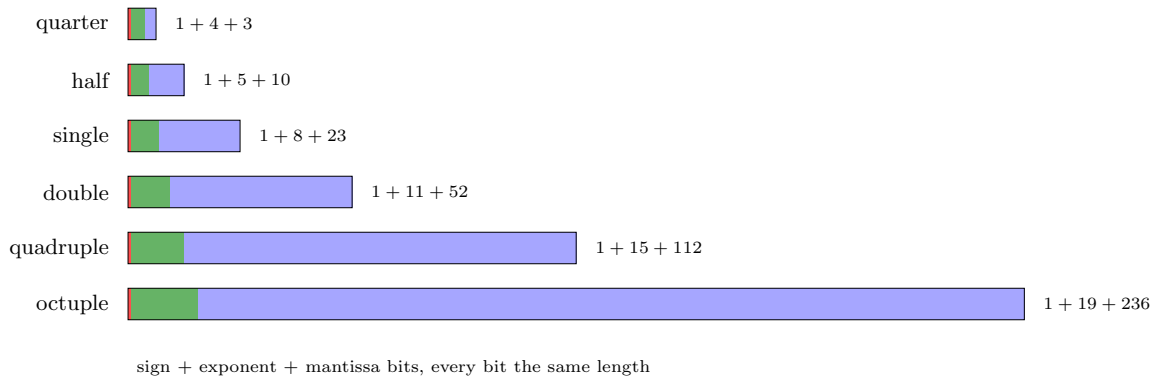


Figure 51: The six formats drawn to one scale, one bit per unit of length, with the sign in red, the exponent in green and the mantissa in blue. As the formats double in size, nearly all of the new bits go to the mantissa: the exponent only grows from 4 to 19 bits while the mantissa grows from 3 to 236. A few exponent bits already give an enormous range, because each one doubles the largest power of 2; the precision needs many more bits, because each mantissa bit only halves the gap between neighbouring values.

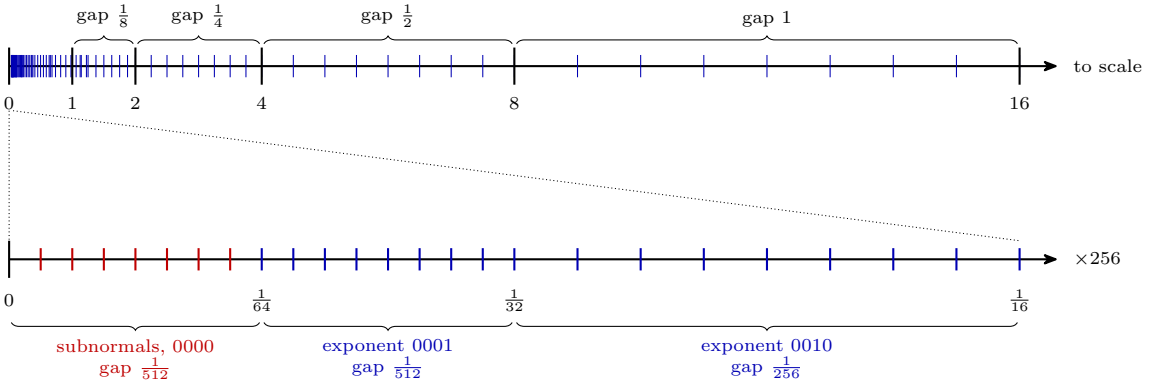


Figure 52: The positive quarter-precision values on a number line. Top: 0 to 16 to scale. Every binade from one power of 2 to the next holds eight evenly spaced values, so the gap doubles at each power of 2, from 1/8 between 1 and 2 to 1 between 8 and 16, and the values crowd together towards 0. Bottom: the stretch from 0 to 1/16 magnified 256 times, so that 1/16 lands directly under 16. The subnormals (exponent field 0000) fill the space below the smallest normal value, 1/64, with the same gap of 1/512 as the first normal binade, so the spacing carries on evenly down to 0 instead of leaving a hole.

Floating point in C and on the AVR

C has two floating-point types, `float` and `double`. They are **usually** IEEE single and double precision, but the C standard only sets minimums. Simplified:

- `float` must represent numbers up to 10^{37} while keeping **6 decimal digits** of accuracy;
- `double` must represent numbers up to 10^{37} while keeping **10 decimal digits**, and must not be less accurate than `float`.

Single precision meets the first easily (about 7 digits, up to 3.4×10^{38}), and double the second (about 16 digits).

On the AVR, avr-gcc makes `double` the same as `float`: 32-bit single precision. That is strictly against the standard, since single precision has about 7 digits, not 10. A program that moves to the AVR from a desktop and relies on `double` silently loses half its digits (c-language-basics[332]).

GCC also offers `_Float16` and `__float128`, for half and quadruple precision, on targets that support them. There is no C type for quarter or octuple precision.

Prefer integer arithmetic

The AVR has **no floating-point hardware**. Its only data type in hardware is the 8-bit integer (instruction-set-architecture[333]), so every `float` operation is a call to a library routine that unpacks the fields, aligns, adds or multiplies the mantissas with integer instructions, renormalises, rounds and checks for special values. Compare two lines that give the same result for every input:

```
PORTB = PINA / 3;        /* integer division:  about 8 instructions, no branches */
PORTB = PINA * 0.334f;   /* floating point:    300+ instructions, with branches   */
```

The second converts the byte to a `float`, calls the software multiply, and converts the result back, and each of those steps is a routine full of tests and loops. Its running time also varies with the input, which matters in an interrupt handler or a timing loop. The first needs no library at all: a compiler turns division by a constant into a multiplication by a fixed-point reciprocal and a shift. One way for a byte: $x \div 3 = (x \times 171) \gg 9$, since $171/512 = 0.33398\ldots$ is just above $\frac{1}{3}$, and this gives exactly $\lfloor x/3 \rfloor$ for every x from 0 to 255. (Checking all 256 inputs also confirms that truncating `PINA * 0.334f` gives the same answers.)

So on a small microcontroller, **work in integers or fixed-point-numbers[334] wherever the result can be the same**: scale a value so that the fraction becomes whole numbers (millivolts instead of volts, tenths of a degree), and divide or shift at the end.

[332] courses/csse2010/c-language-basics.pdf
[333] courses/csse2010/instruction-set-architecture.pdf
[334] courses/csse2010/fixed-point-numbers.pdf

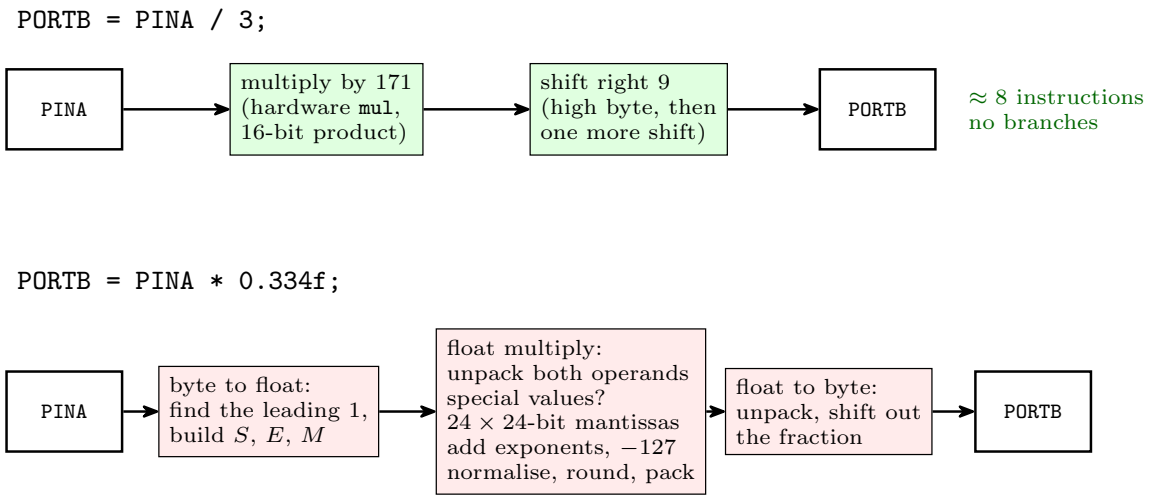


Figure 53: Why the two lines differ so much in cost on the AVR. Dividing a byte by 3 can be done with the hardware 8-by-8 multiply and a shift: multiply by 171 and shift right 9 places, which is the same as taking the high byte of the product and shifting it once more. That is a few instructions with no branches, taking the same time for every input. The float version has no hardware to use. The byte is first converted to a float, which means finding its leading 1 and building the sign, exponent and mantissa fields. The multiply is a library routine that unpacks both operands, tests for special values, multiplies the 24-bit mantissas with 8-bit multiplies, adds the exponents and takes off one bias, then normalises, rounds and packs the result. A third routine converts back to an integer. Each step is a loop or a chain of tests, so the time also depends on the value.

Common mistakes

- **Normalising binary to $[1, 10)$.** In binary the mantissa must be in $[1, 2)$: 3.985×2^{77} is not normalised; 1.993×2^{78} is.
- **Storing the hidden 1.** The mantissa field holds only the bits *after* the point.
- **Using a bias of 128.** IEEE single is excess-127 and double excess-1023, one less than the 2^{k-1} of plain excess notation.
- **Forgetting the bias when decoding.** An exponent field of 132 means 2^5, not 2^{132}.
- **Padding the mantissa on the left.** The zeros go on the right; they are fraction bits.
- **Decoding special patterns as normal numbers.** Check the exponent for all 0s or all 1s first: `0x7F800000` is infinity, not 2^{128}.
- **Treating a subnormal's exponent as 2^{-127}.** It is 2^{-126}, with a hidden 0.
- **Testing floats for equality.** After any arithmetic, compare with a tolerance; `0.1` is not exactly 0.1 to begin with.
- **Assuming addition is associative.** The order of a sum can change its result, and a small term can vanish into a large one.
- **Expecting 64 bits from `double` on the AVR.** It is 32.
- **Using floats on the AVR where integers would do.** It costs hundreds of instructions per operation.

See also

- fixed-point-numbers[335] — the fixed binary point, $m.n$ formats, and the range/precision trade-off this escapes
- binary-number-representations[336] — excess notation, sign and magnitude, hex grouping
- c-language-basics[337] — `float` and `double` among C's types, and floating-point constants
- instruction-set-architecture[338] — the AVR's hardware data types, which do not include floating point
- 2026-10-19-floating-point-numbers[339] — Lecture 23

Instruction Set Architecture

The layer where software meets hardware: what a machine instruction is, what an ISA specifies, and why assembly language exists. See cpu-organisation[340] for the hardware below this line and avr-instructions[341] for a real instruction set.

[335] courses/csse2010/fixed-point-numbers.pdf
[336] courses/csse2010/binary-number-representations.pdf
[337] courses/csse2010/c-language-basics.pdf
[338] courses/csse2010/instruction-set-architecture.pdf
[339] courses/csse2010/2026-10-19-floating-point-numbers.pdf
[340] courses/csse2010/cpu-organisation.pdf
[341] courses/csse2010/avr-instructions.pdf

Machine instructions

A machine instruction is a pattern of bits. `0000 1101 0111 0111` is a valid Atmel AVR instruction. Instructions typically consist of two things:

- **Opcode** (operation code) — defines the operation, e.g. addition.
- **Operands** — what is being operated on, e.g. particular registers or a memory address.

So one word of machine code says *what to do* and *what to do it to*, packed into fixed bit fields. Which bits mean which is entirely a matter of convention — and that convention is the ISA.

The ISA

> An **Instruction Set Architecture** is a **specification** — i.e. a document. It specifies the **interface between hardware and software**.

The point of drawing it as a line is that **the ISA is the only thing the two sides have to agree on**. A compiler writer needs the ISA and nothing else about the chip; a chip designer needs the ISA and nothing else about the programs that will run.

Microarchitecture

> **Microarchitecture** = the architecture of the **control unit** — i.e. the state machine (gates and flip-flops etc.) that implements the control unit.

This is the same state-machines[342] material from earlier in the course, applied at the scale of a whole processor: the control unit is a state machine whose job is to emit the right control signals, in the right order, for whatever the instruction register holds.

ISA vs microarchitecture

One ISA can be implemented by many different microarchitectures. The specification says what the bit patterns mean; it does not say how a chip must go about honouring them.

ISA	Implemented by
8086	many processors, in different ways
Pentium	Pentium … Pentium IV … Core2Duo … i7 (each in different ways), various AMD devices, and other manufacturers

[342] courses/csse2010/state-machines.pdf

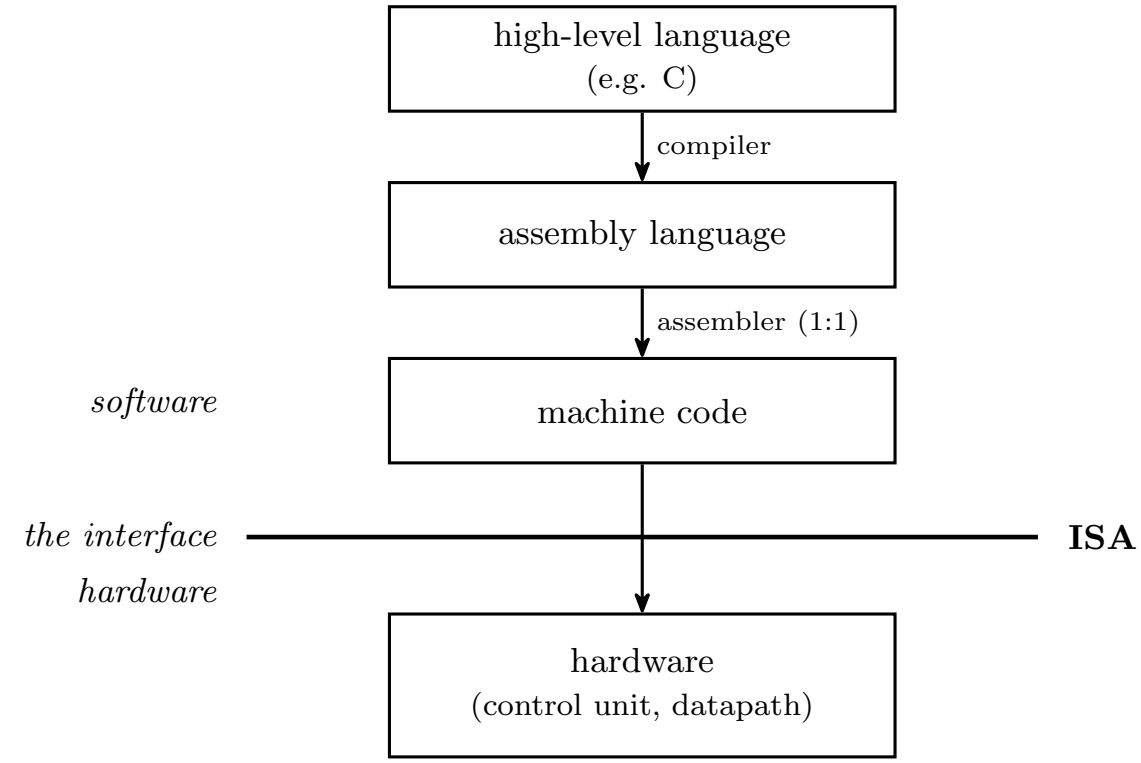


Figure 54: The ISA as an interface. Software above the line is written and compiled without knowing how the hardware works; hardware below the line is designed without knowing what program will run. The only thing both sides must agree on is which bit patterns mean what — and that agreement is a document, not a circuit.

This is why a program compiled decades ago still runs: the ISA held still while the microarchitecture beneath it was rebuilt many times over.

What makes an ISA?

Four things:

1. **Memory models**
2. **Registers**
3. **Instructions**
4. **Data types**

   If you know all these details, you can **write machine code that runs on the CPU** *and* **build the CPU**.

This is the completeness test for a specification: both sides of the line in Figure 54 have to be constructible from the document alone, and nothing else about the chip may be needed by either.

1. Memory models

How does memory look to the CPU? Four issues decide it — **addressable cell size**, **alignment**, **address spaces** and **endianness** — treated in full in memory-models[343].

2. Registers

Two types of register:

- **General purpose** — used for temporary results etc. On AVR, r0–r31.
- **Special purpose**, e.g.
  - **Program Counter (PC)**
  - **Stack pointer (SP)**
  - **Input/Output registers** — 224 of them on the ATmega324A, see avr-io-ports[344]
  - **Status register** — N, Z, C, V among others. **Tanenbaum calls this the Program Status Word.**

[343] courses/csse2010/memory-models.pdf
[344] courses/csse2010/avr-io-ports.pdf

Some other registers are part of the microarchitecture, NOT the ISA — e.g. the **Instruction Register (IR)**. The programmer doesn't need to know about these, and **can't directly change or use them**.

That distinction is the cleanest illustration of what an ISA actually is. The IR unquestionably exists and holds the instruction being executed (fetch-decode-execute-cycle[345]) — but **no instruction names it**, so it is invisible to software and a chip designer is free to change or remove it. The PC is the opposite case: `jmp` writes it, so it has to be in the specification. **Existing in the hardware is not the criterion; being reachable from an instruction is.**

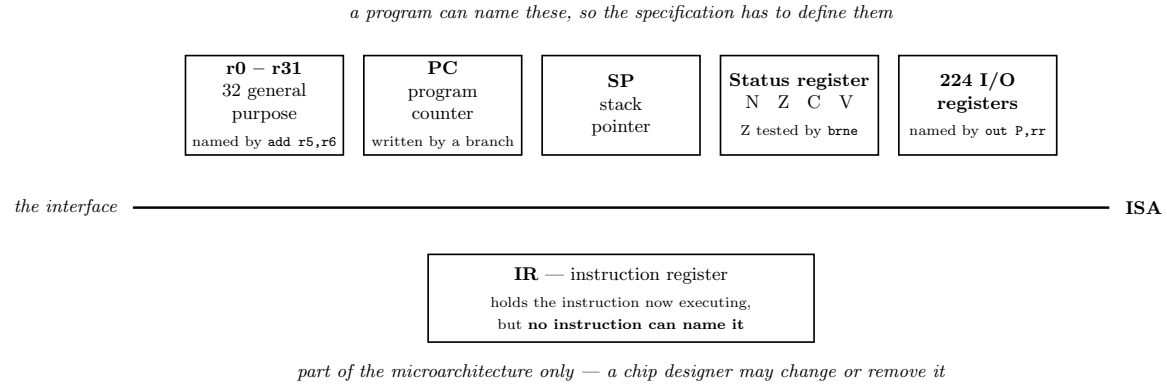


Figure 55: The same interface line as in the previous figure, with the machine's registers sorted across it. The five above the line are part of the ISA; four of them carry an example of an instruction that names the register, and the SP box carries none. The single box below, the IR, is part of the microarchitecture only. Nothing about a register's size or importance decides which side it lands on — only whether some instruction can refer to it.

The AVR's own answer to this pillar — the regularity of r0–r31, the r16–r31 exception, and the X/Y/Z pointer pairs — is in avr-architecture[346].

3. Instructions

This is the main feature of an ISA. Instruction types include:

Type	Does
Input/Output	communicate with I/O devices/ports
Load/Store	move data from/to memory

[345] courses/csse2010/fetch-decode-execute-cycle.pdf
[346] courses/csse2010/avr-architecture.pdf

Type	Does
Move	copy data between registers
Arithmetic	addition, subtraction, …
Logical	Boolean operations
Branching	for deciding which instruction to perform next

The **Input/Output** row is the one where the AVR's answer is least obvious. The ATmega324A's 224 I/O registers are all reachable by `ld`/`st` at their **data-memory** addresses 32 to 255 (`$20` to `$FF`), but only the **first 64** also get the short `in rd, P` / `out P, rr` form that names an I/O register by its own number `P` — because `P` is a six-bit field and $2^6 = 64$. The other 160, the **"extended" I/O registers**, can only be reached the long way. The mechanics, the numbering offset and the register list are in avr-io-ports[347].

With arithmetic and logical merged, the same set can be cut the other way instead — by **how many operands** an instruction takes:

- **Dyadic** — two operands, one result
- **Monadic** — one operand, one result

And **instructions use various addressing modes**: how an operand says *where* its value is, covered in addressing-modes[348].

The AVR instructions met so far, sorted into these boxes, are in avr-instructions[349].

4. Data types

Numeric

- **Integers** of different lengths (8, 16, 32, 64 bits), possibly signed or unsigned
- **Floating point** numbers, e.g. 32 bits (single precision) or 64 bits (double precision)
- Many others not listed

Non-numeric

- **Boolean** (0 means false, 1 means true) — stored in a whole byte or word
- **Bit-map** (collection of Booleans, e.g. 8 in a byte)
- **Characters**
- **Pointers** (memory addresses)

[347] courses/csse2010/avr-io-ports.pdf
[348] courses/csse2010/addressing-modes.pdf
[349] courses/csse2010/avr-instructions.pdf

Different machines provide hardware support for different data types. Two ends of the range:

Intel-64 Architecture

Data type	8 bits	16 bits	32 bits	64 bits	128 bits
Signed integer	✓	✓	✓	✓	
Unsigned integer	✓	✓	✓	✓	
BCD integer	✓				
Floating point			✓	✓	

Atmel AVR

Data type	8 bits	16 bits	32 bits	64 bits	128 bits
Signed integer	✓				
Unsigned integer	✓				
BCD integer					
Floating point					

The only data type an AVR supports in hardware is an 8-bit integer, signed or unsigned. There is no 16-bit integer type, no floating point, no BCD.

Other data types can be supported through software! e.g. **16-bit integer operations can be built out of 8-bit operations, using a sequence of instructions** — which is exactly what the two-instruction 16-bit addition in avr-instructions[350] does. So the hardware/software line here is about **cost, not possibility**: anything missing from the table can still be computed, it just takes more instructions and more cycles. Floating point is the costly case: one `float` multiply is hundreds of instructions (floating-point-numbers[351]), which is why fractions on the AVR are usually kept as fixed-point-numbers[352].

Assembly language

Machine code is too hard to program in. Assembly language is used instead:

- **Human readable** (just!).

[350] courses/csse2010/avr-instructions.pdf
[351] courses/csse2010/floating-point-numbers.pdf
[352] courses/csse2010/fixed-point-numbers.pdf

- **Converted into machine code by software called an assembler.**

Two Atmel AVR examples:

Assembly	Meaning
`add r3,r4`	`r3` ← `r3` + `r4`
`inc r22`	`r22` ← `r22` + 1

The important structural detail is in the arrows of Figure 54: a high-level language compiles **one-to-many** — one line of C becomes however many instructions it takes — whereas assembly assembles **one-to-one**. Each assembly instruction is one machine instruction, written in a form a person can read. That is the whole of what an assembler buys you, and it is why assembly is still "programming the machine" in a way C is not.

The toolchain

For CSSE2010 the assembler is **Microchip Studio**, and two supplied tutorials walk through it end to end: *Intro to Microchip Studio* (create an assembler project, pick the device, edit, build, and step through the simulator) and *Microchip Studio C Tutorial* (the same for a GCC C project). Both are click-through guides to follow at the machine rather than material to learn — they are on Blackboard, and the labs are where they get used.

See also

- memory-models[353] · addressing-modes[354] — pillars 1 and 3, in full
- cpu-organisation[355] · fetch-decode-execute-cycle[356] · state-machines[357]
- avr-architecture[358] · avr-instructions[359] · avr-io-ports[360] · avr-program-structure[361]
- assembly-process[362] — what the assembler does besides translating: directives, segments and two passes · compilation-and-linking[363] — the compiler's one-to-many road, through object files and the linker

[353] courses/csse2010/memory-models.pdf
[354] courses/csse2010/addressing-modes.pdf
[355] courses/csse2010/cpu-organisation.pdf
[356] courses/csse2010/fetch-decode-execute-cycle.pdf
[357] courses/csse2010/state-machines.pdf
[358] courses/csse2010/avr-architecture.pdf
[359] courses/csse2010/avr-instructions.pdf
[360] courses/csse2010/avr-io-ports.pdf
[361] courses/csse2010/avr-program-structure.pdf
[362] courses/csse2010/assembly-process.pdf
[363] courses/csse2010/compilation-and-linking.pdf

- 2026-08-27-avr-introduction[364] — Lecture 10
- 2026-08-31-instruction-set-architecture-part-1[365] — Lecture 11
- 2026-09-03-instruction-set-architecture-part-2[366] — Lecture 12

The I/O Board

The I/O board in the lab kit carries the input and output devices you wire your breadboard circuits to, for experimenting with simple digital circuits. It plugs directly onto the breadboard, and adds a **logic probe** for debugging and a **UART serial interface** for talking to other devices.

It is powered from a single USB connection and has **current-sense circuitry that detects short circuits and protects your designs**.

Pin numbers for every device are in device-pinouts[367]; the kit itself is loaned out in week3-lab-logisim-and-flip-flops[368]. Two more parts of the kit that are used alongside the board, the joystick and the LED matrix, are described at the end.

What's on it

1. USB connector	2. Power LED	3. Power switch	4. 7-segment display
5. Switches	6. LEDs	7. Buttons	8. UART transmit/receive LEDs
9. Logic probe indicator LEDs	10. Fault LED	11. VCC / GND headers	12. I/O device headers
13. Auxiliary power connector	14. High current mode jumper		

Items 13 and 14 are the auxiliary power connector and the high-current-mode jumper. The one extra power connection the labs use is connector J1, which powers the LED matrix in Lab 16 (below).

[364] courses/csse2010/2026-08-27-avr-introduction.pdf
[365] courses/csse2010/2026-08-31-instruction-set-architecture-part-1.pdf
[366] courses/csse2010/2026-09-03-instruction-set-architecture-part-2.pdf
[367] courses/csse2010/device-pinouts.pdf
[368] courses/csse2010/week3-lab-logisim-and-flip-flops.pdf

The board at a glance

The board is a **C shape**: a left-hand spine carrying the USB connector, the power switch, the two status LEDs and the seven-segment display, plus a top arm and a bottom arm that straddle the breadboard's central channel. All the I/O devices come out on the **two 16-pin headers** on the inner edges of those arms, and *every* device is reached through the column of breadboard holes directly above or below its printed label.

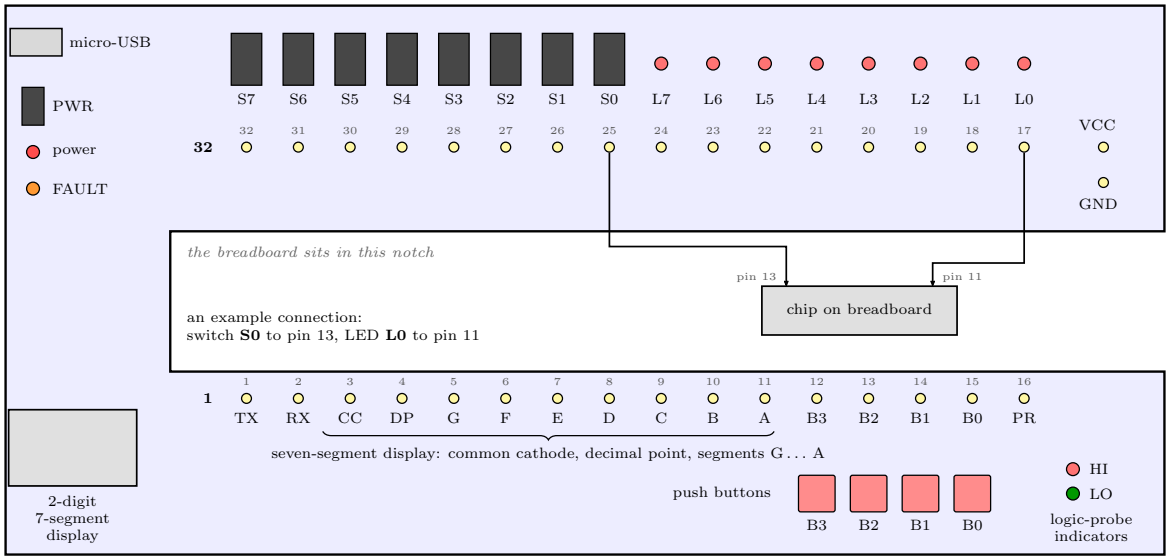


Figure 56: A plan view of the board, not to scale, with the parts on the left spine placed roughly. Each I/O device sits on one pin of one of the two 16-pin headers, and the column of breadboard holes at that pin is how you reach it. The board itself prints only the two end markers, a bold 32 at the left of the top header and a bold 1 at the left of the bottom one. The small numbers at the other pins are their pin numbers, the ones a schematic uses. The connection drawn in the notch is an example: switch S0 to pin 13 of a chip, LED L0 to pin 11.

Two things about the numbering are worth fixing in your head before you wire anything:

- **The top header counts down, left to right.** Its leftmost pin is 32 and its rightmost is 17, so the switches run `S7`…`S0` and then the LEDs `L7`…`L0` as you read left to right. That is the opposite of the reading order you would guess from the pin numbers alone.
- **The bottom header counts up, left to right**, pin 1 to pin 16: `TX RX CC DP G F E D C B A B3 B2 B1 B0 PR`. The middle nine of those are the seven-segment display — `CC` its common cathode, `DP` its decimal point, and `G` down to `A` its seven segments — which is why nine consecutive pins carry single-letter names.

`VCC` and `GND` pads sit at the far right of both arms, which is where the breadboard's power rails get fed from.

Setup

Insert the board into the breadboard, aligning the downward-facing headers with the right holes.

> If aligned correctly, the I/O board **should not require excessive force** to insert.

Connect a micro-USB cable to the connector at the **top-left** corner.

> **DO NOT STORE THE BOARD WITH THE USB CABLE ATTACHED — THIS MAY CAUSE THE USB CONNECTOR TO BREAK OFF!!**
>
> The USB connector is the part people break.

Turn the board on by sliding the **PWR** switch up to ON; the red power LED should light. Then **check the orange FAULT LED does not light**. If it does, power off immediately.

The fault LED

The board uses a **current-limited power switch** to protect itself and anything it powers from short circuits. A short lights the orange FAULT LED.

If it's on: push PWR down to OFF **immediately**, and check your design for shorts before restoring power.

> **WARNING! In the case of a short circuit, the current-limiting switch may become extremely hot.**

Switches, LEDs and buttons

All the I/O devices are reached through the breadboard holes **directly above or below the printed label** on the board:

- **Switches** — `S0` … `S7` (pins 25–32)
- **LEDs** — `L0` … `L7` (pins 17–24)
- **Buttons** — `B0` … `B3` (pins 15 down to 12)

Connect one by running a wire from the column of breadboard terminals above/below the label to wherever it's needed. An example, drawn into the notch of Figure 56: `S0` to pin 13 of an external device, `L0` to pin 11.

Switches and buttons are **inputs to your circuit**, LEDs and the seven-segment display are **outputs from it** — which is the distinction the pentagon symbols in circuit-schematics[369] make, and getting it backwards is one of the common errors listed there.

The logic probe

Attach a wire to the column of terminals above the `PR` label — pin 16, the rightmost pin of the bottom header. Touch the free end to any point in the circuit and the **HI / LO** LEDs show the logic state there.

> As circuits increase in complexity, so does the necessity of having a clear strategy — **probing random points in a large circuit rarely reveals the location of errors**.

Suggested troubleshooting strategy

1. **Check the FAULT light is not on**, which would indicate a short circuit.
2. **Probe VCC and GND at every device** in your circuit, to confirm each one is actually powered.
3. **Start at the inputs and move logically through to the outputs.** Pen and paper helps: write down the *expected* state at each point first, then verify it with the probe. A schematic and a truth table are great for this.

   > The key to any successful debugging strategy is to **assume nothing, know what to expect before you begin testing, and have a plan.**

Step 2 is the one worth internalising — an unpowered chip looks exactly like a wrong-logic chip until you probe its power pins, and grouped VCC/GND entries on a schematic (see circuit-schematics[370]) make it easy to forget one.

Electrical characteristics

VCC	5 V
GND	0 V
Logic high	> 3.5 V

[369] courses/csse2010/circuit-schematics.pdf
[370] courses/csse2010/circuit-schematics.pdf

Logic low	< 1.5 V

The gap between 1.5 V and 3.5 V is *undefined*, which is why a floating pin can read as either (week3-lab-logisim-and-flip-flops[371]). What the logic probe shows for a voltage inside that band is not documented.

The joystick

The kit's two-axis joystick is a small board of its own, made to sit in the breadboard next to the I/O board. Each axis is a **potentiometer** between its own supply pin and ground, and the stick moves the wiper. So each axis gives a voltage that follows the stick across its travel, from 0 V at one side to 5 V at the other, for the A/D converter to read (analog-to-digital-conversion[372]).

Pin	What it is	Connect to
L/R+	top of the left-right potentiometer	VCC
U/D+	top of the up-down potentiometer	VCC
GND ×2	bottom of both	GND, either one
L/R ×2	**x**: 0 V with the stick to the left, 5 V to the right	an analog input, either one
U/D ×2	**y**: 0 V at the bottom, 5 V at the top	an analog input, either one

- **The doubled pins are joined on the board.** One wire to each of L/R, U/D and GND is enough.
- **The two + pins are separate.** Both have to go to VCC, or one axis has nothing to divide.
- **Left, right, up and down are the joystick's own,** marked by the X and Y arrows printed on it. Placed at the right-hand end of the breadboard, with room for the microcontroller board and programmer to its left, the X arrow points to the top of the breadboard and the Y arrow to the left.
- **A potentiometer's wiper is a weak source.** Anything that draws current from it, an LED for instance, pulls its voltage away from the stick's position. LEDs make a quick wiring test, but they must come off before the converter is used.
- **Keep the black foam** that protects the pins when the joystick is put away.

[371] courses/csse2010/week3-lab-logisim-and-flip-flops.pdf
[372] courses/csse2010/analog-to-digital-conversion.pdf

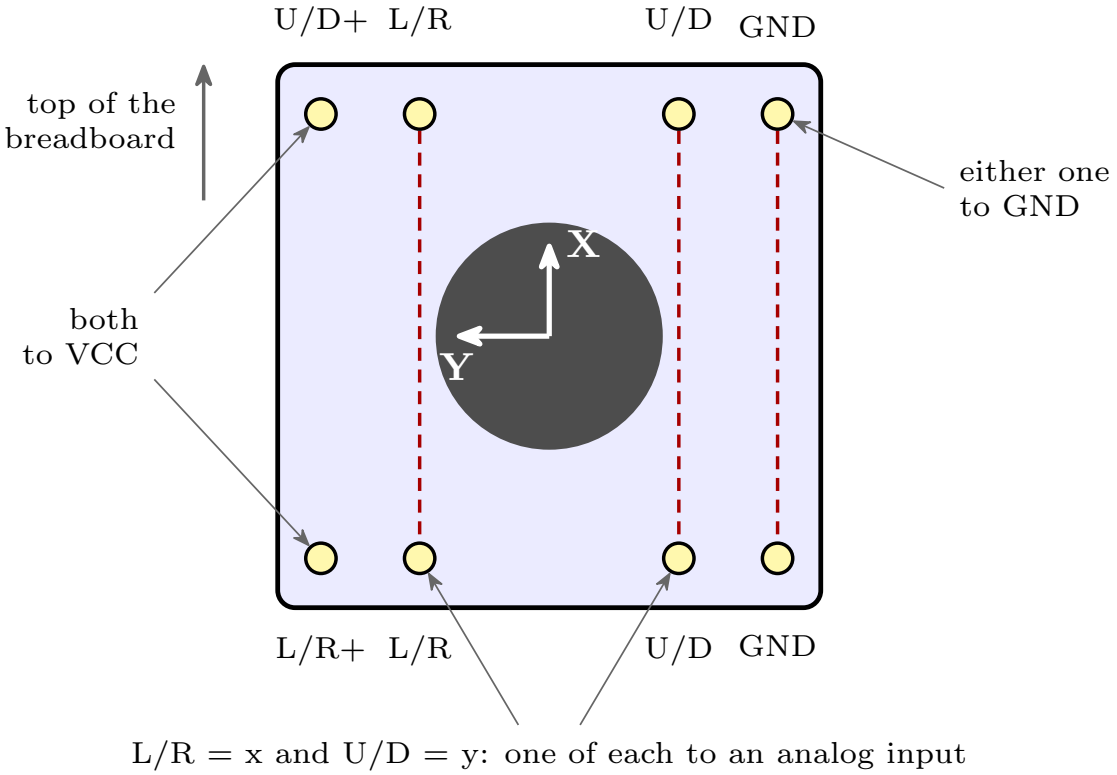


Figure 57: The joystick's eight pins, seen from above with the board placed as it sits in the breadboard, its X arrow pointing to the top. The pins near each corner come in twos. The dashed lines join the pins that are connected on the board itself: the two L/R pins, the two U/D pins and the two GND pins, so one wire to each pair is enough. The two supply pins, U/D+ at the top left and L/R+ at the bottom left, are not joined, and both go to VCC. L/R carries x and U/D carries y to the analog inputs.

The LED matrix

The LED matrix is a display of **16 columns by 8 rows**, each pixel a red LED and a green LED together. It is driven over **SPI**: the AVR is the master and the matrix a slave that takes commands one byte at a time (avr-spi[373]).

Connecting it

It has two headers:

- **K1**, two pins, is its **power**. A two-wire lead joins it to the I/O board's connector **J1**, VCC on J1's left pin and GND on its right.
- **K2**, four pins, is the **SPI connection**, to the AVR's SPI port on PB7 to PB4: SCK on PB7, MISO on PB6, MOSI on PB5 and $\overline{\text{SS}}$ on PB4. Match the order of the pins on K2 against the datasheet before plugging the lead in.

The Err state. PB7 to PB5 are also the pins the programmer uses to load the AVR, so programming sends the matrix a stream of what look to it like garbage commands, and it can show **Err**. Holding the AVR's reset button down for at least half a second after programming clears it before the program starts.

What to send it

Every message starts with a **command byte**. Two commands are enough to draw anything:

Command	Bytes	Does
clear	`0x0F`	every pixel off
set one pixel	`0x01`, *position*, *colour*	sets one pixel to a colour

- **Position byte: `(y << 4) | x`.** The row y, from 0 at the bottom to 7 at the top, is the high nibble. The column x, from 0 at the left to 15 at the right, is the low nibble.
- **Colour byte: `(green << 4) | red`.** Each nibble is a brightness from 0 to 15. Red alone is `0x0F`, green alone `0xF0`, both full is yellow, `0xFF`, and full red with half green is orange, `0x7F`.

The matrix's reference manual is the full description of the commands it accepts.

Lab 16 works through the commands and uses them for a demo, a scrolling marquee across two matrices and an Etch-A-Sketch (week10-lab-adc-and-spi[374]).

[373] courses/csse2010/avr-spi.pdf
[374] courses/csse2010/week10-lab-adc-and-spi.pdf

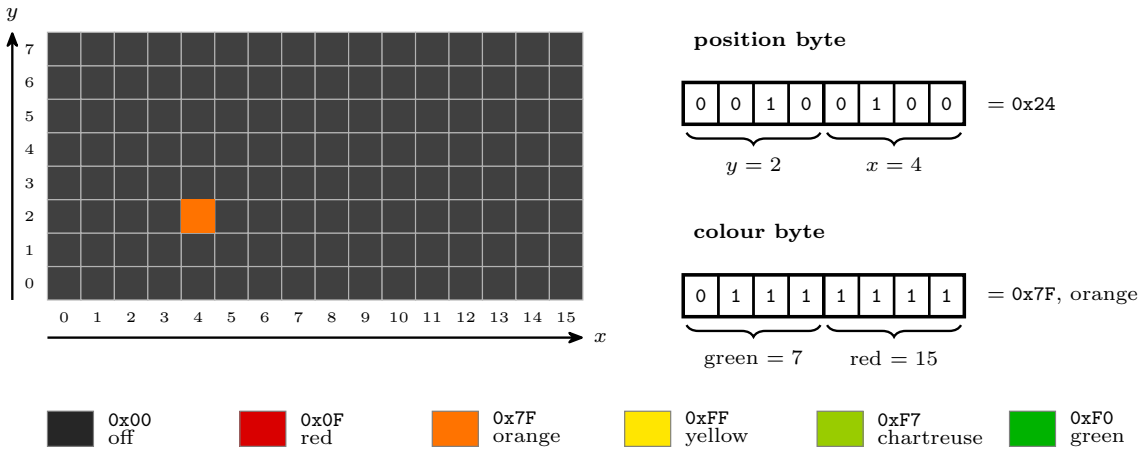


Figure 58: How a pixel is addressed and coloured. The matrix is 16 columns, x from 0 to 15 left to right, by 8 rows, y from 0 to 7 bottom to top. A position byte puts y in its high nibble and x in its low one, so the pixel at x = 4, y = 2 is `0x24`. A colour byte puts the green brightness in its high nibble and the red in its low one, each 0 to 15. The swatches show what some colour bytes look like: the colours here are the light from the red and green LEDs mixed.

See also

- device-pinouts[375] — the pin table behind Figure 56, and the board's schematic symbols
- circuit-schematics[376] — how the board is drawn on a schematic (`IO1` / `IOBOARD`, input and output pentagons)
- avr-io-ports[377] — the same switches and LEDs driven from an AVR rather than from gates
- week3-lab-logisim-and-flip-flops[378] — where the kit is loaned out
- week10-lab-adc-and-spi[379] — Lab 16: the joystick read by the A/D converter, and the LED matrix driven over SPI
- analog-to-digital-conversion[380] · avr-spi[381] — the two AVR peripherals the joystick and the matrix are used with

[375] courses/csse2010/device-pinouts.pdf
[376] courses/csse2010/circuit-schematics.pdf
[377] courses/csse2010/avr-io-ports.pdf
[378] courses/csse2010/week3-lab-logisim-and-flip-flops.pdf
[379] courses/csse2010/week10-lab-adc-and-spi.pdf
[380] courses/csse2010/analog-to-digital-conversion.pdf
[381] courses/csse2010/avr-spi.pdf

Memory Models

The first of the four things that make up an ISA (instruction-set-architecture[382]): **how memory looks to the CPU**. Four issues decide that — **addressable cell size**, **alignment**, **address spaces** and **endianness** — and every one of them is a decision the specification has to make, not something the hardware settles on its own.

Memory as addressable cells

Computer memory is conceptually a **set of addressable cells**:

- Each cell has a **unique address** (or cell number).
- Cells are **usually 8 bits wide** (i.e. hold one byte).
- **Various cell widths have been used in the past** — 8 bits is a convention, not a law.

Drawn as a table, the address indexes the rows and the cell width is how wide each row is:

Address	← cell width →
0	
1	
2	
3	
…	

A "**4 × 3 memory**" in that notation means **4 addresses of 3-bit cells** — and $4 = 2^2$, so two address bits. Built at gate level, it shows that "memory" at this level is nothing more exotic than flip-flops-and-latches[383] and combinational-logic-blocks[384] arranged into storage.

A. Addressable cell size

Most common cell size is 8 bits (1 byte) — but not always.

The AVR is the counterexample sitting on the bench:

AVR memory	Cell width
Data memory	8 bits
Instruction (program) memory	**16 bits**

[382] courses/csse2010/instruction-set-architecture.pdf
[383] courses/csse2010/flip-flops-and-latches.pdf
[384] courses/csse2010/combinational-logic-blocks.pdf
[385] courses/csse2010/csse2010-diagram-notation.pdf

AVR memory	Cell width

Which is why an AVR instruction is "one word" rather than "two bytes" — a program-memory address counts 16-bit cells.

The **bus** transports the content of a cell, **but the data bus may be wider** than one cell: it can retrieve **several cells (addresses) at once**. When it does, the **address bus does not need to be as wide**.

Bus sizes

> **For every doubling of the data bus width, remove the least significant bit of the address bus.**

For a memory of 2^n cells at 8 bits per cell — so an address size of n bits:

Data bus	Address bus	Cells per transfer
8 bits	n bits	one cell at a time
16 bits	$n-1$ bits	two cells at a time
32 bits	$n-2$ bits	four cells at a time

The reason is that a wide transfer always moves an **aligned group** of cells. With a 16-bit data bus the memory only ever hands over cell pairs $(0,1),(2,3),(4,5),\ldots$ — so the bottom address bit never selects anything, and naming the *pair* takes one bit fewer than naming the *cell*.

Worked example. A memory of 64 cells, 8 bits each, addressed `000000` … `111111`:

$$64 = 2^6 \text{ cells} \implies 6\text{-bit addresses}$$

Data bus	Address bus
8	6
16	5
32	4

Read the two buses as a pair, not separately: widening one narrows the other by exactly as much.

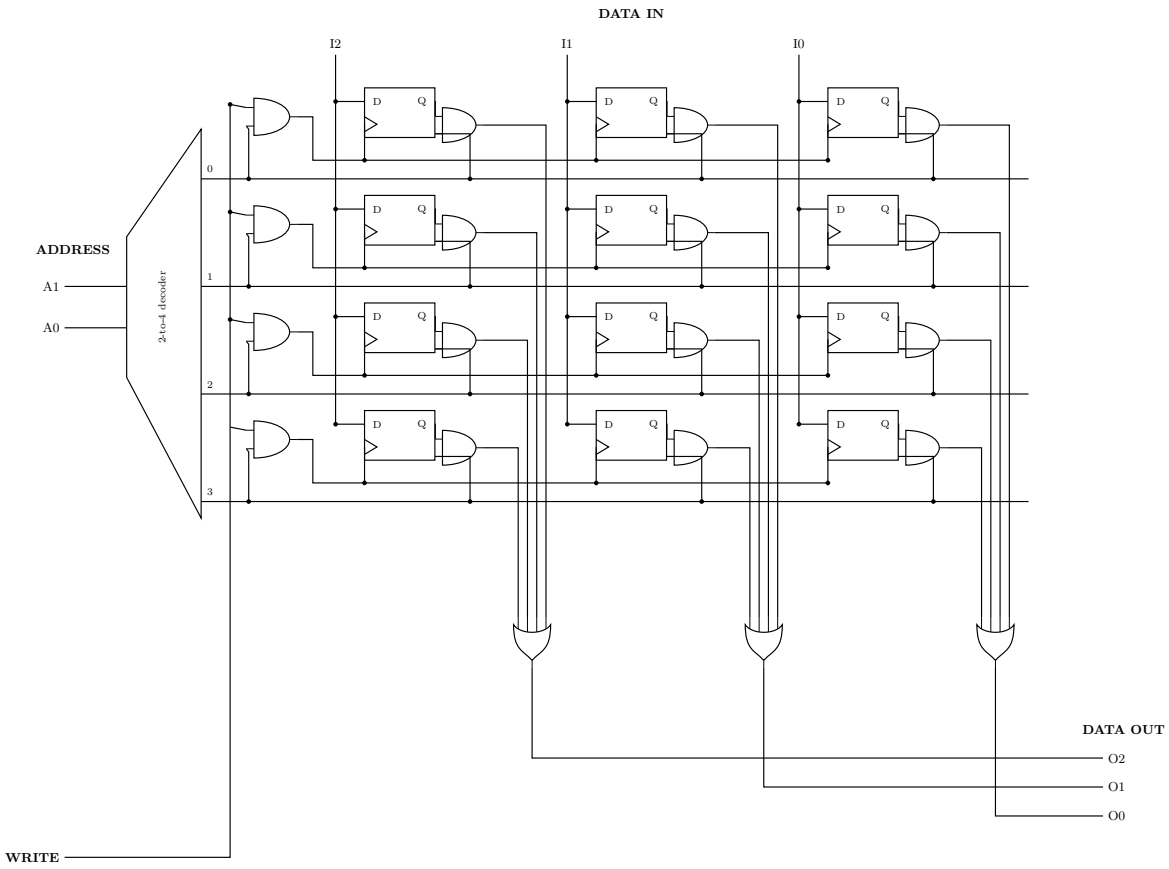


Figure 59: A 4×3 memory at gate level — four addresses of three-bit cells. Twelve D flip-flops in four rows of three: one **row** is one addressable cell. The decoder turns `A1 A0` into exactly one row-select (word) line. **Writing:** the word line ANDed with `WRITE` clocks that row's three flip-flops, so `DATA IN` (`I2 I1 I0`) — which is wired to every row's D inputs all the time — is captured only by the selected row. **Reading:** each flip-flop's Q is ANDed with its own word line and the three cells of a column are OR'd together, so the unselected rows contribute 0 and `DATA OUT` (`O2 O1 O0`) carries the selected row alone. The decoder is drawn as a trapezoid with the wide edge on the right, and the flip-flops as plain rectangles with a clock wedge on the lower-left edge, per this course's conventions (csse2010-diagram-notation[385]). The most significant bit, `I2`/`O2`, is the leftmost column.

Word addresses and byte addresses

The AVR's 16-bit program-memory cells have one practical consequence. The program counter, `jmp`, `call` and every code label count **words**. But a table of constant bytes kept in flash is read a byte at a time by `lpm`, *load program memory*, and `lpm` takes a **byte address** in the Z register. Word w holds bytes $2w$ and $2w+1$, with the **low byte at the even address**, which is the AVR being little endian (section D).

So turning a code label into a byte address means **doubling it**. Take the string `Hello` stored at word 2 by `mesg: .db 72,101,108,108,111` (assembly-process[386]):

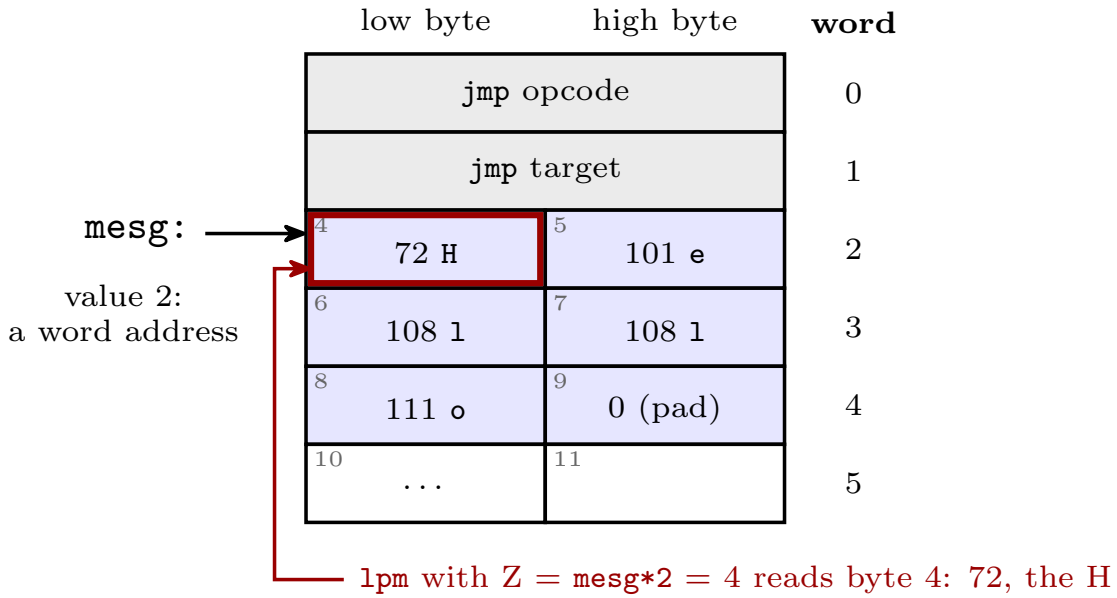


Figure 60: One region of program memory addressed two ways. Each row is a 16-bit word, numbered on the right, and each half is a byte, numbered inside its cell. The label `mesg` has the value 2 because labels in the code segment count words. Byte addressing counts the halves, low byte first, so word 2 starts at byte 4. That is why `lpm` is pointed at `mesg*2`: with Z = 4 it loads 72, the H. The five bytes of the string take two and a half words, and the last half word is a zero pad. Grey marks the `jmp`, blue the string and red the byte that `lpm` reads.

```
        ldi  ZH, high(mesg*2)    ; Z = 4: byte address of word 2
        ldi  ZL, low(mesg*2)
        lpm                      ; r0 <- 72, the H
```

[386] courses/csse2010/assembly-process.pdf

Leaving out the `*2` points Z at byte 2, the low byte of word 1, which is half of the `jmp`.

B. Alignment

Many architectures require natural alignment, e.g.

- 4-byte words starting at addresses 0, 4, 8, …
- 8-byte words starting at addresses 0, 8, 16, …

i.e. an n-byte value must start at an address that is a multiple of n.

Taking an 8-byte word on a machine that fetches 8 bytes at a time:

	Where it sits	Reads needed
Aligned	bytes 8–15, i.e. an 8-byte word at address 8	**1** — it is exactly one fetched block
Non-aligned	bytes 12–19, i.e. an 8-byte word at address 12	**2** — the top half of block 8–15, plus the bottom half of block 16–23

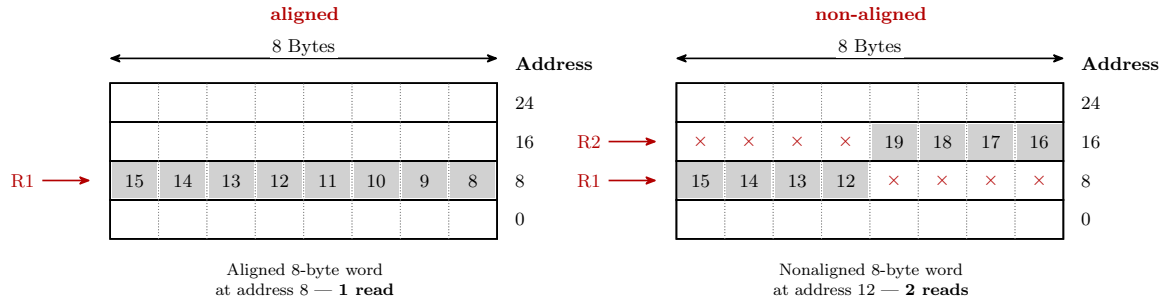


Figure 61: An aligned and a non-aligned 8-byte word. Four blocks of eight bytes; grey marks the bytes of the 8-byte word. On the left the word starts at address 8, so it *is* one block — one read. On the right it starts at address 12, so it takes the top half of the block at 8 and the bottom half of the block at 16 — two reads, marked **R1** and **R2**. Two things about the drawing: **addresses increase upward** (unlike every other memory figure in the course, where they increase downward — see csse2010-diagram-notation[387] on that inconsistency), and **byte numbers within a row run high-to-low left to right**. The × marks a byte in the fetched block that is not part of the word.

[387] courses/csse2010/csse2010-diagram-notation.pdf

Alignment is often required because it is more efficient — the aligned case is always one read, the non-aligned case sometimes two, and the hardware that stitches two blocks together and extracts the middle is hardware that a machine demanding alignment simply does not need.

When alignment is *not* required — the Pentium IV

- **Fetches 8 bytes at a time** from memory (64-bit wide data bus).
- **Alignment is NOT required**, so the processor stays **backwards compatible** with earlier processors.
- A 4-byte word stored at address 4 takes **1 read** (bytes 0 to 7).
- A 4-byte word stored at address 6 must read bytes 0 to 7 (one read) **and** bytes 8 to 15 (second read), then **extract the 4 required bytes from the 16 bytes read**.

Counting reads in general

A data bus w bytes wide delivers one **aligned** w-byte block per read. An n-byte value starting at address a therefore costs

$$\left\lceil \frac{(a \bmod w) + n}{w} \right\rceil \text{ reads}$$

The best case is n/w (rounded up), when the value starts on a block boundary; every other starting offset can push it one block further. So a value read from an arbitrary address costs **either $\lceil n/w \rceil$ or one more than that** — which is why questions of this shape almost always answer "k or $k+1$" rather than a single number.

Worked example: an 8-byte value on a 16-bit data bus, no natural alignment required. Here $w = 2$ and $n = 8$:

Starting address	Spans	Reads
Even	4 aligned 2-byte blocks	**4**
Odd	9 byte-positions, so 5 aligned blocks	**5**

so the answer is **4 or 5**. The odd case is the one to picture: one byte of the value sits alone at the top of the first block and one byte alone at the bottom of the last, and each of those still costs a whole read.

C. Address spaces

Many microprocessors have a single linear memory address space — the **von Neumann architecture**:

- e.g. 2^{32} bytes numbered from 0 to $2^{32} - 1$.
- (These may not be bytes — it depends on the addressable cell size.)
- Program and data share the one space.

However, the Harvard architecture has **separate address spaces for instructions and data**. The same numeric address means a different location depending on which space you are in, and an instruction says which one it means by *what instruction it is*.

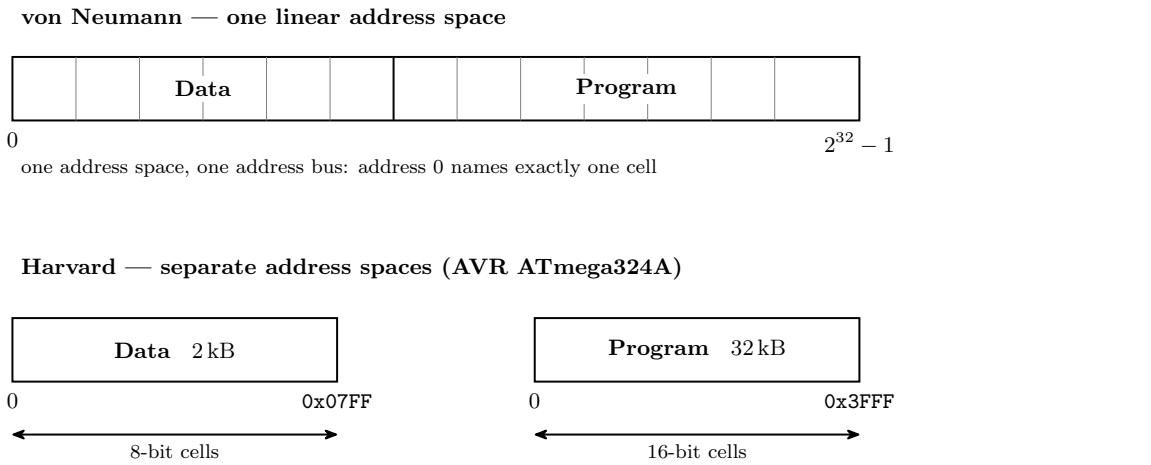


Figure 62: The two address-space models. Above: von Neumann — one linear space, so program and data are separated only by convention and address 0 names exactly one cell. Below: Harvard, in the AVR ATmega324A's shape — two spaces, each starting at its own 0, with different sizes *and* different cell widths — and the cell width under each box is the whole reason the arithmetic below works out the way it does: 32 kB of program memory is 2^{15} bytes but only 2^{14} addresses, because each address there names a 16-bit cell. The end addresses follow from the sizes, so the data box ends at `0x07FF` ($2^{11} - 1$, for 2048 bytes) rather than the `0x08FF` of Figure 63, which is that 2 kB *plus* the 256 locations in front of it.

AVR ATmega324A is Harvard:

Space	Size	Address width
Data	2^{11} (2048) bytes	11 bits

Space	Size	Address width
Instruction	2^{14} 16-bit words — 16,384 instruction words = **32,768 bytes**	14 bits

The arithmetic — convert to cells first, *then* take the logarithm:

$$\text{data: } 2\,\text{kB} = 2 \cdot 2^{10}\,\text{B} = 2^{11} \text{ cells (8-bit)} \implies 11 \text{ bits}$$

$$\text{program: } 32\,\text{kB} = 2^{5} \cdot 2^{10}\,\text{B} = 2^{15}\,\text{B} = 2^{14} \text{ cells (16-bit)} \implies 14 \text{ bits}$$

The program-memory step is the one that catches people: 32 kB of flash is only 2^{14} addresses, not 2^{15}, because each address names a **16-bit** cell. Halve the byte count before taking the log whenever the cells are not bytes.

Laid out as a memory map (see avr-architecture[388] for the rest of the chip):

Two details worth pinning down. `0x3FFF` $= 2^{14} - 1$, which is the 14 bits above. And the data space runs to `0x08FF` — that is the 2 kB of SRAM **plus** the 256 locations (`0x0000`–`0x00FF`) that the register file and I/O registers occupy in front of it; the "2^{11} bytes" above is the SRAM itself, which is what the 11 bits size.

The assembler's view: segments

An assembler treats each address space as a **segment** with its own location counter (assembly-process[391]). On the ATmega324A there are three, because the **EEPROM** is a third address space alongside the two above:

Segment	Directive	Address space	Cell	Counter starts at	The assembler can
code	`.cseg`	program memory, flash, 32 kB	16-bit word	`0x0000`	place instructions and constants
data	`.dseg`	data memory, SRAM	byte	`0x0100`	**only reserve space**
EEPROM	`.eseg`	EEPROM, 1 kB	byte	`0x0000`	place constants

[388] courses/csse2010/avr-architecture.pdf
[389] courses/csse2010/csse2010-diagram-notation.pdf
[390] courses/csse2010/avr-io-ports.pdf
[391] courses/csse2010/assembly-process.pdf

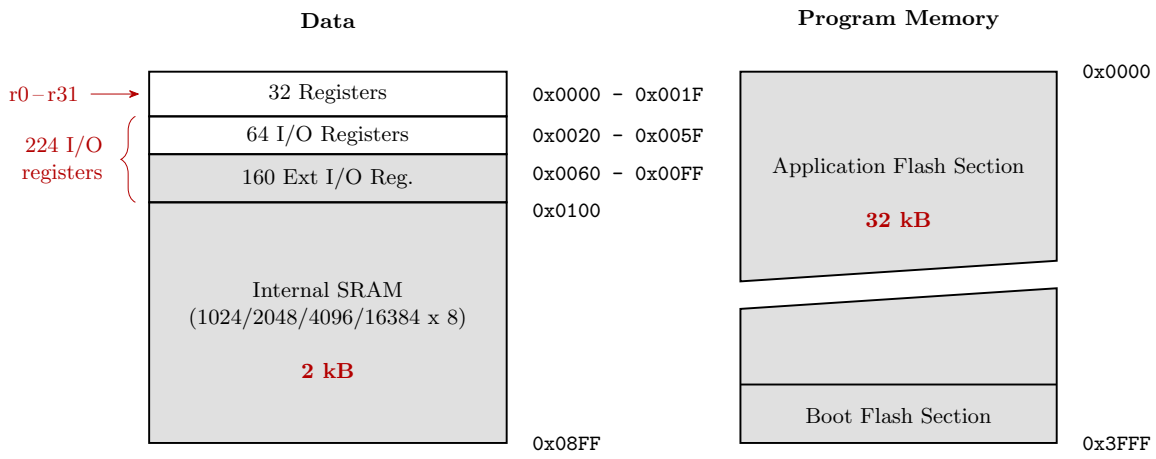


Figure 63: The ATmega324A's two address spaces. Lowest address at the top, hex addresses outside on the right, grey fill on the coarsely-described regions, and the diagonal break across the flash meaning *not to scale*: that is this course's memory-map convention (csse2010-diagram-notation[389]). Marked in red: the register file is `r0`–`r31`, the 64 I/O plus 160 extended I/O registers are the 224 of avr-io-ports[390], the SRAM is 2 kB and the flash 32 kB.

Three consequences of separate spaces:

- **The same number means different places.** Address 0 is the reset `jmp` in the code segment, register `r0` in data memory and the first EEPROM byte in the EEPROM segment.
- **The units differ.** `.org 0x0100` in the code segment is word 256, which is bytes 512 and 513 of flash. In the data segment it is byte `0x0100`.
- **The data counter does not start at 0.** The first 256 data addresses are the registers and I/O registers of Figure 63, so the first variable lands at `0x0100`.

The data segment can only be reserved, not filled, because the assembler's output is programmed into flash and EEPROM, which keep their contents, while SRAM starts empty at every power-up.

The memory behind each space

The three spaces are three different kinds of memory, and the kind explains how each is used (memory-technologies[392]):

[392] courses/csse2010/memory-technologies.pdf

Space	Technology	Volatile?	Erased and rewritten
Program memory	**flash**	no	electrically, **a block at a time**
Data memory	**SRAM**, flip-flops	**yes**	written a byte at a time, freely
EEPROM	**EEPROM**	no	electrically, **one cell (byte) at a time**

The flash is divided into **two blocks**, the **Application Flash Section** and the **Boot Flash Section** of Figure 63. Because flash is erased a block at a time, a boot loader in the boot section can erase and rewrite the application section without erasing itself. Other AVR devices have one flash block, and some have more.

The register file and the SRAM sit at the top of the memory hierarchy[393]. Unlike a desktop processor, the AVR has no cache: its SRAM is small enough and fast enough to keep up with the CPU directly.

D. Endianness

Different machines may support different byte orderings for a multi-byte value.

	Which end goes first (at the lowest address)	Used by
Little endian	the **little** end — least significant byte	Intel microprocessors (Pentium etc.), and **AVR**
Big endian	the **big** end — most significant byte	SPARC, Freescale (formerly Motorola) microprocessors

Most desktop/workstation/server CPUs produced since ~1992 are "bi-endian" (support both) — **some switchable at boot time**, **others at run time**, i.e. they can change dynamically.

Endianness only ever matters when a value spans **more than one cell**: reassembling it means knowing which cell was the low-order one.

[393] courses/csse2010/memory-hierarchy-and-cache.pdf

Worked example

Given this memory:

Address	Contents
0	`00001111`
1	`00110011`
2	`11011101`
3	`00101100`

Read	Value	Why
the **1-byte** word at address 1	`00110011`	one cell — **endianness does not apply**
the **2-byte** word at address 0, **big endian**	`00001111 00110011`	big end first, so address 0 holds the *most* significant byte
the **2-byte** word at address 2, **little endian**	`00101100 11011101`	little end first, so address 2 holds the *least* significant byte and address 3 the most

The mechanical way to do these without getting turned around: write the addresses the value occupies in increasing order, then **assemble the answer most-significant-byte first**. Big endian reads the addresses forwards; little endian reads them backwards.

See also

- instruction-set-architecture[394] — the four pillars this is the first of
- avr-architecture[395] — the ATmega324A's two address spaces in full
- addressing-modes[396] — how an instruction names a memory location
- binary-number-representations[397] · flip-flops-and-latches[398] · combinational-logic-blocks[399]
- assembly-process[400] — segments, location counters, and `.db` packing two bytes to a word

[394] courses/csse2010/instruction-set-architecture.pdf
[395] courses/csse2010/avr-architecture.pdf
[396] courses/csse2010/addressing-modes.pdf
[397] courses/csse2010/binary-number-representations.pdf
[398] courses/csse2010/flip-flops-and-latches.pdf
[399] courses/csse2010/combinational-logic-blocks.pdf
[400] courses/csse2010/assembly-process.pdf

- compilation-and-linking[401] — where a C compiler puts globals, locals and the heap in data memory
- memory-technologies[402] — flash, SRAM and EEPROM as technologies, and why sizes are powers of 2
- memory-hierarchy-and-cache[403] — registers, cache, main memory and disk, by speed and cost
- 2026-08-31-instruction-set-architecture-part-1[404] — Lecture 11 · 2026-10-08-assembly-process[405] — Lecture 20 · 2026-10-22-memory-disks-and-files[406] — Lecture 24

Seven-Segment Display

How a digit is shown on the IO board's seven-segment display: the segment lettering, the byte that lights a pattern, the lookup table for all ten digits, and the common-cathode line that selects which of the two digits is lit. First met in 2026-09-10-more-c-programming-on-the-avr[407]; the board's header is in io-board[408] and the port registers in avr-io-ports[409].

Segments

Seven bars lettered **a** to **g**, plus a decimal point **dp**:

Segment	Position
a	top
b	upper right
c	lower right
d	bottom
e	lower left
f	upper left
g	middle
dp	the dot, lower right

— clockwise from the top for a to f, then g across the middle. Each segment is an LED; a **1 on its line lights it**.

[401] courses/csse2010/compilation-and-linking.pdf
[402] courses/csse2010/memory-technologies.pdf
[403] courses/csse2010/memory-hierarchy-and-cache.pdf
[404] courses/csse2010/2026-08-31-instruction-set-architecture-part-1.pdf
[405] courses/csse2010/2026-10-08-assembly-process.pdf
[406] courses/csse2010/2026-10-22-memory-disks-and-files.pdf
[407] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[408] courses/csse2010/io-board.pdf
[409] courses/csse2010/avr-io-ports.pdf

The byte

The display is driven from one 8-bit port, one bit per segment, in this order:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
dp	g	f	e	d	c	b	a

So the byte for a digit is read off by asking, segment by segment, whether it is lit. Two worked examples:

Digit	Lit	dp g f e d c b a	Sum	Decimal
0	a b c d e f	`0 0 1 1 1 1 1 1`	$32+16+15$	**63**
2	a b g e d	`0 1 0 1 1 0 1 1`	$64+16+8+3$	**91**

`PORTA = 63;` shows a 0; `PORTA = 91;` shows a 2. The 2 and both bytes are drawn in 2026-09-10-more-c-programming-on-the-avr[410].

All ten digits

The lookup table from the program below, decoded:

Digit	Decimal	Binary (dp g f e d c b a)	Segments lit
0	63	`0011 1111`	a b c d e f
1	6	`0000 0110`	b c
2	91	`0101 1011`	a b d e g
3	79	`0100 1111`	a b c d g
4	102	`0110 0110`	b c f g
5	109	`0110 1101`	a c d f g
6	125	`0111 1101`	a c d e f g
7	7	`0000 0111`	a b c
8	127	`0111 1111`	a b c d e f g
9	111	`0110 1111`	a b c d f g

Displays differ on whether 6 lights segment a, the top bar, and whether 9 lights segment d, the bottom bar. This table lights both, giving a hooked 6 and an underlined 9.

[410] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf

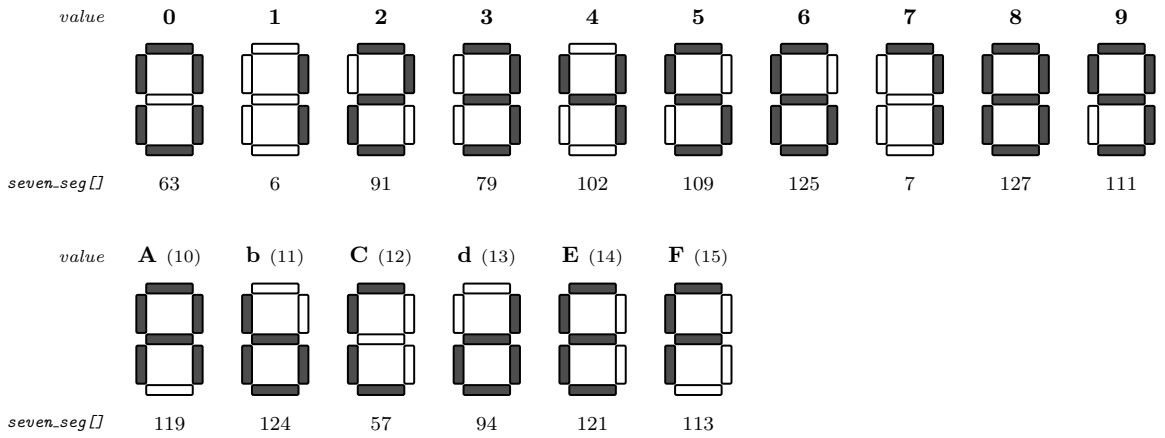


Figure 64: All sixteen values a nibble can hold, as seven-segment patterns, with the byte that produces each one in decimal underneath — the sixteen entries of the lookup table in order. Heavy segments are lit; the bit order is `dp g f e d c b a` with `a` at bit 0, so each number is read straight off the lit segments: 63 is every segment but `g`, 6 is `b` and `c` alone. The hex letters `b` and `d` are lowercase because their capitals would be indistinguishable from 8 and 0 on seven segments; `A`, `C`, `E` and `F` have no such clash.

Hex digits A–F

Four input bits hold 0–15, and the six values past 9 can be shown too, as hex letters. Two of them have to be lowercase: an uppercase **B** on seven segments is an **8**, and an uppercase **D** is a **0**.

Value	Shows	Segments lit	`dp g f e d c b a`	Decimal
10	**A**	a b c e f g	`0111 0111`	119
11	**b**	c d e f g	`0111 1100`	124
12	**C**	a d e f	`0011 1001`	57
13	**d**	b c d e g	`0101 1110`	94
14	**E**	a d e f g	`0111 1001`	121
15	**F**	a e f g	`0111 0001`	113

With these appended the table has sixteen entries and every value a masked nibble can produce is a valid index, so the guard in the program below can go:

```
uint8_t seven_seg[16] = { 63,6,91,79,102,109,125,7,127,111,
                          119,124,57,94,121,113 };
```

This is Lab 11's Task 2 (week8-lab-avr-c-programming[411]), and the same table drives the hex adder of its Task 3.

The program

```
#include <avr/io.h>

/* Seven segment display values */
uint8_t seven_seg[10] = { 63,6,91,79,102,109,125,7,127,111};

int main(void) {
  uint8_t digit;

  /* Set port A pins to be outputs, port C pins to be inputs */
  DDRA = 0xFF;
  DDRC = 0;    /* This is the default, could omit. */
  while(1) {
    /* Read in a digit from lower half of port C pins */
    /* We read the whole byte and mask out upper bits */
    digit = PINC & 0x0F;
    /* Write out seven segment display value to port A */
    if(digit < 10) {
      PORTA = seven_seg[digit];
    } else {
      PORTA = 0;
    }
  }
}
```

Port A drives the eight segment lines; the low four bits of port C are the input digit. The loop reads the digit, masks off the upper nibble (bit-masking[412]), looks the pattern up and writes it. **The `if` guards the table** — four bits can be 10–15, and `seven_seg[10]` is past the end of the array (c-pointers-and-arrays[413]); those inputs blank the display instead.

Lookup table, not arithmetic. There is no formula that turns a digit into its segment pattern, so the program does not try: the ten answers are precomputed and indexed. It is the general pattern for any input-to-output mapping that is small and irregular, and it is what the array is *for*.

[411] courses/csse2010/week8-lab-avr-c-programming.pdf
[412] courses/csse2010/bit-masking.pdf
[413] courses/csse2010/c-pointers-and-arrays.pdf

Two digits, one set of lines: `CC`

On the IO board the display has **two digits** but the header carries **one** set of segment pins — `DP`, `G` … `A` — plus one more, `CC`, the *common cathode* (io-board[414]). The segment lines go to both digits; `CC` decides which one is actually lit:

`CC`	Lit
1	left digit
0	right digit

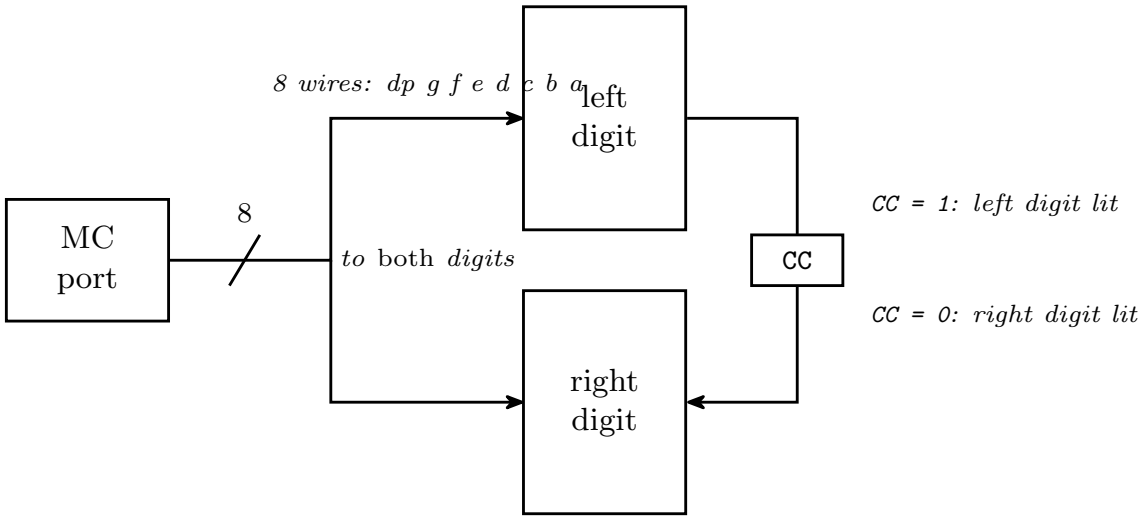


Figure 65: Why one port can drive two digits. The eight segment lines are wired to both digits in parallel, so both always receive the same pattern; the `CC` line selects which digit's common cathode is connected to ground, and only that digit lights. One byte and one select bit therefore describe the whole display at any instant. To show two *different* digits the program alternates — left pattern with `CC = 1`, then right pattern with `CC = 0` — fast enough that the eye sees both at once.

Writing a digit therefore takes **two outputs**: the segment byte on port A, and `CC` on whichever pin it is wired to. The program above drives a single digit and never touches `CC`; in Lab 11 `CC` is simply jumpered to a button, so pressing it moves the same digit across.

Display multiplexing

[414] courses/csse2010/io-board.pdf

Showing a two-digit number means **time-sharing** the segment lines: write the tens pattern with `CC = 1`, wait, write the ones pattern with `CC = 0`, wait, repeat. Alternate fast enough — every millisecond, say — and persistence of vision merges the two into a steady two-digit display, each digit at half brightness because it is lit half the time. Slow the alternation to a few times a second and the trick becomes visible, with the digits taking turns. The waiting is a timer's job (avr-timers-and-counters[415]); the whole program, and a timing diagram of the two digits taking turns, is Lab 12's Task 2 (week8-lab-avr-timers[416]). Two things the ordering has to get right: write the segments *before* switching `CC`, so the newly selected digit never flashes the other digit's pattern, and keep the tick regular, or one digit will look brighter than the other.

See also

- io-board[417] — the header pins `CC DP G F E D C B A`, and where the display sits on the board
- avr-io-ports[418] — `DDRA`, `PORTA`, `PINC`
- bit-masking[419] — `PINC & 0x0F`
- c-pointers-and-arrays[420] — the lookup table
- c-preprocessor-and-avr-headers[421] — why `PORTA` can be assigned to
- 2026-09-10-more-c-programming-on-the-avr[422] — Lecture 14
- week8-lab-avr-c-programming[423] — Lab 11: the display wired up, hex digits, the adder · week8-lab-avr-timers[424] — Lab 12: display multiplexing on a timer tick · week9-lab-avr-interrupts[425] — Lab 14: the display as a stopwatch

[415] courses/csse2010/avr-timers-and-counters.pdf
[416] courses/csse2010/week8-lab-avr-timers.pdf
[417] courses/csse2010/io-board.pdf
[418] courses/csse2010/avr-io-ports.pdf
[419] courses/csse2010/bit-masking.pdf
[420] courses/csse2010/c-pointers-and-arrays.pdf
[421] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[422] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[423] courses/csse2010/week8-lab-avr-c-programming.pdf
[424] courses/csse2010/week8-lab-avr-timers.pdf
[425] courses/csse2010/week9-lab-avr-interrupts.pdf