Lab 9 — Microchip Studio and AVR Assembly Introduction

Table of contents

What the lab covers . . . 1
Documentation . . . 2
Worked: what is the result of this program? . . . 2
Worked: the status register after a compare . . . 2
Boolean identities . . . 3
Bit masking . . . 3
Working — the arithmetic and logic program . . . 4
Working — the bit manipulation snippets . . . 6
See also . . . 7

Learning Lab 9, the first assembly lab. See avr-instructions[1] for the instruction set, bit-masking[2] for the masking technique the exercises turn on, and avr-architecture[3] for the status register.

What the lab covers

- AVR assembly language — instruction intro, documentation, machine code equivalent
- Microchip Studio (previously Atmel Studio) — assembling and simulating code

Objectives: you should know how to create a project for the ATmega324A, enter assembly language instructions, build the project, simulate instructions and examine register values.

[1] courses/csse2010/avr-instructions.pdf
[2] courses/csse2010/bit-masking.pdf
[3] courses/csse2010/avr-architecture.pdf

Documentation

- **ATmega324A datasheet** — have it open from the start of every remaining lab this semester.
- **AVR Instruction Set Manual** and **AVR Instruction Set Reference**
- On Blackboard: **Learning resources → Atmel AVR resources**
- Follow the **Microchip Studio tutorial** (`Intro_to_Microchip_Studio.pdf`) for the lab tasks — see instruction-set-architecture[4].

Admin: by the end of semester two USB-A cables are needed for data (one for programming, one for serial communication), one of them from about halfway through. On a personal laptop that may mean a **dongle** — there are workarounds but they all have downsides, so get one sooner rather than later.

Worked: what is the result of this program?

```
ldi r16, 0x56
ldi r17, 0x34
add r17, r16
```

`r17` will contain `0x8A`, `r16` will contain `0x56`.

`0x34` + `0x56` = `0x8A` (that is 52 + 86 = 138), and `add rd, rr` writes to **`rd`** — so `r17` takes the sum and **`r16` is left untouched**. An answer that ends with `r16 = 0x90` or `0x8A` has the destination backwards.

Worked: the status register after a compare

```
ser r17
ldi r18, 0xF0
cp r17, r18
```

Z = 0, N = 0, V = 0.

`ser r17` sets every bit, so `r17 = 0xFF`; `r18 = 0xF0`. Neither `ser` nor `ldi` touches a flag, so everything comes from the compare, which computes `0xFF` − `0xF0` = `0x0F` and throws the result away:

- **Z = 0** — the discarded result `0x0F` is not zero.

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

- **N = 0** — its MSB is 0.
- **V = 0** — as signed values this is $(-1) - (-16) = 15$, comfortably in range.

(C is also 0: no borrow, since `0xFF` $\geq$ `0xF0` unsigned.)

Boolean identities

These identities all hold, and together they are the toolkit the exercises need:

AND	OR	XOR
$X \cdot 0 = 0$	$X + 0 = X$	$X \oplus 0 = X$
$X \cdot 1 = X$	$X + 1 = 1$	$X \oplus X = 0$
$X \cdot X = X$	$X + X = X$	

Each operation has one value that **passes a bit through** and one that **forces it** — which is exactly what makes masking work. $X \oplus 1 = \overline{X}$ is the identity that makes XOR the toggling operation. Full treatment in bit-masking[5].

Bit masking

Sometimes only a few bits of a register matter. A bitwise AND can **mask out** the bits that are not of interest; masking can also selectively **set**, **clear**, **toggle** or **extract** bits. **Status register flags will be set also.**

Worked: which `r16` values make `andi r16, 0x01` set Z?

The even numbers.

`0x01` is `0000 0001`, so the AND clears every bit except bit 0 and the result is bit 0 alone. Z is set exactly when that result is zero — i.e. when **bit 0 of `r16` was 0**, which is precisely when `r16` is even. So a single masked AND is an even/odd test, and more generally any single-bit mask turns Z into a test of that one bit.

> **Provided task**
>
> **Exercises**
>
> Write an AVR assembly language program which performs the following operations on values `0x55` and 23_{10} and puts the result in the given register:

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

- Bitwise AND → `r16`
- Bitwise OR → `r5`
- Bitwise EOR → `r17`
- Addition → `r6`
- Difference (subtraction) → `r7`
- One's complement (inversion) of `0x55` → `r8`
- Two's complement (negation) of `0x55` → `r9`

Predict the results **and** status register values (Z, C, N, V) after each operation. Simulate the code in Microchip Studio and check your predictions.
Write AVR assembly language code snippets to:

- Copy the least significant three bits of register `r7` to register `r8`. Other bits of `r8` should be 0.
- Toggle the most significant four bits of register `r9`. Other bits should be unchanged.
- Clear the least significant four bits of register `r10`. Other bits should be unchanged.
- Set the most significant four bits of register `r11` to be 1. Other bits should be unchanged.

Test these code snippets.

Working — the arithmetic and logic program

The two operands:

$$\texttt{0x55} = \texttt{0101 0101} = 85 \qquad 23_{10} = \texttt{0001 0111} = \texttt{0x17}$$

The constraint that shapes the program: `ldi` only loads into r16–r31, but five of the seven destinations (`r5`–`r9`) are low registers. So the operands are loaded into a pair of high registers and **copied down** with `mov` before each operation. `mov` affects no flags, so each block's flags come only from its own operation.

The first destination, `r16`, is also a natural place to hold `0x55` — which would destroy it before the later operations run. Holding the operands in `r18`/`r19` instead avoids that.

```
    ldi  r18, 0x55        ; first operand
    ldi  r19, 23          ; second operand

    mov  r16, r18         ; --- bitwise AND -> r16
    and  r16, r19

    mov  r5, r18          ; --- bitwise OR -> r5
```

```
or   r5, r19

mov  r17, r18        ; --- bitwise EOR -> r17
eor  r17, r19

mov  r6, r18         ; --- addition -> r6
add  r6, r19

mov  r7, r18         ; --- difference -> r7
sub  r7, r19

mov  r8, r18         ; --- one's complement of 0x55 -> r8
com  r8

mov  r9, r18         ; --- two's complement of 0x55 -> r9
neg  r9
```

Predicted results and flags

Operation	Result	Binary	Z	C	N	V
`and r16, r19`	`0x15` (21)	`0001 0101`	0	*unchanged*	0	0
`or r5, r19`	`0x57` (87)	`0101 0111`	0	*unchanged*	0	0
`eor r17, r19`	`0x42` (66)	`0100 0010`	0	*unchanged*	0	0
`add r6, r19`	`0x6C` (108)	`0110 1100`	0	0	0	0
`sub r7, r19`	`0x3E` (62)	`0011 1110`	0	0	0	0
`com r8`	`0xAA` (170)	`1010 1010`	0	**1**	**1**	0
`neg r9`	`0xAB` (171)	`1010 1011`	0	**1**	**1**	0

Checking the interesting rows:

- **`add`** — $85 + 23 = 108$, no carry out of bit 7, and both operands positive giving a positive result, so V = 0.
- **`sub`** — $85 - 23 = 62$; $23 < 85$ so there is no borrow and C = 0.
- **`com`** — $\overline{\texttt{0101 0101}} = \texttt{1010 1010}$. N = 1 because the MSB is now set, and **C = 1 because `com` always sets the carry flag** — not a carry out of anything, just a defined convention.
- **`neg`** — $\overline{\texttt{0x55}} + 1 = \texttt{0xAA} + 1 = \texttt{0xAB}$, which is -85 read as two's complement. **C = 1 because `neg` sets carry unless the result is `$00`**, and V = 0 because V is only set when the result is exactly `$80`.

The logical operations (`and`, `or`, `eor`) all **clear V and leave C completely alone** — see avr-instructions[6]. That is why C reads *unchanged* rather than 0 for the first three rows: it holds whatever it held before the program ran.

Working — the bit manipulation snippets

All four are the same shape: build a mask, pick the instruction that does what you want to the masked bits. The choice follows straight from the identities above, and `andi`/`ori` cannot be used because every target here is a **low** register — so the mask goes into a high scratch register (`r16`) and the two-register form does the work.

Copy the least significant three bits of `r7` to `r8`, other bits of `r8` zero — *extract*, so AND with 1s on the bits to keep:

```
mov  r8, r7
ldi  r16, 0x07       ; 0000 0111
and  r8, r16
```

Toggle the most significant four bits of `r9` — *toggle*, so EOR with 1s on the bits to flip ($X \oplus 1 = \overline{X}$, $X \oplus 0 = X$):

```
ldi  r16, 0xF0       ; 1111 0000
eor  r9, r16
```

Clear the least significant four bits of `r10` — *clear*, so AND with **0s on the bits to clear** ($X \cdot 0 = 0$, $X \cdot 1 = X$):

```
ldi  r16, 0xF0       ; 1s mark what to KEEP
and  r10, r16
```

Set the most significant four bits of `r11` — *set*, so OR with 1s on the bits to set ($X + 1 = 1$, $X + 0 = X$):

```
ldi  r16, 0xF0
or   r11, r16
```

The thing worth noticing: **`0xF0` is the mask in three of the four, meaning three different things.** With `and` it names the bits that *survive*; with `or` and `eor` it names the bits that *change*. AND's mask is written for what you keep, the other two for what you touch — mixing that up is the standard bug.

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

See also

- bit-masking[7] · avr-instructions[8] · avr-architecture[9] · boolean-algebra[10]
- 2026-08-27-avr-introduction[11] — Lecture 10, the assembly language introduction
- instruction-set-architecture[12] — the Microchip Studio toolchain

[7] courses/csse2010/bit-masking.pdf
[8] courses/csse2010/avr-instructions.pdf
[9] courses/csse2010/avr-architecture.pdf
[10] courses/csse2010/boolean-algebra.pdf
[11] courses/csse2010/2026-08-27-avr-introduction.pdf
[12] courses/csse2010/instruction-set-architecture.pdf