AVR Assembly Program Structure

Table of contents

The template 1
Comments and directives 2
Why the vector table comes first 2
Labels and `jmp` 3
Conditional control flow 4
When the program calls procedures 4
Variables and constants 6
Why it ends in an infinite loop 7
See also 7

The skeleton every AVR assembly program in this course is built on — what goes at the top, where execution starts, and why it ends in an infinite loop. See avr-instructions[1] for what fills it in and avr-io-ports[2] for what the initialisation section usually does.

The template

```
                            ; Comments start with semicolons
.include "m324Adef.inc"

    jmp RESET               ; <- vector table
    ; Interrupt handlers go here
    ; (more on this in week 9/10)

RESET:                      ; Starting point   <- RESET handler
    ...                     ; Initialisation code
                            ; e.g. set DDRs
```

[1] courses/csse2010/avr-instructions.pdf
[2] courses/csse2010/avr-io-ports.pdf

```
loop:                       ; <- main code
    ...                     ; Do stuff
    jmp loop                ; Repeat
```

Three sections, and each is there for a reason:

Section	What it is
Vector table	`jmp RESET`, then the interrupt handlers
RESET handler	the starting point — initialisation code, e.g. setting the data direction registers
Main code	the work, ending in a jump back to its own label

Comments and directives

Comments start with semicolons. Everything from a `;` to the end of the line is ignored by the assembler.

`.include "m324Adef.inc"` is an **assembler directive**, not an instruction — nothing is generated for it. **The contents of the file are included directly in the program**, which is what makes `DDRA`, `PORTC`, `TCNT0` and the rest usable as names instead of I/O register numbers (see avr-io-ports[3]). The leading dot is the giveaway: a line starting `.` speaks to the assembler, a line starting with a mnemonic speaks to the CPU.

Every line has the same shape, with up to four fields: a **label**, an **operation**, its **operands** and a **comment**, as in `loop:  jmp loop  ; Repeat`. The operation is either an instruction or, with the leading dot, a **directive**. Two more directives turn up in most programs: `.def temp = r16` gives a register a name, and `.equ` gives a number a name. The fields, and every directive with what it does, are in assembly-process[4].

Why the vector table comes first

The chip begins executing at program address `0x0000` — the top of the program memory map in memory-models[5] — and the addresses immediately after it are reserved for interrupt vectors, one per interrupt source. So **the very first instruction has to jump over them** to reach the real start of the program. That is all `jmp RESET` is doing.

[3] courses/csse2010/avr-io-ports.pdf
[4] courses/csse2010/assembly-process.pdf
[5] courses/csse2010/memory-models.pdf

Everything between that jump and the `RESET:` label is the vector table itself. It stays empty (comments only) in a program that uses no interrupts — but the `jmp` has to be there from the first program, because without it execution would fall straight into whatever occupied the vector slots. Once a program does use an interrupt, each vector slot it needs holds a `jmp` to that source's handler at a fixed address, placed with `.org`; the table, the handlers and how to enable them are in avr-interrupts[6].

Labels and `jmp`

A **label** is a name for a position in the program:

```
        jmp RESET
        …
RESET:  …
        …

loop:   …
        …
        jmp loop
```

The assembler replaces each label with the program-memory address it ended up at, so `jmp loop` becomes a jump to a real address. Two properties matter in practice:

- A label is written **at the start of the line, ending in a colon**; the instruction it names can be on the same line or the next one.
- Labels let you write code **without ever computing an address by hand**, which is most of what an assembler buys you over machine code (instruction-set-architecture[7]).

`jmp RESET` at the top of the template needs the address of a label the assembler has not reached yet. It copes by **reading the file twice**: the first pass only works out where every label lands, using each line's size, and the second fills the addresses into the instructions (assembly-process[8]). In the template `RESET` is the address straight after the vector table, worked out the same way.

`jmp` itself is unconditional. For jumping only *sometimes*, the branch instructions below take a label in exactly the same way.

[6] courses/csse2010/avr-interrupts.pdf
[7] courses/csse2010/instruction-set-architecture.pdf
[8] courses/csse2010/assembly-process.pdf

Conditional control flow

A branch instruction takes a label just as `jmp` does, but **jumps only if a status register bit says so** — so a decision is always written as two instructions, one that sets the flags and one that reads them:

`breq label`	branch if $\mathbf{Z = 1}$
`brne label`	branch if $\mathbf{Z = 0}$

The usual flag-setter is `cp`, which is **a subtraction whose result is not saved**: it computes `rd - rr` purely to set Z, N, C, V and the rest, so `cp r5, r6` followed by `breq` branches exactly when the two registers were equal. **The instruction reference specifies which status register bits are set by which instructions**, per instruction rather than by one general rule — so a compare is not always needed, because whatever ran last may already have set Z. avr-instructions[9] tabulates which does what.

Figure 1 is worth reading as the reason labels matter. The branch does not encode "go back three instructions" in a form you have to count out by hand; it names a label, and the assembler works out the address — the same service it performs for `jmp RESET` at the top of the file.

Two consequences for the template above. First, **a loop that should sometimes end is still written as a label plus a branch**, not as a special construct: the body falls through when the condition fails. Second, **nothing may sit between the flag-setter and the branch that disturbs the flag being tested** — the status register is a single shared piece of state, and the instruction in between will happily overwrite it.

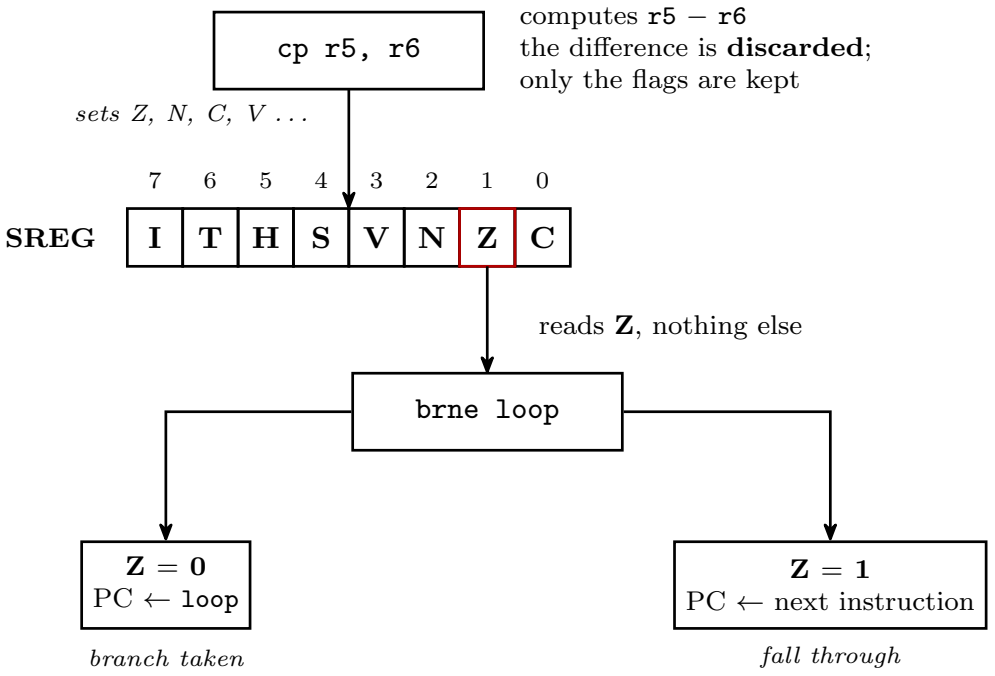
The two branches above are the equality pair. The other eight — including the signed and unsigned *less than* pairs that a loop bound usually needs — and the loop idioms built from them are in avr-branches-and-jumps[10].

When the program calls procedures

The template above has no stack, because nothing in it calls anything. The moment a program uses `call` or `rcall` — or `push` — the stack pointer must be set first, and it goes at the top of `RESET:`, before any of the other initialisation:

```
RESET:
    ldi  r16, low(RAMEND)
    out  SPL, r16
    ldi  r16, high(RAMEND)
    out  SPH, r16
```

[9] courses/csse2010/avr-instructions.pdf
[10] courses/csse2010/avr-branches-and-jumps.pdf



`breq loop` is the same picture with the test inverted: it branches when ***Z = 1****.*

Figure 1: Compare-then-branch: the whole of an AVR conditional. `cp` performs the subtraction and throws the difference away, keeping only what it did to the flags; `brne` then reads exactly one of them and either loads the label's address into the program counter or lets execution fall through to the next instruction. The flag names and their order are those of the AVR status register; the outline on Z marks the one flag `brne` tests.

```
    ...                         ; then set DDRs and the rest as before
```

`RAMEND` is the last SRAM address, `0x08FF`; the stack starts there and grows downward. Why it has to, and what `call` and `ret` do with it, is procedures-and-the-stack[11]. A C program gets these four instructions for free from the compiler's start-up code.

Variables and constants

The template holds only code. Variables and tables of constants go in the other **segments** (assembly-process[12]), chosen by `.dseg` and `.cseg`:

```
.include "m324Adef.inc"
.def    temp = r16              ; a name for a register

        .dseg                   ; SRAM: space only
count:  .byte 1                 ; one byte, at 0x0100

        .cseg                   ; flash: code and constants
        jmp  RESET              ; vector table
RESET:
        ldi  temp, low(RAMEND)  ; stack pointer, if anything is called
        out  SPL, temp
        ldi  temp, high(RAMEND)
        out  SPH, temp
        ...                     ; initialisation
loop:
        ...                     ; main code
        jmp  loop

digits: .db  63, 6, 91, 79, 102, 109, 125, 7, 127, 111   ; seven-segment patterns, 0
 ↪  to 9
```

Three rules come with it:

- **A variable is a label on a `.byte` in the data segment.** It gets an address, `0x0100` for the first one, but no value. The program has to store something there, with `sts count, temp` for example, before reading it.
- **Constants go in the code segment, out of the path of execution.** The CPU runs whatever is in program memory at the PC, so a table in the middle of the code would be executed as if it were instructions. After the main loop's `jmp loop` is safe, because execution never gets there. Between `jmp RESET` and `RESET:` is safe for the same

[11] courses/csse2010/procedures-and-the-stack.pdf
[12] courses/csse2010/assembly-process.pdf

reason, but only in a program that uses no interrupts, because that is where the interrupt vectors go. `digits` is ten bytes, an even number, so it fills five words with no pad (assembly-process[13]).

- **A constant is read with `lpm`, at its byte address**, the label times 2, in Z (memory-models[14]).

Why it ends in an infinite loop

`loop: … jmp loop` is not a placeholder. A microcontroller program has nowhere to return **to** — there is no operating system underneath it, and the CPU keeps fetching whatever comes next for as long as it has power. So the main body of an AVR program is a loop that never exits, and "do this repeatedly", which is how nearly every lab task is worded, is the normal shape rather than a special case.

The division of labour that follows: **anything that should happen once goes in the RESET handler** (setting DDRs, loading constants), and **anything that should happen forever goes in the loop** (reading inputs, computing, writing outputs).

See also

- avr-branches-and-jumps[15] — the full branch table, jumps, and the `if`/loop idioms · procedures-and-the-stack[16] — `call`, `ret` and the stack
- avr-interrupts[17] — what goes in the vector table, and the handlers after it
- avr-instructions[18] · avr-io-ports[19] · avr-architecture[20]
- assembly-process[21] — statement fields, directives, segments and the two-pass assembler · instruction-set-architecture[22] — assembly, the assembler and the toolchain
- memory-models[23] — the program address space the vector table sits at the bottom of · memory-technologies[24] — flash and SRAM, the memories behind `.cseg` and `.dseg`
- week6-lab-avr-io-ports[25] — Lab 10, where the template is first used

[13] courses/csse2010/assembly-process.pdf
[14] courses/csse2010/memory-models.pdf
[15] courses/csse2010/avr-branches-and-jumps.pdf
[16] courses/csse2010/procedures-and-the-stack.pdf
[17] courses/csse2010/avr-interrupts.pdf
[18] courses/csse2010/avr-instructions.pdf
[19] courses/csse2010/avr-io-ports.pdf
[20] courses/csse2010/avr-architecture.pdf
[21] courses/csse2010/assembly-process.pdf
[22] courses/csse2010/instruction-set-architecture.pdf
[23] courses/csse2010/memory-models.pdf
[24] courses/csse2010/memory-technologies.pdf
[25] courses/csse2010/week6-lab-avr-io-ports.pdf

- 2026-10-08-assembly-process[26] — Lecture 20: how the assembler turns the template into machine code

[26] courses/csse2010/2026-10-08-assembly-process.pdf