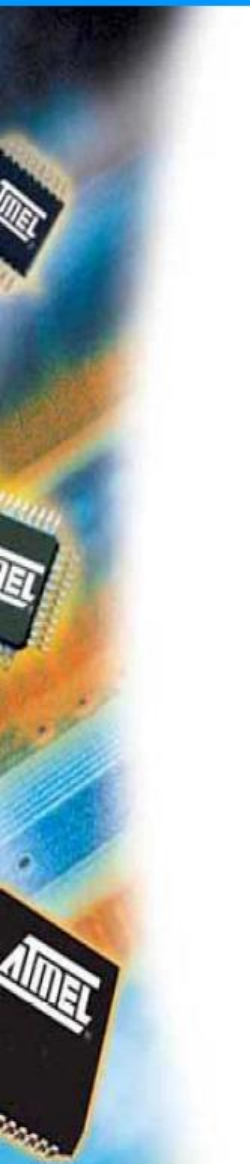
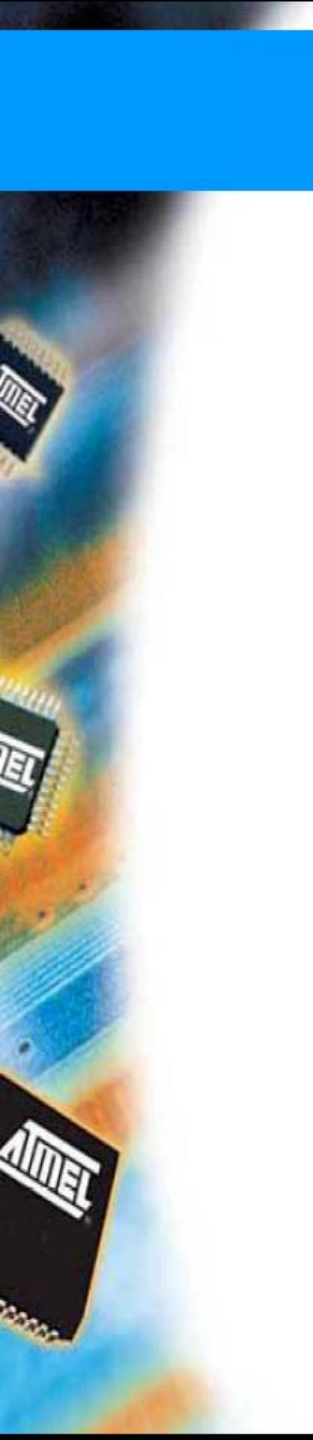
BME VIK – Measurement lab 2



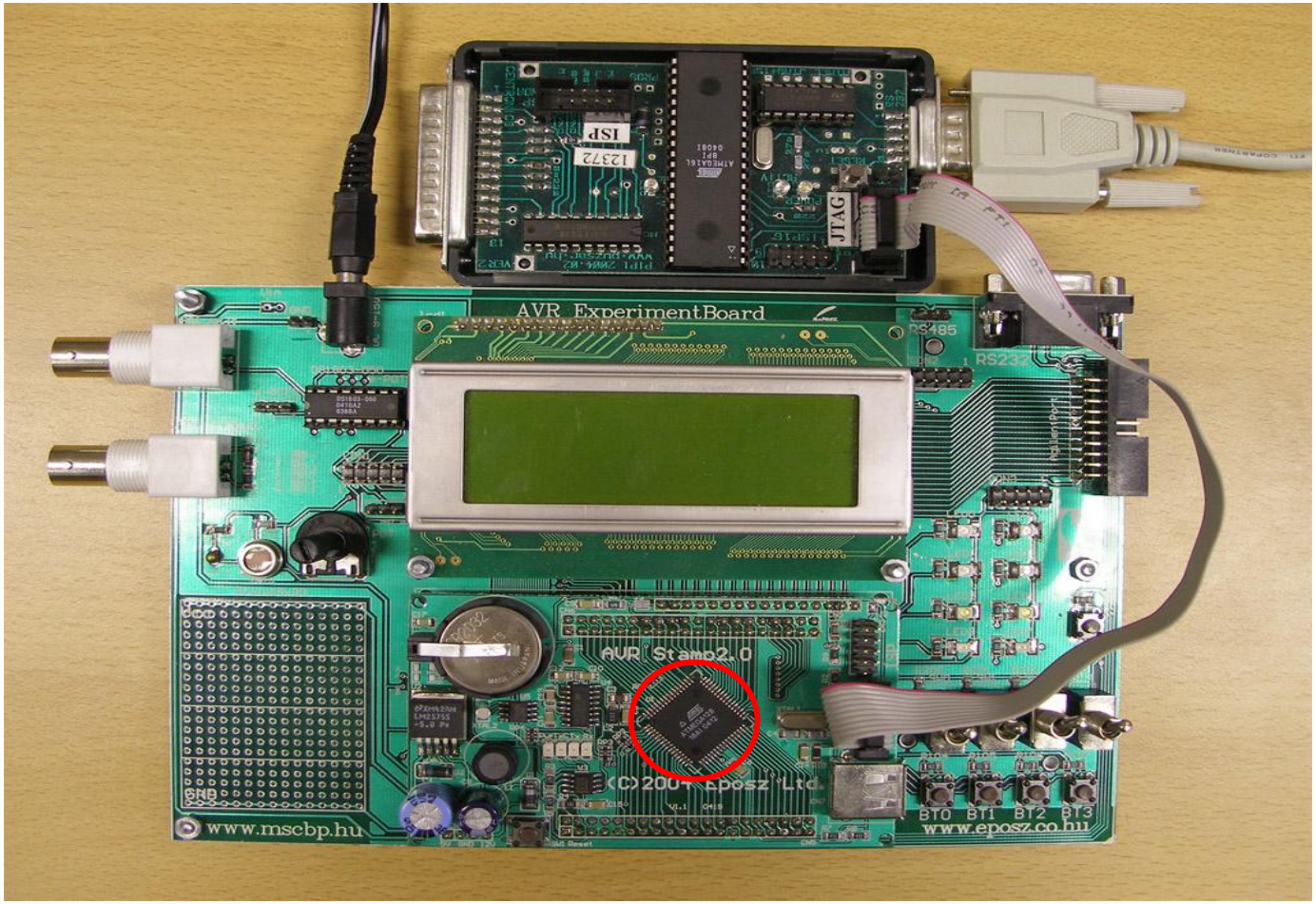
ATmega128 microcontroller

Topics



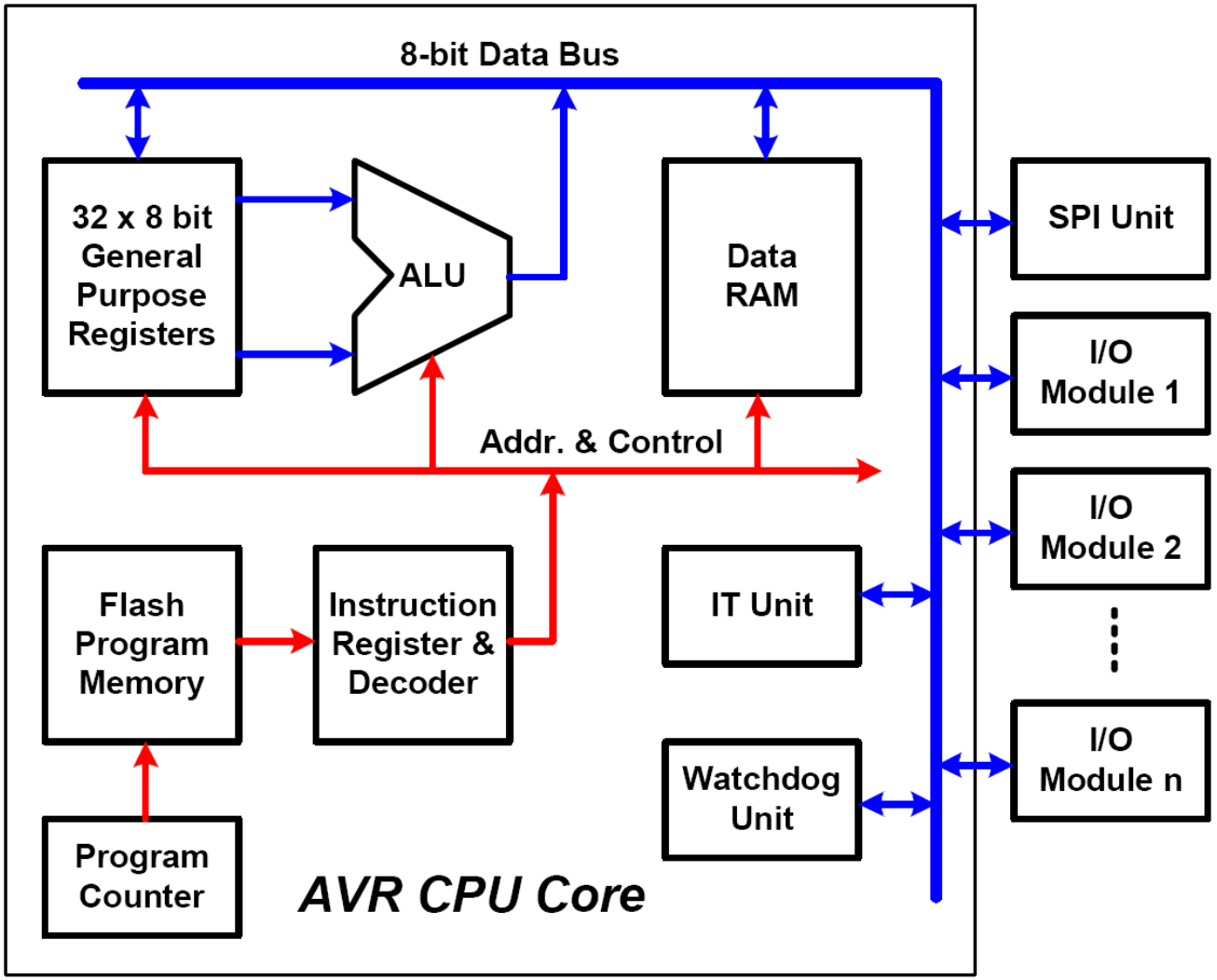
- **ATmega128 hardware**
- Assembly intro
- I/O ports
- Development environment

ATmega128 development board



Description: avr_experimentboard_v103.pdf

AVR block diagram

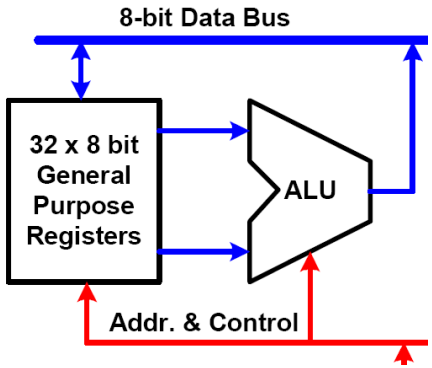


ATmega128 hardware

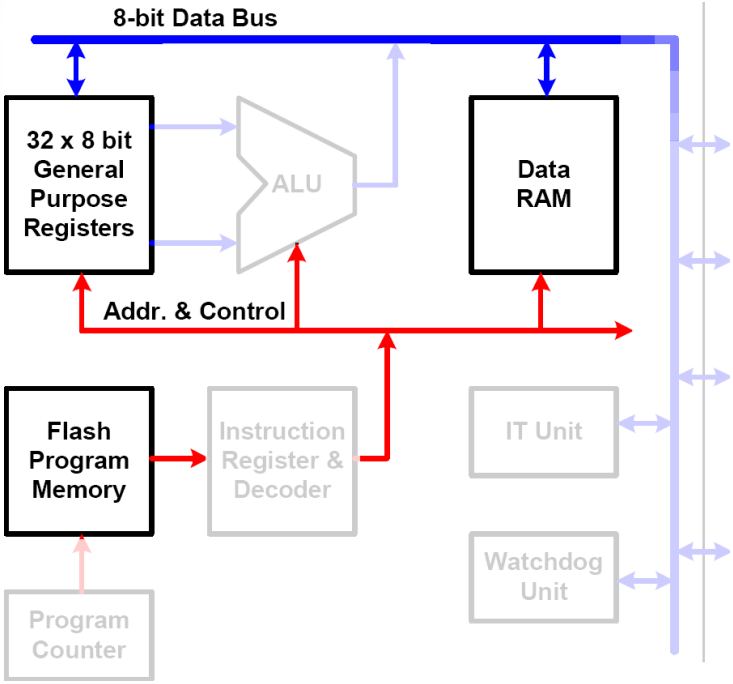
CPU:

- 8 bit registers, 16 (and 32) bit wide instruction codes
  → **8 bit uC**
- 134 RISC instruction
- clock ≤ 16 MHz (our @ 11,0592 MHz)
- 1 instruction/cycle (except: branch, jump etc)
- System on a Chip (SoC): internal memory, integrated peripherals

- RISC: **load/store architecture**
  - no direct operations on memory
    = load to register → calc → write to mem
  - Registers directly connected to ALU, operation *source* and *target* are regs
    (one of them may be constant value as well)



ATmega128 hardware



Memory:

- **128K Flash (program)**
- **4K internal SRAM (data)**

Harvard-architecture

- 4K EEPROM (nonvolatile)

Program memory:

word addressing + 16 bit addresses
= 2^{16} * 2 byte = 128KB

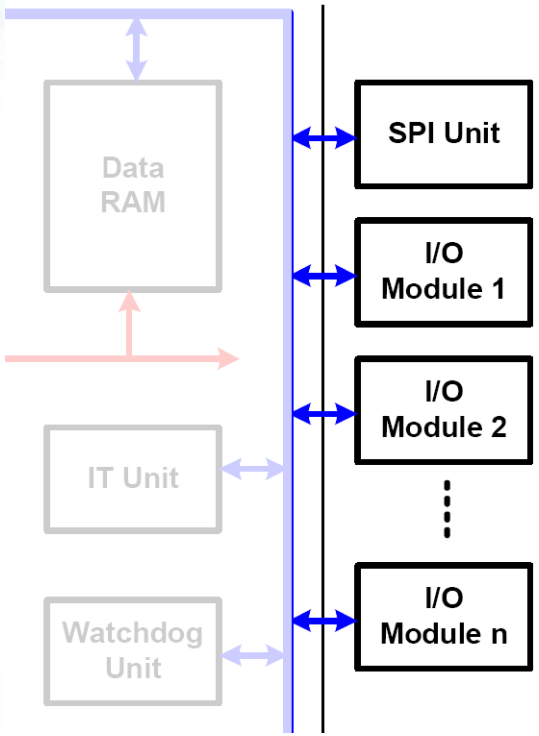
Data memory:

- 32 generic registers
- I/O registers
  - normal + extended
  - extended I/O: embedded into data memory address range, like SRAM

Data memory map

$0000 -	32 register
$0020 -	64 I/O reg.
$0060 -	160 ext. I/O reg.
$0100 -	Int. SRAM (4096 x 8)
RAMEND = $10FF -	(Ext. SRAM)

ATmega128 hardware



ATmega peripherals

- Digital I/O ports
- HW timers
- A/D converter, SPI, UART…

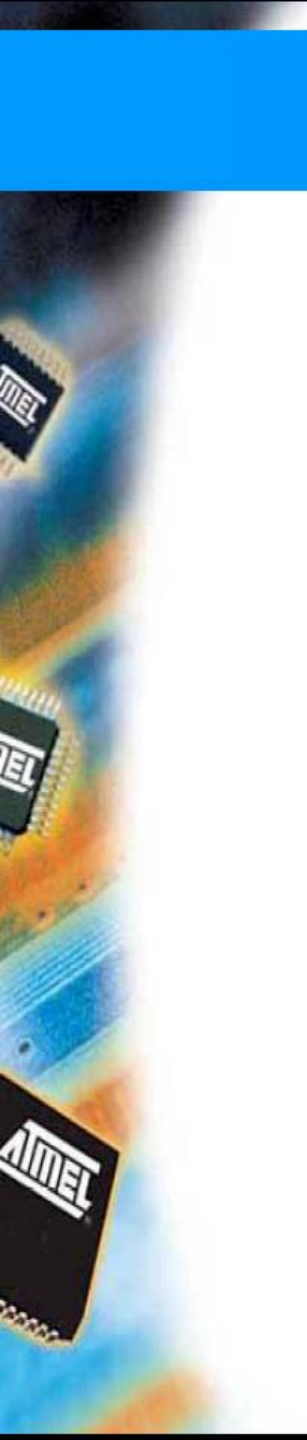
Dev. board:

- LED, button, switch
- potentiometer, photoresistor, temperature sensor
- 4x20 char LCD…

How to handle:

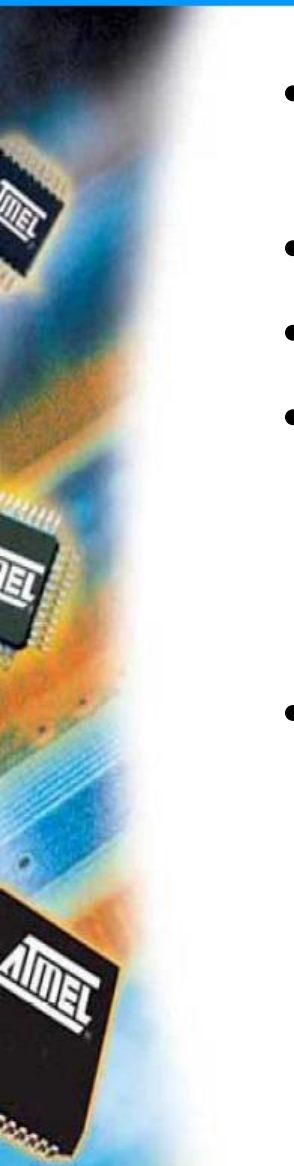
- Via I/O registers (later)

Topics



- ATmega128 hardware
- ➢ **Assembly intro**
- I/O ports
- Development environment

Assembly intro

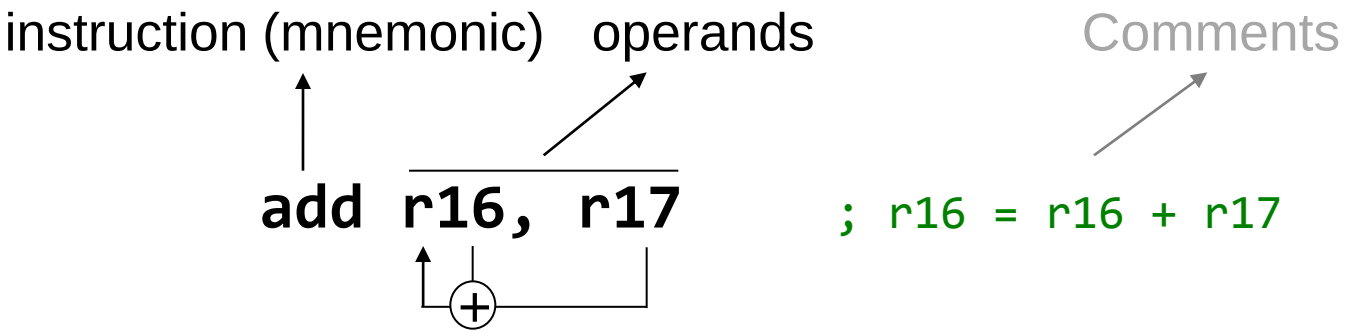


- Low level programming
  - No optimizations, directly to machine code
- Architecture-dependent (x86, PowerPC, AVR, ARM…)
- Machine code < assembly < C
- Where
  - Code size: small embedded systems
  - Speed: video codec, gfx, math apps…
  - No C sources: reverse engineering (e.g. keygens)
- *Not* in enterprise applications
  - Development cost
  - C compilers are quite good at optimization

Assembly intro

- Simple instructions, simple logic
  - Feel the power of the hardware ☺
- Low level: one thing per instruction
  - C: a = b * 5 + 24
  - AVR assembly:
    - Load b from memory to register1
    - Load 5 to another register2
    - Multiply them – results are put in other registers
    - Load 24 to a register3
    - Add multiplication results to register3
    - Store register3 to memory address a

AVR assembly - instructions



Mnemo.	Operands	Description	Operation	Flags	#Clk
ADD	Rd, Rr	Add without Carry	Rd ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry	Rd ← Rd + Rr + C	Z,C,N,V,H	1
ADIW	Rd, K	Add Immediate to Word	Rd+1:Rd ← Rd+1:Rd + K	Z,C,N,V	2
SUB	Rd, Rr	Subtract without Carry	Rd ← Rd - Rr	Z,C,N,V,H	1

- Case insensitive
- Number formats (same 8 bit value):

```
66, 0x42, $42, 0b0100_0010, 'B', -190
```

AVR assembly - registers

General purpose registers: **R0...R31**

- 32 x 8 bit "always at hand"
- like C variables: most operations are done on them
- **only R16-R31 for constants** (**LDI**, **ANDI**, **SUBI**...)

```
ldi r16, 0b10110011 ; R16 = 0xB3
ldi r7,  123        ;"Invalid reg."
mov r7,  r16        ; R7 = R16
```

Mnemo.	Operands	Description	Operation
MOV	Rd, Rr	Copy Register	Rd ← Rr
LDI	Rd, K	Load Immediate	Rd ← K

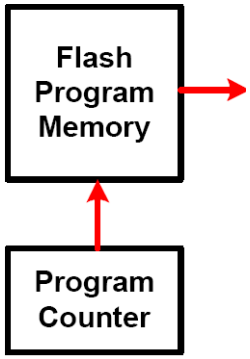
Special:

- **PC** – program counter
- **SREG** – status register
- **SP** – stack pointer

AVR assembly – special registers

PC: Program Counter

- Current instruction's prog. memory address (16 bit)
- Jumps, branches directly modify its value
- **Use labels**, let the assembler calculate the *absolute or relative* address („k"):

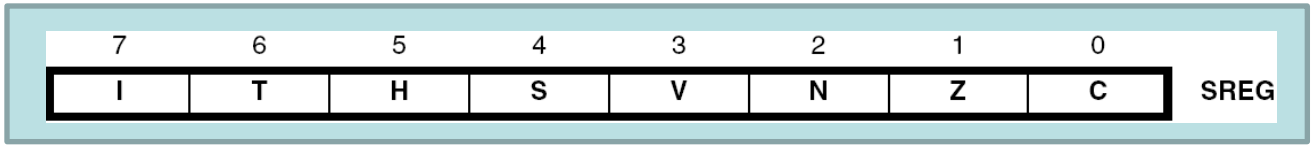


JMP	k	Jump	PC ← k	None	3
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1/2

```
M_LOOP:                 ; main loop label: addr of inc r16
    inc r16             ; r16 = r16 + 1
    out PORTC, r16      ; write to LEDs
    jmp M_LOOP          ; PC ← M_LOOP (jump to inc r16)
```

AVR assembly – special registers

SREG: Status register – flags („status bits”)



- I: global IT enable, … N: Negative, Z: Zero, C: Carry
- **Conditional brances** are based on **SREG** bits
- ALU ops automatically modify **SREG** bits

Mnemo.	Operands	Description	Operation	Flags
SEI		Global Interrupt Enable	I ← 1	I
CLC		Clear Carry	C ← 0	C
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V
SUBI	Rd, K	Subtract Immediate	Rd ← Rd - K	Z,C,N,V,H

Instruction	R16	N	Z	C
`ldi r16, 254`	0xFE	0	0	0
`inc r16`	0xFF	1	0	0
`inc r16`	0x00	0	1	0
`inc r16`	0x01	0	0	0
`subi r16, 3`	0xFE	1	0	1

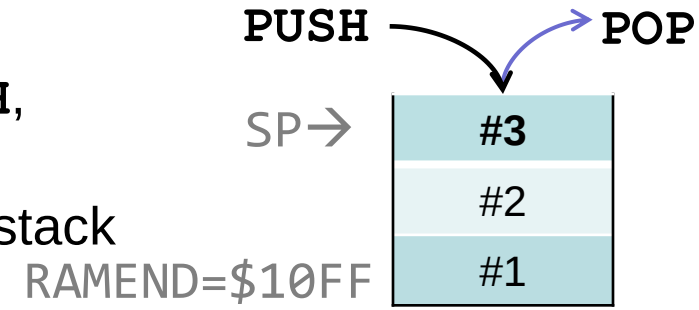
N: 7th bit == 1
Z: all bits == 0

C: carry

AVR assembly – special registers

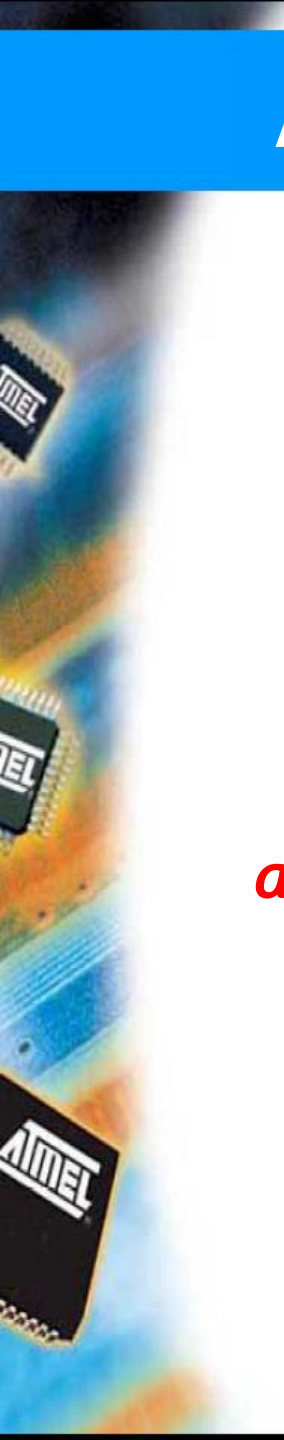
SP: Stack pointer

- Stack: LIFO mem at the end of internal SRAM
- return address of subroutines and temporarily save register values
- Save register content to stack: **`PUSH`**, load: **`POP`**
- **`SP`**: 16 bit register pointing to top of stack
  - **`PUSH`** decreases **`SP`**



- **`CALL`** (subroutines: ~function calls) pushes **`PC`** to stack; **`RET`** pops **`PC`** at the end of the routine → jump back to caller

AVR assembly – instruction types



- Arithmetic and logic
- Branch and jump
- Data manipulation
- Bit set/test

avr128_prog.pdf *+ avr_instr_set.pdf*

+ F1 (context sensitive help)

Arithmetic instructions

- **`INC/DEC, ADD/ADC, SUB(I), NEG, MUL…`**
  - No division (div with 2: right shift)

SREG!

DEC	Rd	Decrement	Rd ← Rd - 1	Z,N,V	1
ADD	Rd, Rr	Add without Carry	Rd ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry	Rd ← Rd + Rr + C	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Immediate	Rd ← Rd - K	Z,C,N,V,H	1

Utasítás	R16	R17	N	Z	C
`ldi r16, 0`	0	?	1	0	1
`ldi r17, 100`	0	100	1	0	1
`add r16, r17`	100	100	0	0	0
`add r16, r17`	200	100	1	0	0
`add r16, r17`	44	100	0	0	1
`adc r16, r17`	145	100	1	0	0
`subi r16, 144`	1	100	0	0	0
`dec r16`	0	100	0	1	0

(+ Carry)

Logic / bit manipulations

- Logic: **AND(I), OR(I), EOR, COM,** CLR/SER, CBR/SBR…

AND	Rd, Rr	Logical AND	Rd ← Rd • Rr	Z,N,V
ANDI	Rd, K	Logical AND with Immediate	Rd ← Rd • K	Z,N,V
ORI	Rd, K	Logical OR with Immediate	Rd ← Rd v K	Z,N,V
EOR	Rd, Rr	Exclusive OR	Rd ← Rd ⊕Rr	Z,N,V
COM	Rd	One's Complement [invertálás]	Rd ← $FF - Rd	Z,C,N,V

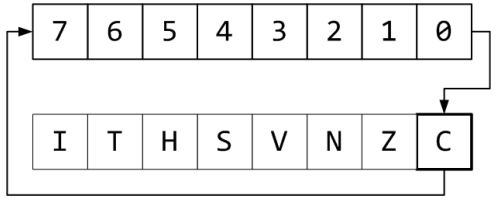
⇩

EOR/LDI,
ANDI/ORI

- Bit manipulations: **LSL, LSR, ROL, ROR, SEI, CLC…**

LSL	Rd	Logical Shift Left	Rd(n+1) ← Rd(n), Rd(0) ← 0,C ← Rd(7)	Z,C,N,V,H	1
LSR	Rd	Logical Shift Right	Rd(n) ← Rd(n+1), Rd(7) ← 0,C ← Rd(0)	Z,C,N,V	1
ROL	Rd	Rotate Left Through Carry	Rd(0) ← C,Rd(n+1) ← Rd(n),C ← Rd(7)	Z,C,N,V,H	1
ROR	Rd	Rotate Right Through Carry	Rd(7) ← C,Rd(n) ← Rd(n+1),C ← Rd(0)	Z,C,N,V	1

ROR:



0	0	0	1	0	0	0	1
0	0	0	0	1	0	0	0
1	0	0	0	0	1	0	0

```
ldi   r16, 0b11001010 ; r16 = 0b11001010
andi  r16, 0b00001111 ; = 0b00001010
ori   r16, 0b00110000 ; = 0b00111010
com   r16             ; = 0b11000101
lsl   r16             ; = 0b10001010
eor   r16, r16        ; = 0b00000000
```

Jumps and conditional branches

- **JMP**: jump

Infinite loop	C alternative	

`M_LOOP:`
`  ...`
`  jmp M_LOOP`
	`while (1) {`
	`  ...`
	`}`
		`loop:`
		`  ...`
		`  goto loop;`

- **CALL, RET**: subroutine call and return, PC→stack

Subroutine	Function

`M_LOOP: …`
`  call FV`
`  …`
`FV:  …instructions…`
`  ret`
	`void main () {…`
	`  fv();`
	`}`
	`void fv() { …instructions…`
	`  return; }`

  - *Subtasks should be in subroutines if possible!*

Jumps and conditional branches

- **`BR*`** (branch): based on SREG bits

BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None	1/2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1/2
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None	1/2
BRCC	K	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None	1/2
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None	1/2
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None	1/2
…	…	…	…	…	…

„for" loop	for loop

`    ldi r16, 100`
`LOOP:`
`    <instruction1>`
`    dec r16`
`    brne LOOP`
`    <instruction2>`
	`for (uint8_t i=100; i>0; i--) {`
	`    <instruction1>`
	`}`
	`<instruction2>`

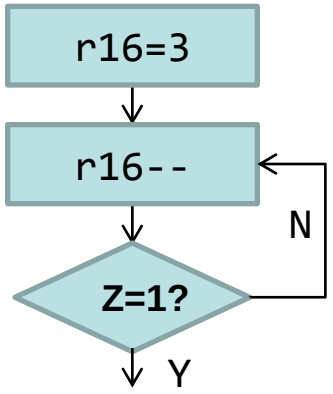
Jumps and conditional branches

„for” loop with BRNE

DEC	Rd	Decrement	Rd ← Rd - 1	Z,N,V	1
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1/2

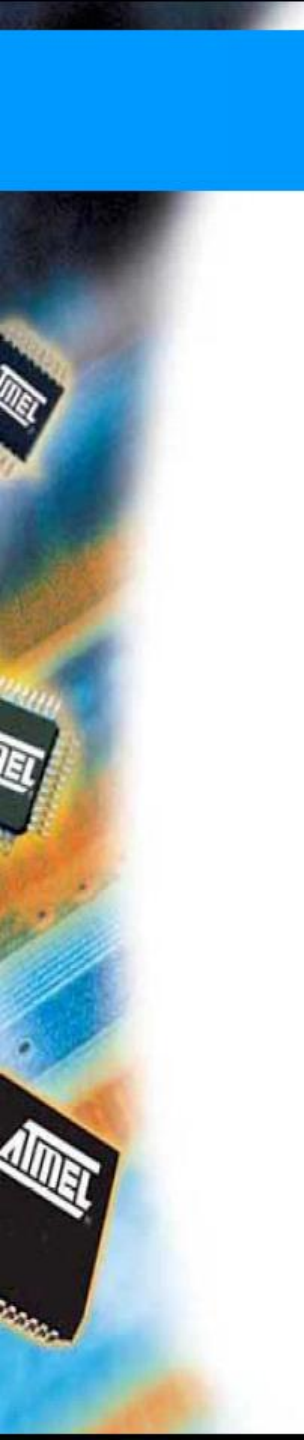
„for” loop

```
     ldi r16, 3
LOOP:
     dec r16
     brne LOOP
…
```



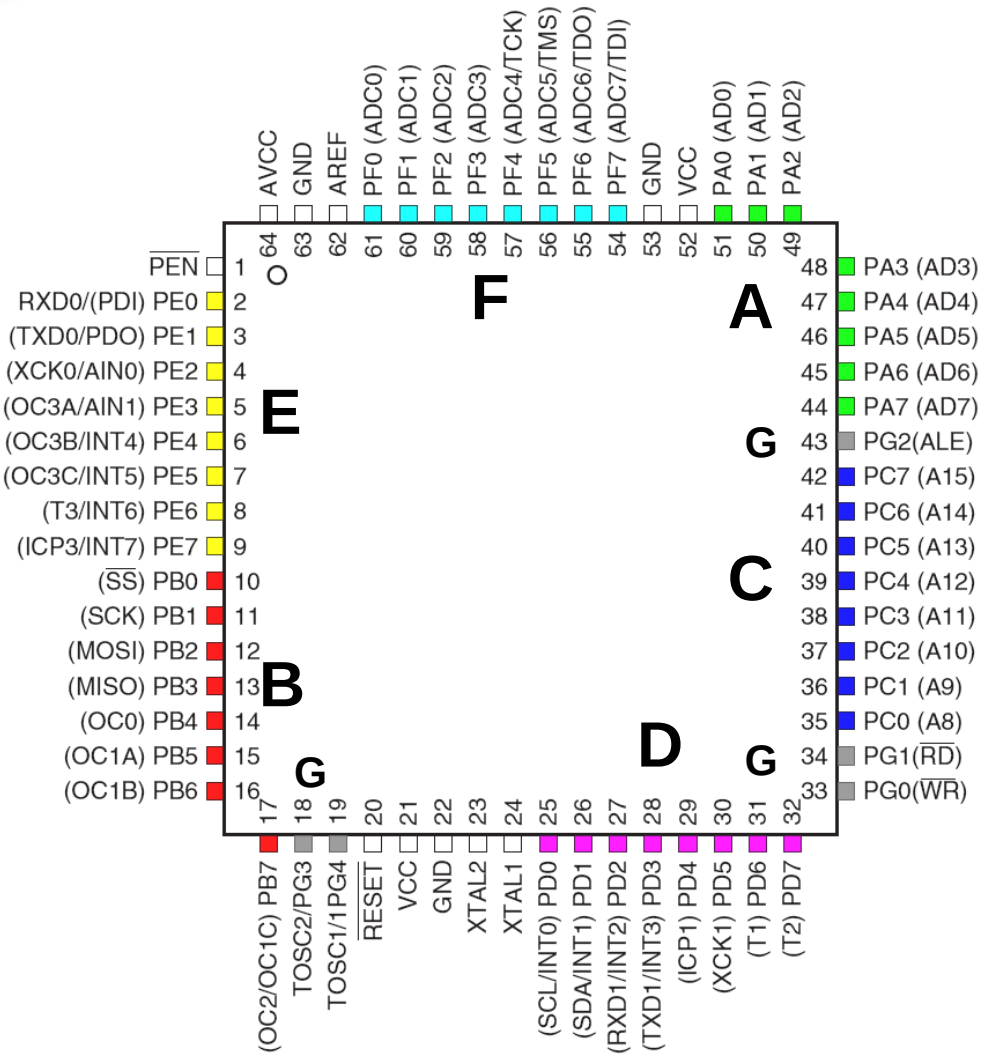
Instruction	R16	N	Z	C	BRNE
ldi r16, 3	3	0	0	0	-
dec r16	2	0	0	0	-
brne LOOP	2	0	0	0	jump
dec r16	1	0	0	0	-
brne LOOP	1	0	0	0	jump
dec r16	0	0	1	0	
brne LOOP	0	0	1	0	step
…					

Topics



- ATmega128 hardware
- Assembly intro
- ➢ **I/O ports**
- Development environment

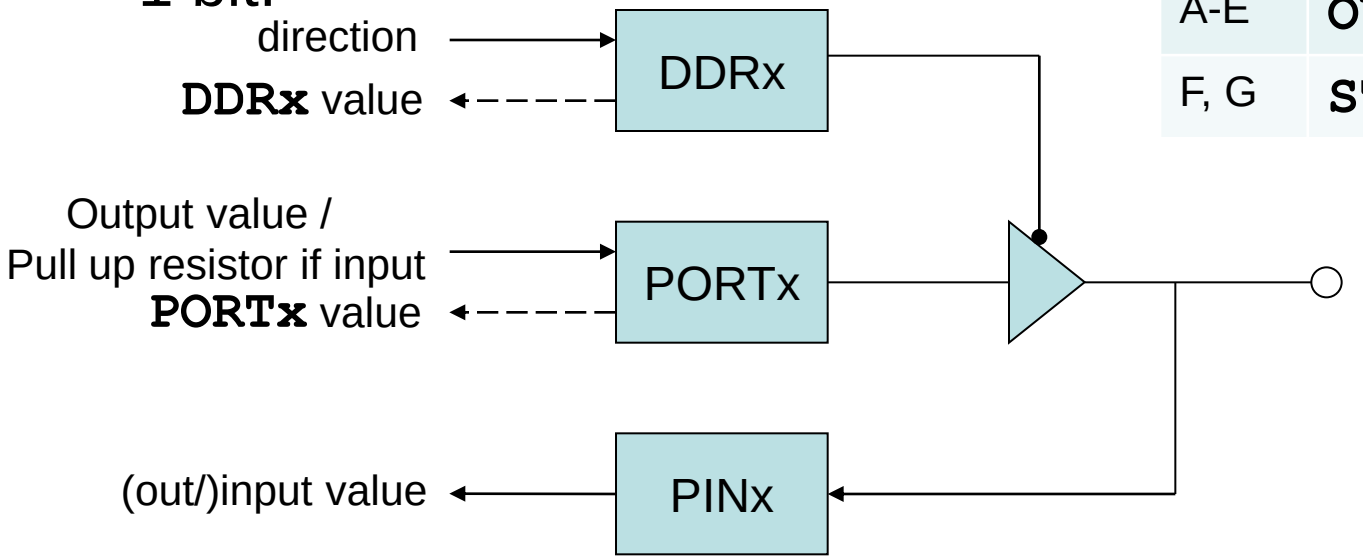
ATmega128 I/O pins



- I/O ports: 6x8 + 1x5 pins together
- Input/output adjustable per each pin
- 3 I/O registers / port
- F, G: extended I/O → mem

I/O ports

- 8 pins = 8 bits = 1 port
- 1 bit:



Port	DDRx PORTx	PINx
A-E	**OUT**	**IN**
F, G	**STS**	**LDS**

I/O	DDRx (data direction)	PORTx (data out)	PINx (data in)
IN	**0**	0: Hi-Z, 1: pull-up res.	**Input value**
OUT	**1**	**Output value**	PORTx value

I/O ports - peripherals

port	7	6	5	4	3	2	1	0
B	BT3							
C	LED7	LED6	LED5	LED4	LED3	LED2	LED1	LED0
E	BT2	BT1	BT0	INT				
G				SW2	SW3		SW1	SW0

- See header of assembly templates
- LED: port C
- Buttons: port E + B
  - Active low, released 1, pressed 0
- Switch: port G

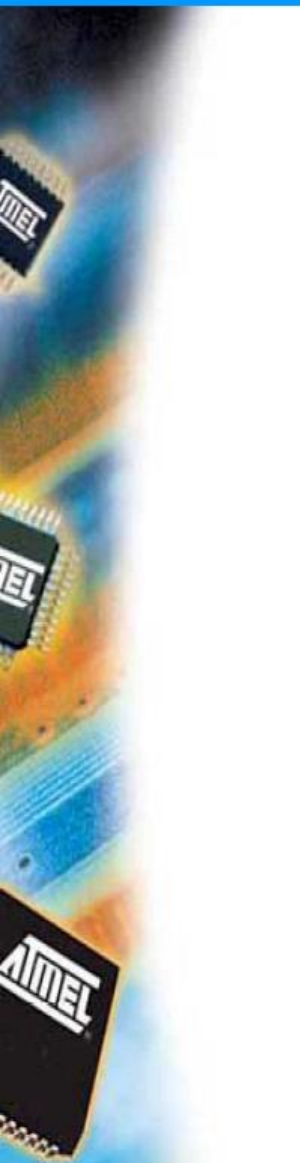
I/O ports - peripherals

Buttons to LEDs

IN	Rd, P	In Port	Rd ← P	None	1
OUT	P, Rr	Out Port	P ← Rr	None	1

```
.def temp = r16          ; define: r16 as „temp”
.def btn  = r17
    ldi  temp, 0xFF      ; all 8 LEDs are output
    out  DDRC, temp      ; write to Data Direction Reg
    ldi  temp, 0x00      ; buttons: in
    out  DDRE, temp      ;
M_LOOP:
    in   btn, PINE       ; read buttons
    com  btn             ; invert
    andi btn, 0xF0       ; mask: only bits for BT2…INT (7-4)
    out  PORTC, btn      ; write to leds
    jmp  M_LOOP
```

Topics



- ATmega128 hardware
- Assembly intro
- I/O ports
- **Development environment**

AVR Studio

- Assembly or C (C: AVR-GCC, WinAVR)
- Direct programming in hardware (ISP, In System Programming) and debug (JTAG-ICE, In Circuit Emulation)
- Good simulation environment (uC + integrated peripherals) – possible to test program without real hw

Test your code at home!