Addressing Modes

Table of contents

The four modes 1
Reading the trade-off 4
See also 4

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[1]) — see avr-instructions[2] 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`

[1] courses/csse2010/instruction-set-architecture.pdf
[2] courses/csse2010/avr-instructions.pdf

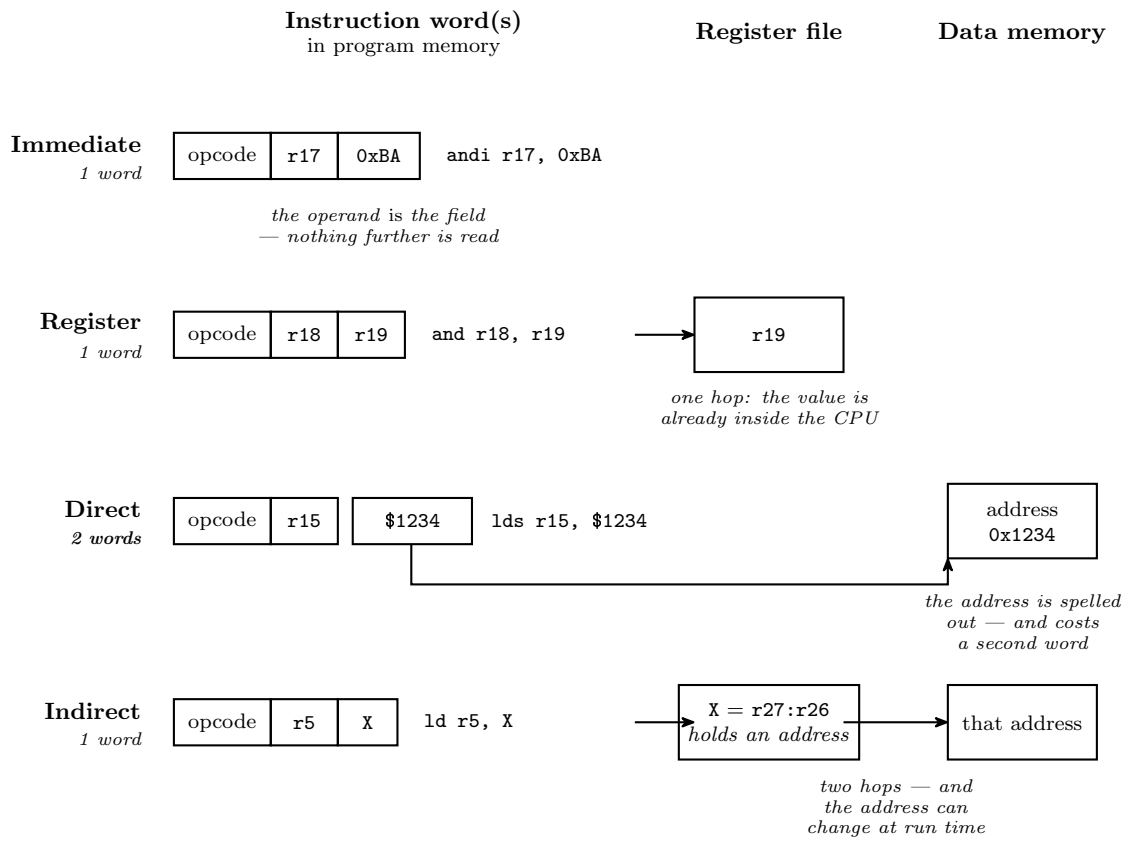


Figure 1: The four modes as one question: how far does the machine have to go to reach the operand? Immediate stops at the instruction itself. Register takes one hop, into the register file. Direct takes one hop into memory, but needs a second instruction word to carry the address. Indirect takes two 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.

Figure 1 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**.

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[3].

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.

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

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

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[4]).

See also

- instruction-set-architecture[5] — addressing modes as part of the "instructions" pillar
- avr-instructions[6] — `ld`/`lds`/`st`/`sts` and the immediate instructions
- avr-architecture[7] — the X, Y and Z pointer registers indirect mode uses
- memory-models[8] — the address space being addressed
- 2026-09-03-instruction-set-architecture-part-2[9] — Lecture 12

[4] courses/csse2010/avr-io-ports.pdf
[5] courses/csse2010/instruction-set-architecture.pdf
[6] courses/csse2010/avr-instructions.pdf
[7] courses/csse2010/avr-architecture.pdf
[8] courses/csse2010/memory-models.pdf
[9] courses/csse2010/2026-09-03-instruction-set-architecture-part-2.pdf