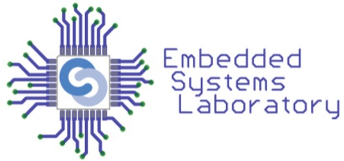
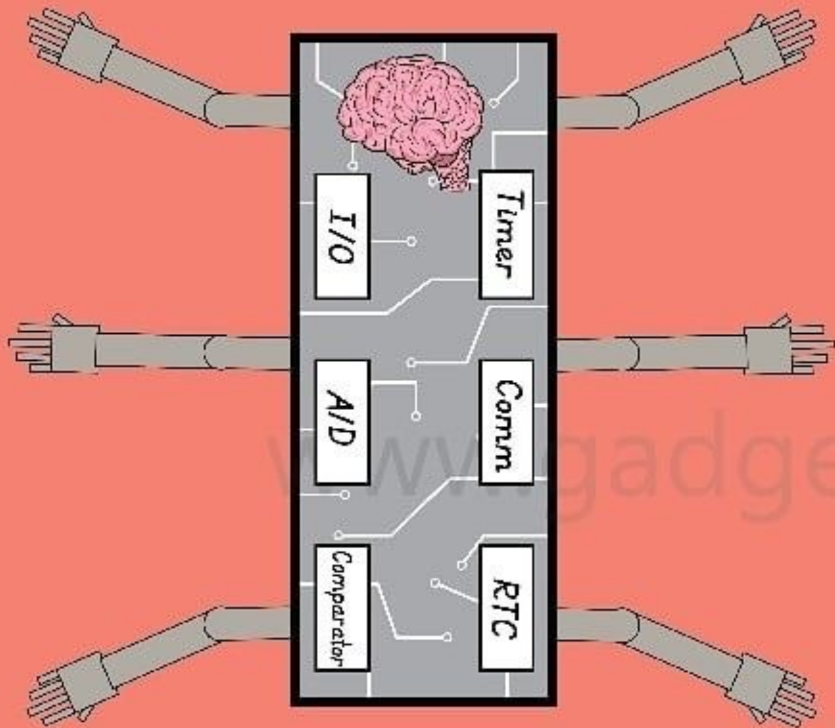
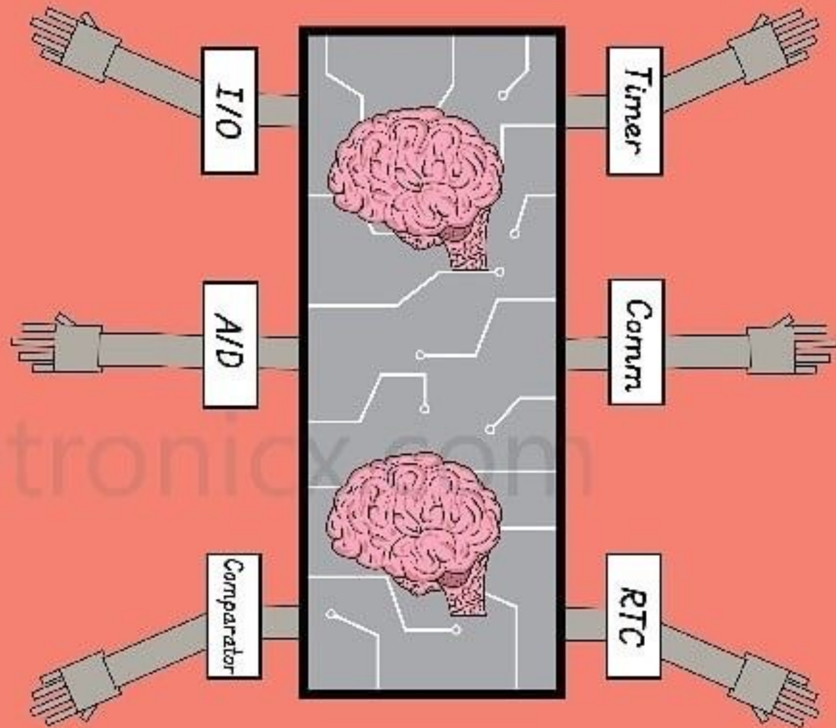


# PROIECTAREA CU MICROPROCESOARE

## Cursul 1
## Microprocesoare și Microcontrollere, GPIO

Facultatea de Automatică și Calculatoare
Politehnica București




www.gadgetronicx.com

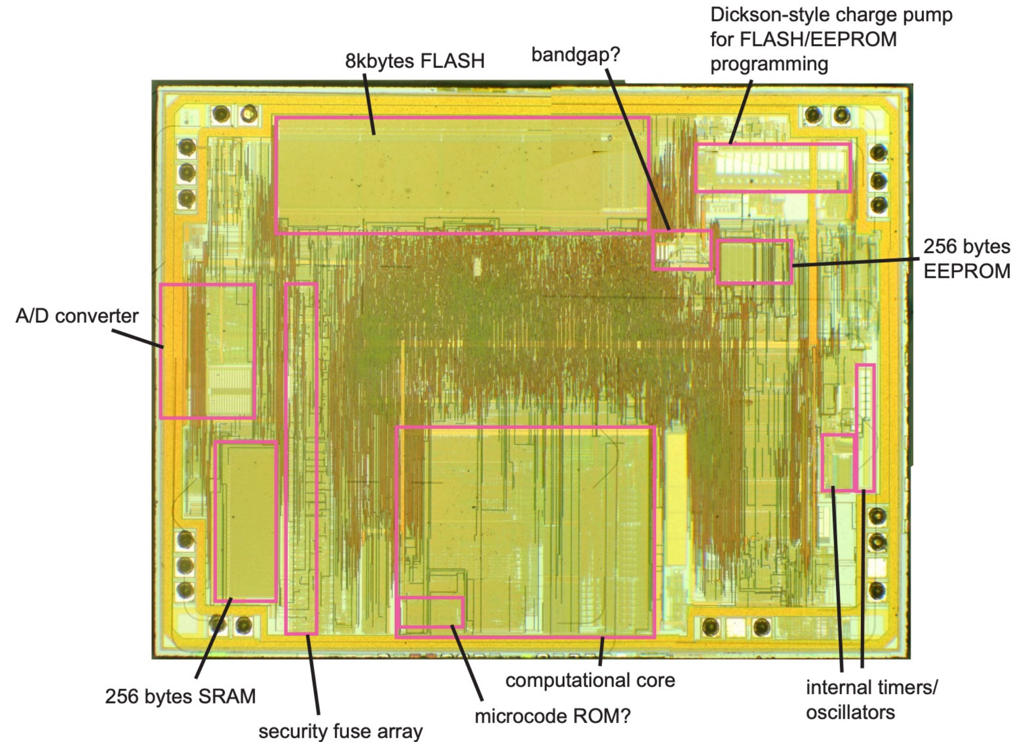
# Ce este un microcontroller?

Un microcontroler (MCU) este un calculator într-un un singur circuit integrat

- Unitate centrală de procesare (CPU) relativ simplă
- Dispozitive periferice precum memorii, dispozitive de intrare/ieșire și timere

Potrivit unor estimări, mai mult de jumătate din toate procesoarele vândute în întreaga lume sunt microcontrolere

# Pastila de siliciu a unui microcontroller




https://www.bunniestudios.com/blog/?page_id=40

# Microcontroler VS Microprocesor

**Microcontroler**

Un microcontroler este un mic computer pe un singur circuit integrat (CI).

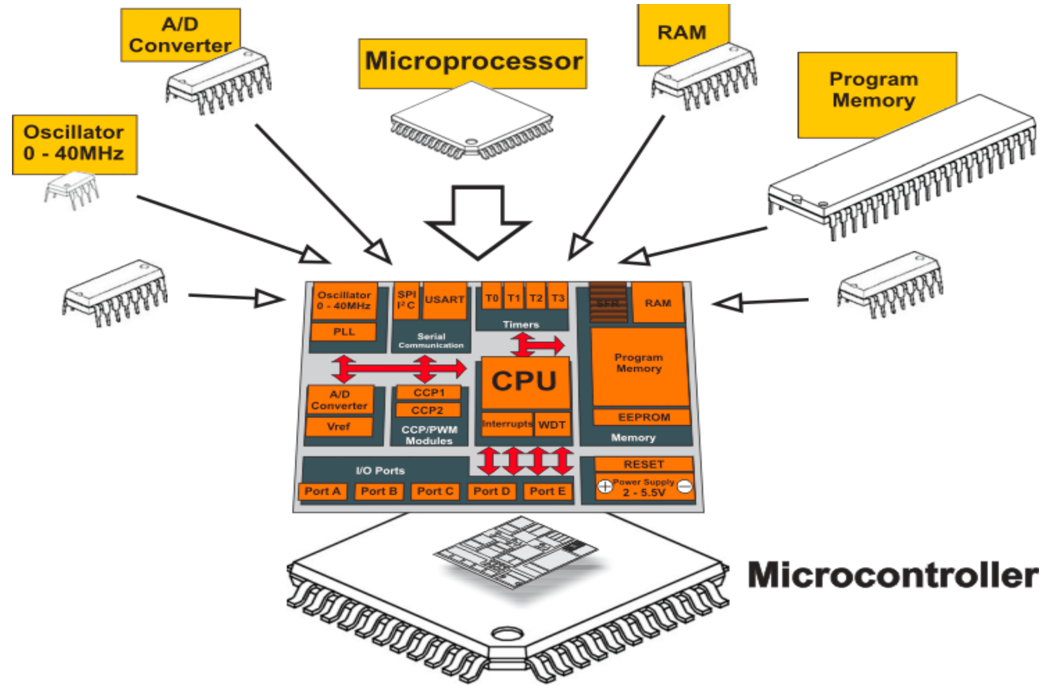
**Microprocesor**

Un microprocesor este un procesor central de calculator (CPU) pe un singur circuit integrat (CI).

# Microcontroler VS Microprocesor

| Caracteristică | Microcontroler | Microprocesor |
| --- | --- | --- |
| Funcționalitate | Include CPU, memorie și I/O | Include doar CPU |
| Cost | Mai ieftin | Mai scump |
| Complexitate | Mai simplu | Mai complex |
| Utilizare | Dispozitive încorporate | Calculatoare personale, servere etc. |

# Microcontroler VS Microprocesor




https://www.electronicsforu.com/resources/difference-between-microprocessor-and-microcontroller

# Tipuri de procesoare

## În general-purpose computing:

• Diversitatea arhitecturilor seturilor de instrucțiuni este astăzi limitată, arhitectura Intel x86 dominând covârșitor pe toate.

•Cu toate acestea, x86 începe să aibă competiție din partea ARM și a altor jucători mai mici (RISC-V, de ex.)

## În embedded computing:

• Nu există un asemenea monopol.

• Dimpotrivă, diversitatea procesoarelor poate fi descurajatoare pentru un proiectant de sisteme.

## Lucruri care contează:

• Periferice

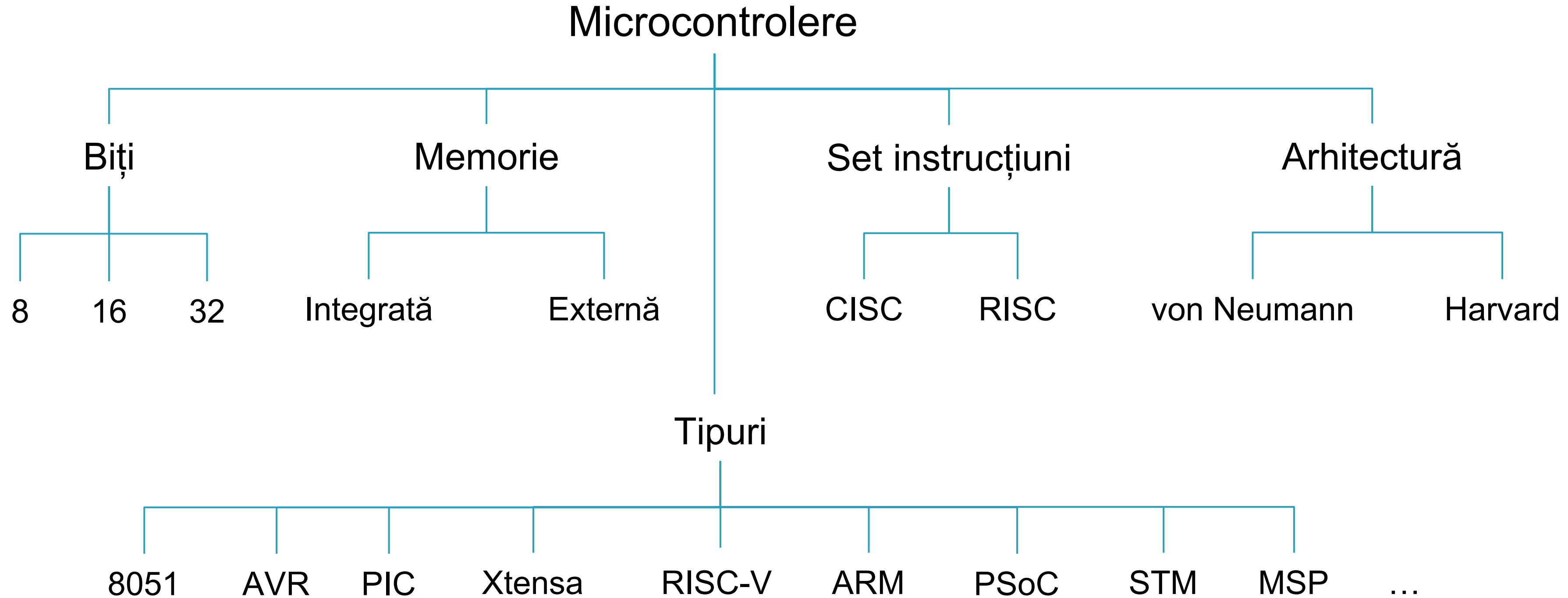
• Concurență și sincronizare

• Frecvența de ceas

• Dimensiuni memorie (SRAM și Flash)

• Dimensiuni pachete de date

# Tipuri de microcontrolere



# Cum alegi procesorul potrivit pentru proiectul tău?

Care sunt metricile principale la care ar trebui să te uiți:

- Consumul de energie
- Frecvența de operare
- Pini IO
- Memorie
- Funcții interne
- **Software availability & support!**

# De exemplu:

- Consumul de energie
  - Curentul maxim consumat de dispozitiv trebuie să nu depășească 2.5mA
- Frecvența de operare
  - Dispozitivul trebuie să gestioneze o interfață wireless de 1Mbps
  - kHz e prea lent
  - GHz este overkill
- Pini IO
  - Dispozitivul are un senzor de imagine, display, ADC, senzori pe I2C, card SD, butoane și LEDuri care necesită un total de 42 de pini de date

# De exemplu:

- Memorie
  - Trebuie să avem suficientă memorie ca să stocăm
    - Programul (ne-volatil)
    - Datele: Constante (parametri calibrare, valori implicite),  variable de program
- Funcții interne
  - Transferul de date din memorie în display (DMA)

# De exemplu:

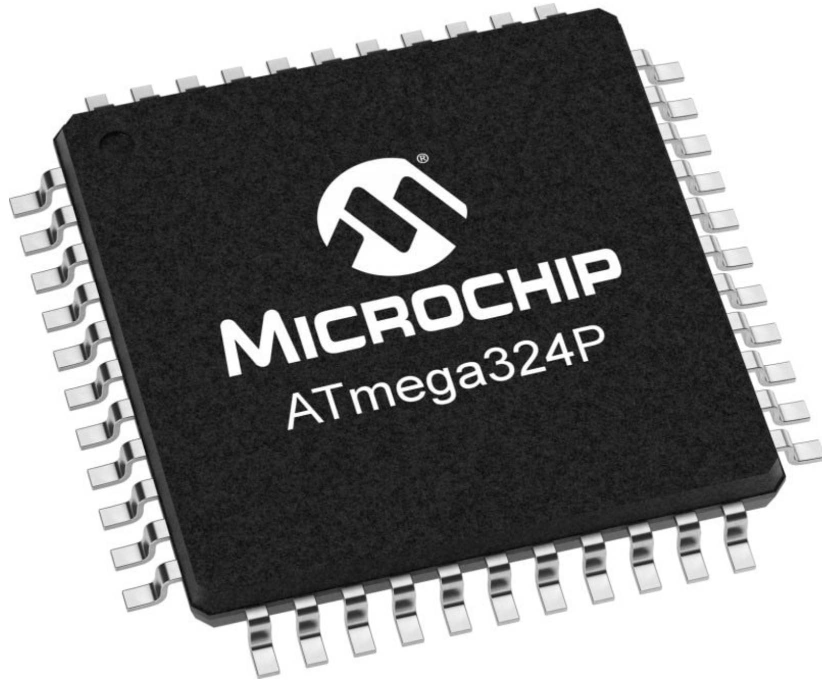
Vrei să stochezi datele de la un accelerometru

- Câte 12 biți pentru fiecare valoare a accelerației pe X, Y și Z
- Eșantionare de 2000 ori pe secundă (2kHz) = 12 x 3 x 2000 biți pe secundă (72 kb sau 9 kB)
- Dacă avem RAM intern de 100kB – se va umple în ceva mai mult de 10 secunde!

Alternativă: stocare externă

- SRAM – scump, capacitate redusă
- Flash – ieftin, dar consumă multă energie (de ex., o citire ~5mA și o scriere ~10mA)

# Microcontrolerul de la laborator




https://www.microchip.com/en-us/product/ATmega324P

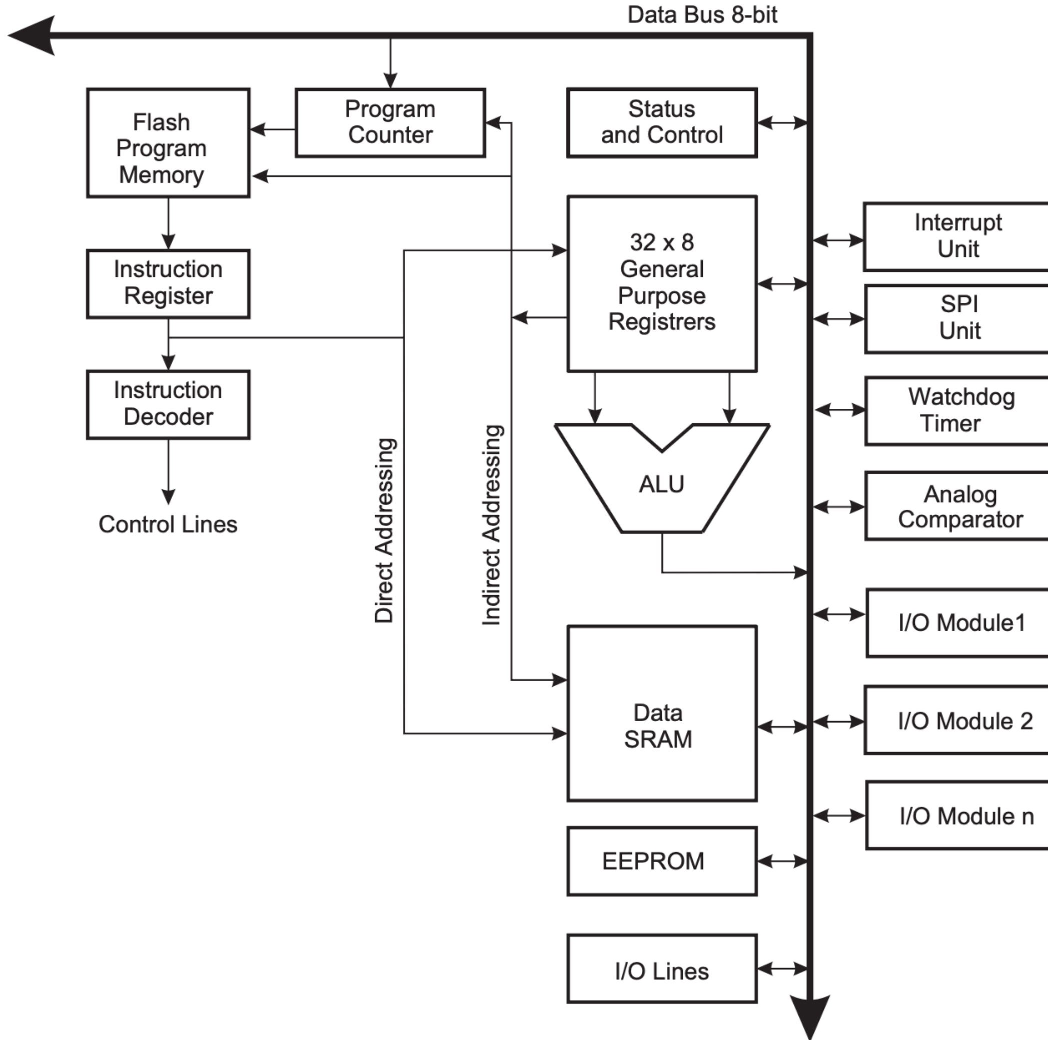
## Feature

High Performance, Low Power Atmel® AVR® 8-Bit Microcontroller Family

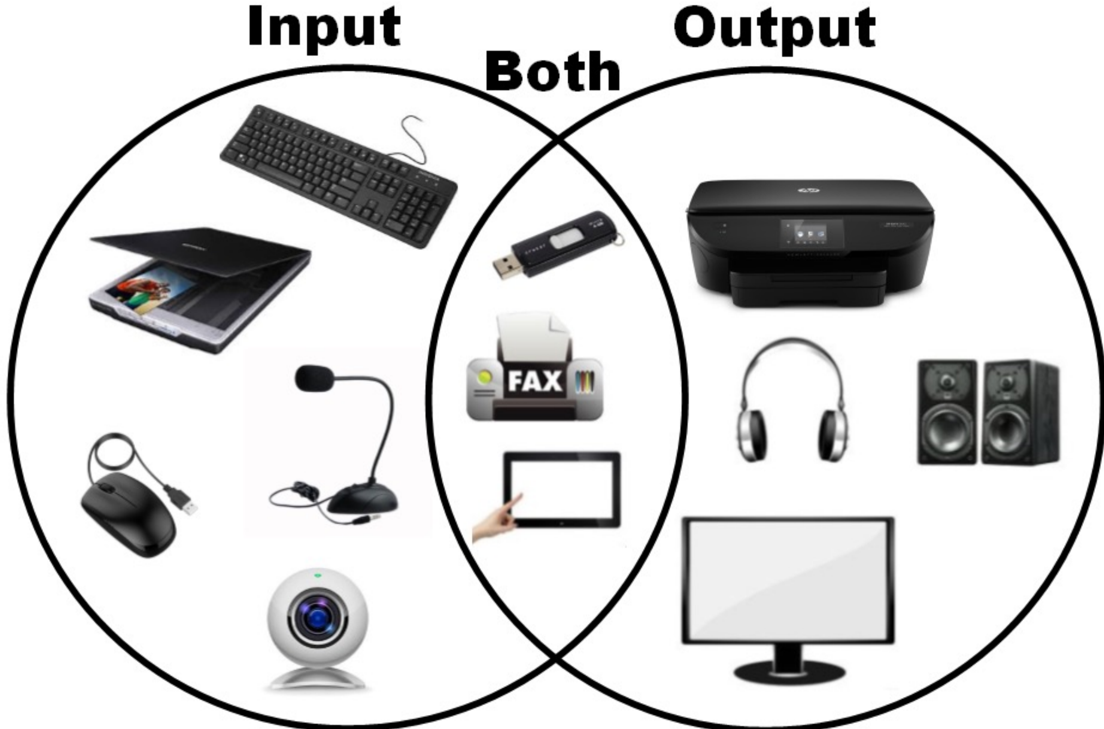
- Advanced RISC Architecture
  - 131 Powerful Instructions
  - Most Single Clock Cycle Execution
  - 32 x 8 General Purpose Working Registers
  - Fully Static Operation
  - Up to 20 MIPS Throughput at 20MHz
  - On-chip 2-cycle Multiplier
- High Endurance Non-volatile Memory Segments
  - 32KBytes of In-System Self-Programmable Flash Program Memory
  - 1KBytes EEPROM
  - 2KBytes Internal SRAM
  - Write/Erase Cycles: 10,000 Flash/100,000 EEPROM
  - Data Retention: 20 Years at 85°C/100 Years at 25°C(1)
  - Optional Boot Code Section with Independent Lock Bits
    - In-System Programming by On-chip Boot Program
    - True Read-While-Write Operation
  - Programming Lock for Software Security
- Atmel QTouch® Library Support
  - Capacitive Touch Buttons, Sliders and Wheels
  - QTouch and QMatrix acquisition

# AVR CPU Core

- Procesor RISC construit în jurul unei magistrale de 8 biți
- 32 registre generale
- Bandă de asamblare cu 2 etape
- Single-cycle ALU (integers only)
- Majoritatatea instrucțiunilor executate în 1 ciclu de ceas (1 MIPS)



# Input & Output (I/O)



# Dispozitive I/O

- Tastatură, mouse, microfon, scanner, camera foto/video etc.
  - Diversitate foarte mare
  - Multe tipuri de dispozitive, capabilități foarte diferite
  - Chiar și dispozitivele de același tip pot să difere semnificativ
- Viteză
  - Variază de la un dispozitiv la altul, în general mai scăzută decât CPU
  - Unele dispositive au nevoie de timpi foarte rapizi pentru acces și transfer de date
- Acces
  - Secvențial vs. aleatoriu
  - În general 2 operații: Read și Write

# Ce operații trebuie să execute software-ul cu un periferic?

- Get and set parameters
- Receive and transmit data
- Enable and disable functions

Cum expunem o interfață din hardware pentru ca software-ul să execute aceste operații?

- Instrucțiuni speciale CPU (de ex., x86 in/out)
- Tratăm perifericele ca și cum ar fi memorie (Memory-Mapped IO)

# Port I/O

- Registrele perifericului sunt mapate ca porturi și ocupă un spațiu separat de adresă

- Folosim instrucțiuni speciale I/O pentru a scrie/citi porturile
- Acces protejat obținut prin faptul că aceste instrucțiuni pot fi executate doar în modul kernel/supervizor
- Exemplu: IBM360 și succesorii

# Memory Mapped IO

- Registrele perifericelor sunt mapate în spațiul obișnuit de adresă

- Folosim instrucțiuni obișnuite (de ex. mov) pentru a citi/scrie registrele hardware ale unui periferic
- Mecanism de protecție a accesului la memorie (memorie virtuală, paginare) pentru a asigura accesul protejat la registrele perifericelor

# Exemplu: din datasheet ATmega324P

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Page |
|---|---|---|---|---|---|---|---|---|---|---|
| 0x18 (0x38) | Reserved | - | - | - | - | - | - | - | - | |
| 0x17 (0x37) | TIFR2 | - | - | - | - | - | OCF2B | OCF2A | TOV2 | 163 |
| 0x16 (0x36) | TIFR1 | - | - | ICF1 | - | - | OCF1B | OCF1A | TOV1 | 144 |
| 0x15 (0x35) | TIFR0 | - | - | - | - | - | OCF0B | OCF0A | TOV0 | 115 |
| 0x14 (0x34) | Reserved | - | - | - | - | - | - | - | - | |
| 0x13 (0x33) | Reserved | - | - | - | - | - | - | - | - | |
| 0x12 (0x32) | Reserved | - | - | - | - | - | - | - | - | |
| 0x11 (0x31) | Reserved | - | - | - | - | - | - | - | - | |
| 0x10 (0x30) | Reserved | - | - | - | - | - | - | - | - | |
| 0x0F (0x2F) | Reserved | - | - | - | - | - | - | - | - | |
| 0x0E (0x2E) | Reserved | - | - | - | - | - | - | - | - | |
| 0x0D (0x2D) | Reserved | - | - | - | - | - | - | - | - | |
| 0x0C (0x2C) | Reserved | - | - | - | - | - | - | - | - | |
| 0x0B (0x2B) | PORTD | PORTD7 | PORTD6 | PORTD5 | PORTD4 | PORTD3 | PORTD2 | PORTD1 | PORTD0 | 99 |
| 0x0A (0x2A) | DDRD | DDD7 | DDD6 | DDD5 | DDD4 | DDD3 | DDD2 | DDD1 | DDD0 | 99 |
| 0x09 (0x29) | PIND | PIND7 | PIND6 | PIND5 | PIND4 | PIND3 | PIND2 | PIND1 | PIND0 | 99 |
| 0x08 (0x28) | PORTC | PORTC7 | PORTC6 | PORTC5 | PORTC4 | PORTC3 | PORTC2 | PORTC1 | PORTC0 | 99 |
| 0x07 (0x27) | DDRC | DDC7 | DDC6 | DDC5 | DDC4 | DDC3 | DDC2 | DDC1 | DDC0 | 99 |
| 0x06 (0x26) | PINC | PINC7 | PINC6 | PINC5 | PINC4 | PINC3 | PINC2 | PINC1 | PINC0 | 99 |
| 0x05 (0x25) | PORTB | PORTB7 | PORTB6 | PORTB5 | PORTB4 | PORTB3 | PORTB2 | PORTB1 | PORTB0 | 98 |
| 0x04 (0x24) | DDRB | DDB7 | DDB6 | DDB5 | DDB4 | DDB3 | DDB2 | DDB1 | DDB0 | 98 |
| 0x03 (0x23) | PINB | PINB7 | PINB6 | PINB5 | PINB4 | PINB3 | PINB2 | PINB1 | PINB0 | 98 |
| 0x02 (0x22) | PORTA | PORTA7 | PORTA6 | PORTA5 | PORTA4 | PORTA3 | PORTA2 | PORTA1 | PORTA0 | 98 |
| 0x01 (0x21) | DDRA | DDA7 | DDA6 | DDA5 | DDA4 | DDA3 | DDA2 | DDA1 | DDA0 | 98 |
| 0x00 (0x20) | PINA | PINA7 | PINA6 | PINA5 | PINA4 | PINA3 | PINA2 | PINA1 | PINA0 | 98 |

# Memory-mapped IO e folosit pentru sisteme embedded

De ce nu Port I/O?

- Instrucțiunile speciale I/O ar fi dependente de ISA (x86, ARM, RISC-V etc.)
  - Devine foarte complicat în momentul în care fiecare microcontroller are propriul set de instrucțuni
- Ai nevoie de hardware special ca să execuți instrucțiunile și să asiguri protecția accesului – costă mai mult

Memory-mapped I/O

- Poate să (re)folosească instrucțiunile din ISA ce referențiază memoria.
  - Poate să reutilizeze codul pentru citire sau scriere (de ex. memcpy() )
- Mecanismul de protecție al memoriei oferă mai multă flexibilitate decât instrucțiuni speciale cu acces protejat (protejezi anumite registre)
- Poți să reutilizezi hardware-ul deja existent pentru memory management/protection - economisești spațiu în siliciu, ții costul și consumul de energie scăzut

# Citirea și scrierea MMIO NU este același lucru ca citirea și scrierea în RAM!

- MMIO citește și scrie registre hardware ale unui periferic
  - Citirile și scrierile pot face perifericul să declanșeze sau să oprească o operație
- Dacă citești date poți face perifericul să declanșeze o acțiune!
  - De ex. dacă faci clear la un flag de întrerupere, perifericul citește următorul octet de pe bus
- Dacă scrii date poți face perifericul să facă ceva cu ele
  - De ex. trimite datele pe care le-ai scris în buffer pe portul serial

# GPIO – General Purpose I/O

Fiecare pin fizic GPIO corespunde unui bit din memoria procesorului. Acel bit poate fi citit (output) sau scris (input)

Poți folosi GPIO ca să controlezi o serie întreagă de dispozitive (lumini, relee, motoare etc.) sau poți folosi la comunicație

Indică faptul că un eveniment s-a petrecut

- Trimite o comandă la chipset-ul radio ca să trimită datele din buffer
- Generază o întrerupere prin care semnalizezi procesorului că un buton a fost apăsat
- Citește starea unui pin pentru a recepționa mesaj de configurare

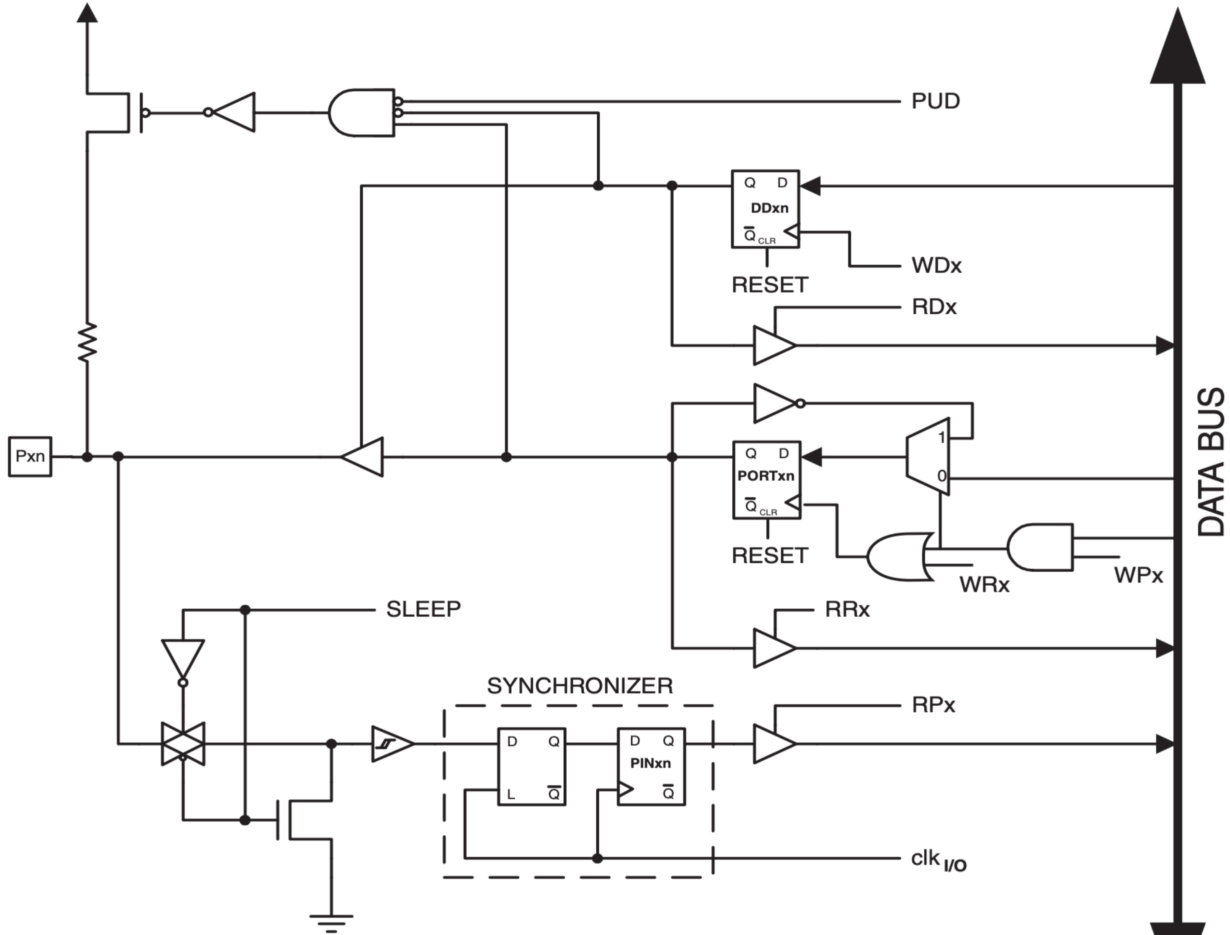
Debugging

- Oare s-a executat acea bucată din codul meu?
- Oare timer-ul generează întreruperi la intervale corecte de timp?

De ce folosim GPIO?

- Operațiile GPIO sunt rapide și ieftine

# Topologia unui pin GPIO



# Configurarea unui pin GPIO

**Table 11-1. Port Pin Configurations**

| DDxn | PORTxn | PUD (in MCUCR) | I/O | Pull-up | Comment |
|---|---|---|---|---|---|
| 0 | 0 | X | Input | No | Tri-state (Hi-Z) |
| 0 | 1 | 0 | Input | Yes | Pxn will source current if ext. pulled low. |
| 0 | 1 | 1 | Input | No | Tri-state (Hi-Z) |
| 1 | 0 | X | Output | No | Output Low (Sink) |
| 1 | 1 | X | Output | No | Output High (Source) |

# Citire și scriere GPIO

```
unsigned char i;

/* Define pull-ups and set outputs high */
/* Define directions for port pins */
PORTB = (1<<PB7)|(1<<PB6)|(1<<PB1)|(1<<PB0);
DDRB = (1<<DDB3)|(1<<DDB2)|(1<<DDB1)|(1<<DDB0);
/* Insert nop for synchronization*/
__no_operation();
/* Read port pins */
i = PINB;
/* Write a pin */
PORTB |= 1<<DDB0;
```

# Fun stuff: Drive Strength

| Symbol | Parameter | Condition | Min. | Typ. | Max. |
|---|---|---|---|---|---|
| $V_{OL}$ | Output Low Voltage[3], | $I_{OL}$ = 20 mA, $V_{CC}$ = 5V<br>$I_{OL}$ = 10 mA, $V_{CC}$ = 3V | | | 0.9<br>0.6 |
| $V_{OH}$ | Output High Voltage[4], | $I_{OH}$ = -20 mA, $V_{CC}$ = 5V<br>$I_{OH}$ = -10 mA, $V_{CC}$ = 3V | 4.2<br>2.3 | | |

Although each I/O port can sink more than the test conditions (20 mA at VCC = 5V, 10 mA at VCC = 3V) under steady state conditions (non-transient), the following must be observed:
1.)The sum of all IOL, for ports PB0-PB7, XTAL2, PD0-PD7 should not exceed 100 mA.
2.)The sum of all IOL, for ports PA0-PA3, PC0-PC7 should not exceed 100 mA.
If IOL exceeds the test condition, VOL may exceed the related specification. Pins are not ensured to sink current greater than the listed test condition.

Although each I/O port can source more than the test conditions (20 mA at VCC = 5V, 10 mA at VCC = 3V) under steady state conditions (non-transient), the following must be observed:
1.)The sum of all IOH, for ports PB0-PB7, XTAL2, PD0-PD7 should not exceed 100 mA.
2.)The sum of all IOH, for ports PA0-PA3, PC0-PC7 should not exceed 100 mA.
If IOH exceeds the test condition, VOH may exceed the related specification. Pins are not ensured to source current greater than the listed test condition.

# AVR Programming

Cum programăm un microcontroller?

- Codul este compilat și rezultă un fișier binar (.hex) ce conține instrucțiunile în cod mașină

Binarul trebuie să ajungă în memoria de program a microcontrolerului

- Folosind un programator extern (In-System Programmer sau JTAG)
- Sau folosind un bootloader
  - Bootloaderul ocupă spațiu în memoria de program!

După programare i se aplică automat un RESET procesorului și acesta începe execuția de la adresa de start

- Depinde de configurație (de ex. unde este scris bootloaderul), poate să nu fie 0

# PROGRAMMING IN C

# Mediul de programare

De obicei programăm în C, modul de lucru este identic cu programarea unui PC

- Cross-platform compiling este regula în programarea embedded

*avr-gcc* este compilatorul de bază pentru AVR

- Nu avem tastatură sau display
- Dar avem stdin și stdout pe care le putem ruta spre un port serial, de exemplu
- Mai simplu de atât: LEDuri!

GCC e configurat pentru arhitectura AVR

- Arhitectură pe 8 biți
- *int* și pointeri pe 16 biți
- Macrodefiniții pentru toate porturile IO, compilatorul știe să folosească instrucțiunile de lucru cu spațiul IO

```
#define PORTC *( volatile uint8_t *) ( 0x028 )
```

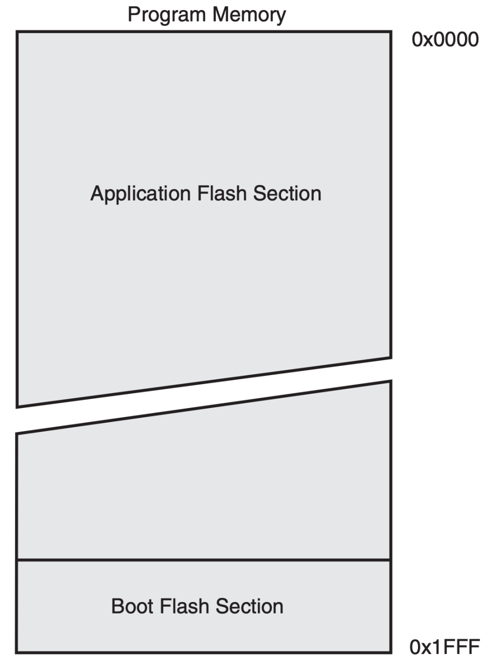
# Variabile și constante

AVR are o arhitectură Harvard, memoria de date și cea de program sunt separate

- SRAM pentru date
- Flash pentru program

Stocarea constantelor

- RAM – nu e indicat, memoria RAM este de mici dimensiuni
- Flash – memoria de program este mai mare, scriere la compilare
- EEPROM – pot fi (re)scrise în timpul execuției programului, dar cu latență mare!

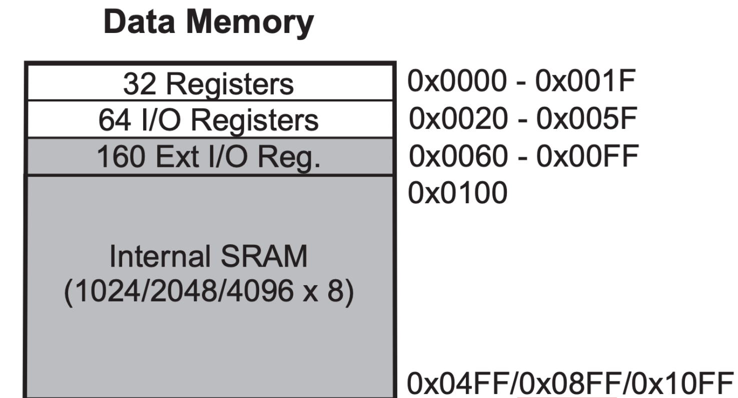



Harta memoriei de program (Flash) la ATmega324P

# Variabile și constante

## Stocarea variabilelor

- Nu avem foarte mult spațiu pentru variabile
  - De ex. ATMega324P are 2kB de SRAM pentru variabile, spațiu în care este mapat și IO și stiva!
- Variabilele locale unei funcții se stochează pe stivă
  - Nu abuzați de recursivitate!
- Contextul întreruperilor se salvează tot pe stivă
  - Nu folosiți întrerupere în întrerupere decât dacă știți foarte bine ce faceți!
- Variabilele statice se alocă în secțiunea .bss
- Variabilele dinamice se alocă pe heap
  - Pentru microcontrolerele cu puțin RAM (ca ATmega324P) nu se recomandă alocarea dinamică!
  - Codul pentru alocatorul de memorie ocupă spațiu în memoria de program!




Alocarea adreselor SRAM la ATmega324P
memory-mapped IO pentru registre generale și periferice
și 2048 (2kB) adrese disponibile pentru variabile și stivă

# Hello World!

De obicei e cel mai simplu program care poate fi scris pe un MPU

```c
/* IO register header */
#include <avr/io.h>
/* For the delay() function */
#include <util/delay.h>

/* Specify the MCU clock frequency */
#define F_CPU 12000000UL

int main()
{
    /* Set pin 0 or PORT C as output */
    DDRC = (1 << PC0);

    while(1)
    {
        /* toggle pin 0 of Port C */
        PORTC ^= (1 << PC0);

        _delay_ms(500);
    }
}
```

# Compilare

Aveți nevoie de avr-gcc și alte utilitare

- Un ghid bun este aici: https://tinusaur.com/guides/avr-gcc-toolchain/

Compilare: `$ avr-gcc main.c -Os -Wall -gdwarf-2 -save-temps -o main.elf`

- În urma compilării rezultă un fișier .elf (Executable and Linkable Format)
- Un fișier ELF poate avea conținutul memoriei de program (Flash), memoriei EEPROM (dacă există date pe care vrem să le scriem inițial) și configurații Fuse-Lockbits specifice microcontrollerului
- Flag-uri utile: -Os optimize for size, -Wall all warnings, -gdwarf-2 produce informații debug

Generarea binarului: `$ avr-objcopy main.elf -O ihex main.hex`

- Fișierul binar conține codul mașină al instrucțiunilor programului compilat
- Nu este un format human-friendly

main.hex

```
:10000000CF93DF93CDB7DEB780E090E0DF91CF9163
:0200100089551
:00000001FF
```

# Dezasamblare

- Putem să dezasamblăm fișierul .elf pentru a inspecta codul în limbaj de asamblare rezultat în urma compilării
- Util când dorim să facem debugging sau când vrem să aflăm cum merg lucrurile sub capotă

Dezasamblare: `$ avr-objdump main.elf –d`

În urma rulării rezultă un listing de cod în limbaj de asamblare cu liniile de cod C intercalate

# Rezultatul compilării

```
00000000 <__vectors>:
__vectors():
0: 0c 94 3e 00 jmp 0x7c  ; 0x7c <__ctors_end>
4: 0c 94 48 00 jmp 0x90  ; 0x90 <__bad_interrupt>
8: 0c 94 48 00 jmp 0x90  ; 0x90 <__bad_interrupt>
c: 0c 94 48 00 jmp 0x90  ; 0x90 <__bad_interrupt>
10: 0c 94 48 00 jmp 0x90 ; 0x90 <__bad_interrupt>
14: 0c 94 48 00 jmp 0x90 ; 0x90 <__bad_interrupt>
18: 0c 94 48 00 jmp 0x90 ; 0x90 <__bad_interrupt>
1c: 0c 94 48 00 jmp 0x90 ; 0x90 <__bad_interrupt>
20: 0c 94 48 00 jmp 0x90 ; 0x90 <__bad_interrupt>
24: 0c 94 48 00 jmp 0x90 ; 0x90 <__bad_interrupt>
28: 0c 94 48 00 jmp 0x90 ; 0x90 <__bad_interrupt>

................................................

60: 0c 94 48 00 jmp 0x90 ; 0x90 <__bad_interrupt>
64: 0c 94 48 00 jmp 0x90 ; 0x90 <__bad_interrupt>
68: 0c 94 48 00 jmp 0x90 ; 0x90 <__bad_interrupt>
6c: 0c 94 48 00 jmp 0x90 ; 0x90 <__bad_interrupt>
70: 0c 94 48 00 jmp 0x90 ; 0x90 <__bad_interrupt>
74: 0c 94 48 00 jmp 0x90 ; 0x90 <__bad_interrupt>
78: 0c 94 48 00 jmp 0x90 ; 0x90 <__bad_interrupt>
0000007c <__ctors_end>:
```

| Vector No. | Program Address[2] | Source | Interrupt Definition |
|---|---|---|---|
| 1 | $0000[1] | RESET | External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset |
| 2 | $0002 | INT0 | External Interrupt Request 0 |
| 3 | $0004 | INT1 | External Interrupt Request 1 |
| 4 | $0006 | INT2 | External Interrupt Request 2 |
| 5 | $0008 | PCINT0 | Pin Change Interrupt Request 0 |
| 6 | $000A | PCINT1 | Pin Change Interrupt Request 1 |
| 7 | $000C | PCINT2 | Pin Change Interrupt Request 2 |
| 8 | $000E | PCINT3 | Pin Change Interrupt Request 3 |
| 9 | $0010 | WDT | Watchdog Time-out Interrupt |
| 10 | $0012 | TIMER2_COMPA | Timer/Counter2 Compare Match A |
| 11 | $0014 | TIMER2_COMPB | Timer/Counter2 Compare Match B |
| 12 | $0016 | TIMER2_OVF | Timer/Counter2 Overflow |
| 13 | $0018 | TIMER1_CAPT | Timer/Counter1 Capture Event |
| 14 | $001A | TIMER1_COMPA | Timer/Counter1 Compare Match A |
| 15 | $001C | TIMER1_COMPB | Timer/Counter1 Compare Match B |
| 16 | $001E | TIMER1_OVF | Timer/Counter1 Overflow |
| 17 | $0020 | TIMER0_COMPA | Timer/Counter0 Compare Match A |
| 18 | $0022 | TIMER0_COMPB | Timer/Counter0 Compare match B |

# Rezultatul compilării

```
__trampolines_start():
    7c: 11 24          eor r1, r1     ; r1 = 0
    7e: 1f be          out 0x3f, r1   ; SREG = r1
    80: cf ef          ldi r28, 0xFF ; 255
    82: d8 e0          ldi r29, 0x08 ; 8
    84: de bf          out 0x3e, r29 ; SPH = 0x8
    86: cd bf          out 0x3d, r28 ; SPL = 0xFF
    88: 0e 94 4a 00    call 0x94      ; 0x94 <main>
    8c: 0c 94 59 00    jmp 0xb2       ; 0xb2 <_exit> 00000090

<__bad_interrupt>: __vector_22():
    90: 0c 94 00 00    jmp 0 ; 0x0 <__vectors>
```

0x7c e adresa la care programul sare la RESET

Inițializare stivă. Stack pointer poziționat pe ultima adresă din RAM (0x08FF pentru ATmega324P)

Apel la rutina main()

Orice întrerupere generează un RESET

Unde găsim ce registre sunt la adresele 0x3f, 0x3e, 0x3d etc.? În datasheet:

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Page |
|---|---|---|---|---|---|---|---|---|---|---|
| 0x3F (0x5F) | SREG | I | T | H | S | V | N | Z | C | 18 |
| 0x3E (0x5E) | SPH | SP15 | SP14 | SP13 | SP12 | SP11 | SP10 | SP9 | SP8 | 19 |
| 0x3D (0x5D) | SPL | SP7 | SP6 | SP5 | SP4 | SP3 | SP2 | SP1 | SP0 | 19 |

# Rezultatul compilării

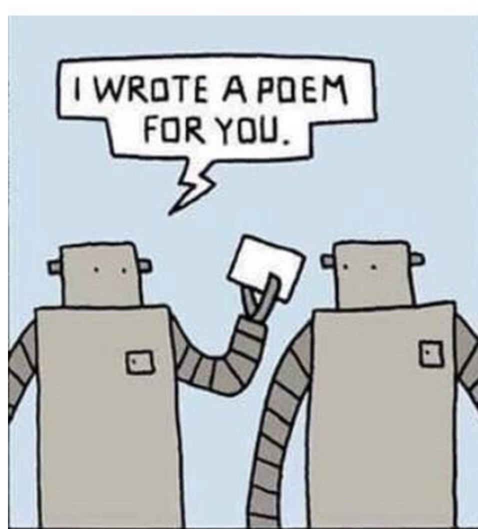
```
00000094 <main>:
main():
#include <util/delay.h>
#define F_CPU 12000000UL

int main() {
    DDRC |= (1 << PC0);

   94: 38 9a sbi 0x07, 0 ; DDRC = 0x01

    while(1){
        PORTC ^= (1 << PC0);
   96: 91 e0 ldi r25, 0x01 ; r25 = 1
   98: 88 b1 in r24, 0x08  ; r24 = PORTC
   9a: 89 27 eor r24, r25  ; r24 = r24 ^ 1
   9c: 88 b9 out 0x08, r24 ; PORTC = r24

    _delay_ms():
   9e: 2f e9 ldi r18, 0x9F  ; 159
   a0: 36 e8 ldi r19, 0x86  ; 134
   a2: 81 e0 ldi r24, 0x01  ; 1
   a4: 21 50 subi r18, 0x01 ; 1
   a6: 30 40 sbci r19, 0x00 ; 0
   a8: 80 40 sbci r24, 0x00 ; 0
   aa: e1 f7 brne .-8       ; 0xa4 <main+0x10>
   ac: 00 c0 rjmp .+0       ; 0xae <main+0x1a>
   ae: 00 00 nop b0: f3 cf rjmp .-26 ; 0x98 <main+0x4>
```

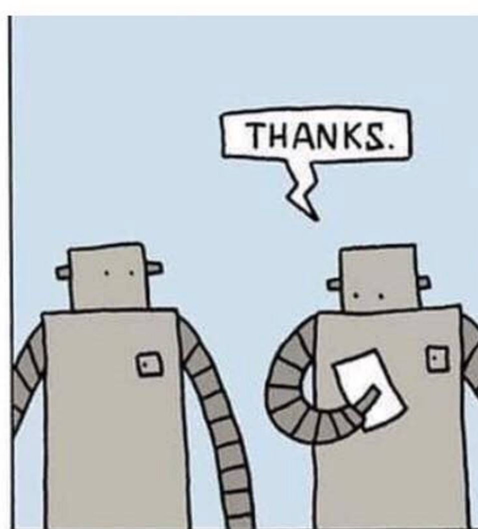
0x94 e prima adresă din main()

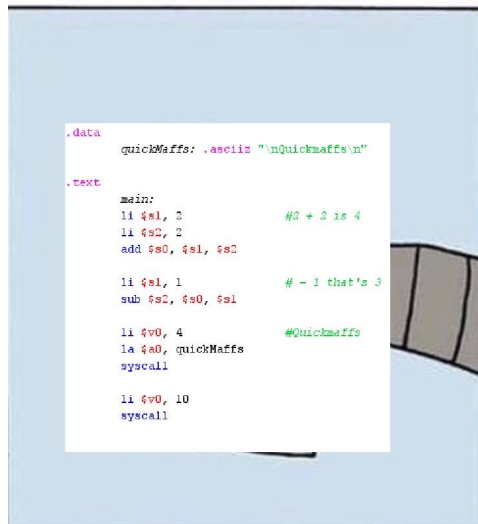
Setează bitul 0 din DDRC (0x07 din RAM)

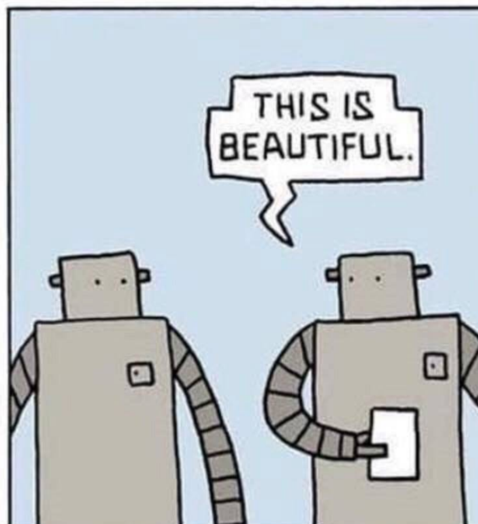
Rutina de delay – numără cicli de ceas până la expirare

Sare înapoi în main după expirarea delay-ului


I WROTE A POEM FOR YOU.


THANKS.


.data
quickMaffs: .asciiz "\nQuickmaffs\n"
.text
main:
li $s1, 2 #2 + 2 is 4
li $s2, 2
add $s0, $s1, $s2
li $s1, 1 # - 1 that's 3
sub $s2, $s0, $s1
li $v0, 4 #Quickmaffs
la $a0, quickMaffs
syscall
li $v0, 10
syscall


THIS IS BEAUTIFUL.