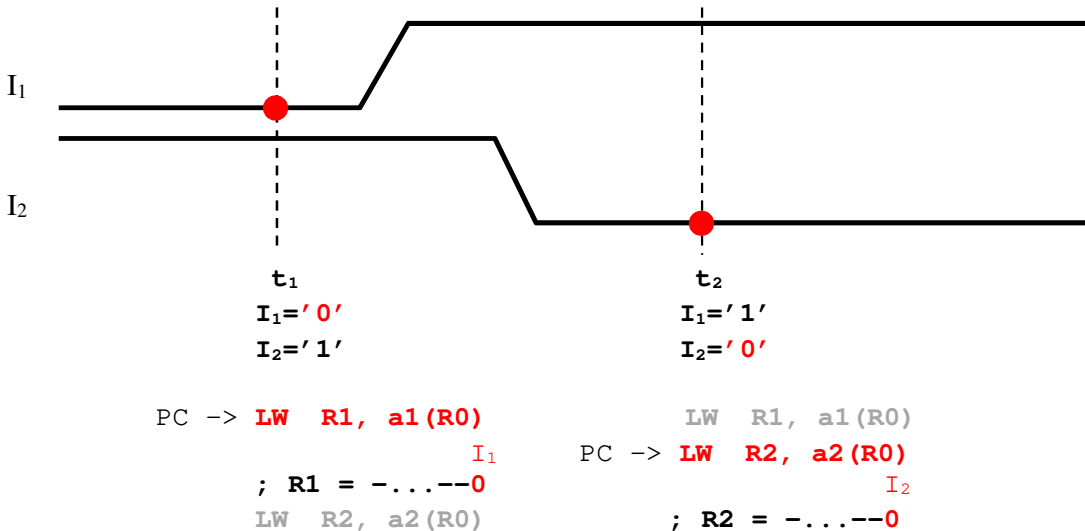


# LABORATOR 3 - Emularea software a SLC-urilor

## Gruparea intrărilor

În prelegerea 2, capitolul 3, este prezentat principiul emulării blocurilor logice combinaționale. Drept exemplu a fost aleasă emularea unei porți AND. Schema sistemului care face emularea este prezentată în figura 7 din prelegerea 2. Această schemă are numai scop didactic, făcând principiul ușor de înțeles.

În practică însă, **schema nu poate fi folosită** deoarece programul de emulare poate folosi valori ale intrărilor care nu au existat în realitate. Vom considera următorul exemplu:



**Obs: ,-, înseamnă valoare nedeterminată**

La portul de intrare de 1 bit cu adresa  $a1$  este conectată intrarea  $I_1$  iar la portul de intrare de 1 bit cu adresa  $a2$  este conectată intrarea  $I_2$ . Deoarece cele două intrări sunt conectate fiecare la portul propriu, sunt necesare două citiri pentru memorarea valorilor lui  $I_1$  și  $I_2$ :

```
LW R1, a1(R0)
LW R2, a2(R0)
```

La momentul  $t_1$  intrările au valorile  $I_1 = '0'$  și  $I_2 = '1'$ . Ca urmare pe bitul 0 din  $R1$  se va memora valoarea logică a lui  $I_1$ , adică ,0'. Până se execută al doilea `LW`,  $I_1$  și apoi  $I_2$  evoluează ca în figura de mai sus.

La momentul  $t_2$  când se execută `LW R2, a2(R0)` intrările au valorile logice  $I_1 = '1'$  și  $I_2 = '0'$ . Ca urmare pe bitul 0 din  $R2$  se va memora valoarea logică a lui  $I_2$ , adică ,0'. Restul programului „va crede” că  $I_1 = '0'$  și  $I_2 = '0'$ , deși intrările nu au avut niciodată aceste valori în intervalul de timp din figură.

**Soluție este ca toate intrările să fie citite în același moment.** Cel mai simplu este ca  $I_1$  și  $I_2$  să fie citite prin intermediul aceluiași port. Această soluție complică un pic programul de emulare, așa cum se va vedea în continuare, dar simplifică logica de decodificare.

## Prezentarea metodei analitice

Fie microsistemul din figura 1 care emulează funcțiile logice  $f_1, f_2, \dots, f_k$  ce depind de variabilele booleene de intrare  $x_1, x_2, \dots, x_n$ :

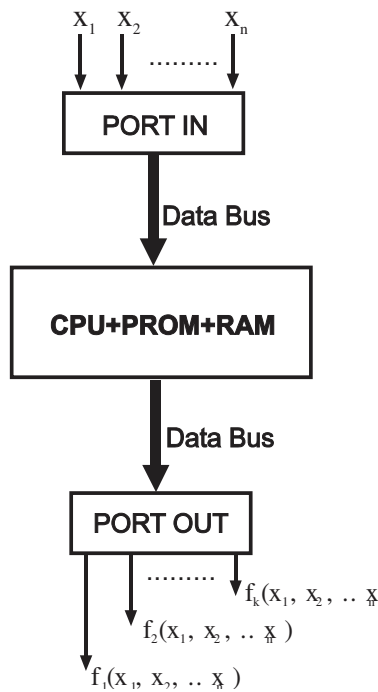


figura 1

### La nesfârșit

- Citește valoarea variabilelor de intrare  $x_1, x_2, \dots, x_n$  prin intermediul portului de intrare și memorează.
  - Calculează  $f_1$  și memorează.
  - Calculează  $f_2$  și memorează.
  - .
  - .
  - .
  - Calculează  $f_k$  și memorează.
  - Scrie valorile calculate pentru  $f_1, f_2, \dots, f_k$  în portul de ieșire.
- repetă.

Ansamblu hardware+software descris anterior se comportă exact ca un multipol logic implementat prin metode clasice de sinteză logică, cu precizarea că întârzierile introduse de această metodă sunt mai mari. În funcție de complexitatea funcțiilor de ieșire și de viteza procesorului se pot ajunge la întârzieri de ordinul milisecundelor.

Acest fapt nu deranjează dacă sistemul din care provin variabilele de intrare și spre care se generează funcțiile de ieșire este lent: de exemplu  $x_1 \dots x_n$  reprezintă starea unor contacte mecanice iar  $f_1 \dots f_k$  reprezintă comenzi către motoare. Timpul de răspuns al unui astfel de sistem este de ordinul zecimilor de secundă așa că întârzieri de ordinul milisecundelor nu deranjează.

**Observație:** Această metodă se folosește cu precădere în cazul în care numărul de variabile de intrare este mai mare sau egal cu 6 și funcțiile de ieșire au o formă analitică simplă. Mai multe detalii în continuare și în laboratorul următor.

## Chestiuni teoretice: setarea, resetarea, testarea și inversarea unui bit sau a mai multor biți dintr-un cuvânt.

Deoarece în C nu există tipul bit sau boolean, variabilele care ar trebui să fie de tip boolean sau bit se vor reprezenta ca biți ai unui cuvânt. Frecvent este nevoie ca să forțăm la ,0', la ,1' sau să inversăm un bit sau un grup de biți dintr-un cuvânt lăsându-i nemodificați pe ceilalți. Operația de forțare se realizează cu ajutorul măștilor și a operatorilor logici care operează la nivel de bit (Bitwise operators).

În general o mască este o constantă care este folosită pentru operațiile la nivel de bit. Prin intermediul măștii unul sau mai mulți biți dintr-un octet sau cuvânt pot fi forțați la ,1', la ,0' sau inversați printr-o singură operație la nivel de bit. Operatorii logici folosiți sunt operatorii logici care operează pe toți biții unui cuvânt, la nivel de bit. Acești operatori sunt:

& (and),      | (or),      ^ ( xor)    ~ (not)

În nici un caz nu se vor folosi operatorii logici !, && și ||. De ce?

### Setarea unui bit într-un cuvânt

Pentru a seta un bit într-un cuvânt, lăsând ceilalți biți nemodificați, vom folosi două proprietăți din algebra booleană. Fie  $x$  o variabilă booleană. Cele două proprietăți sunt:

**Agresivitatea lui ,1' față de operația OR:**  $x \text{ OR } 1' = 1'$

**Neutralitatea lui ,0' pentru operația OR:**  $x \text{ OR } 0' = x$

Pentru a seta un bit vom folosi agresivitatea lui ,1' față de OR. Pentru a păstra valoarea unui bit vom folosi neutralitatea lui ,0' față de OR. Fie  $B$  un octet definit ca

$$\begin{array}{cccccccc} \text{bit} & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ B & = & v_7 & v_6 & v_5 & v_4 & v_3 & v_2 & v_1 & v_0 \end{array}$$

unde  $v_7$  este valoarea bitului 7,  $v_6$  este valoarea bitului 6, etc.

De exemplu, pentru a seta bitul 3 din  $B$  trebuie construită o constantă  $M$  numită mască astfel încât

$$\begin{array}{cccccccc} \text{bit} & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ B & = & v_7 & v_6 & v_5 & v_4 & \mathbf{v_3} & v_2 & v_1 & v_0 & \text{OR} \\ M & = & \underline{m_7} & \underline{m_6} & \underline{m_5} & \underline{m_4} & \underline{\mathbf{m_3}} & \underline{m_2} & \underline{m_1} & \underline{m_0} & = \\ & & v_7 & v_6 & v_5 & v_4 & \mathbf{1} & v_2 & v_1 & v_0 \end{array}$$

Valoarea lui  $m_3$  pentru care

$$v_3 \text{ OR } m_3 = 1'$$

este ,1' ( $v_3 \text{ OR } 1' = 1'$ ), conform teoremei agresivității lui ,1' față de operatorul OR.

Pentru ceilalți biți valoarea lui  $m_i$  pentru care

$$v_i \text{ OR } m_i = v_i$$

este ,0' ( $v_i \text{ OR } 0' = v_i$ ), conform neutralității lui ,0' față de operatorul OR.

În concluzie setarea bitului 3 din  $B$  se face prin intermediul operatorului OR iar masca are un ,1' în poziția 3 și ,0' în rest:

$$\begin{array}{cccccccc} \text{bit} & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ B & = & v_7 & v_6 & v_5 & v_4 & \mathbf{v_3} & v_2 & v_1 & v_0 & \text{OR} \\ m & = & \underline{0} & \underline{0} & \underline{0} & \underline{0} & \underline{\mathbf{1}} & \underline{0} & \underline{0} & \underline{0} & = \\ & & v_7 & v_6 & v_5 & v_4 & \mathbf{1} & v_2 & v_1 & v_0 \end{array}$$

**Regulă: Setarea bitului  $i$  într-un cuvânt  $W$  se face prin intermediul operației OR între  $W$  și masca  $M$ , unde  $M$  este o constantă care are valoarea ,1' în poziția  $i$  și ,0' în rest.**

În cazul în care cuvântul are 8 biți (este un octet) se pot seta 8 biți și în consecință trebuie definite 8 măști. Acestea sunt:

Poziția bitului	Masca binară	Masca hexa	Specificare în C
0	00000001	01	$1 \ll 0$
1	00000010	02	$1 \ll 1$
2	00000100	04	$1 \ll 2$
3	00001000	08	$1 \ll 3$
4	00010000	10	$1 \ll 4$
5	00100000	20	$1 \ll 5$
6	01000000	40	$1 \ll 6$
7	10000000	80	$1 \ll 7$

În C aceste măști se vor specifica cu prefixul 0b sau 0x. Prefixul 0b nu este standard, fiind disponibil numai în cazul anumitor compilatoare. Acest mod de specificare este greoi și poate duce la erori. De exemplu, pentru un cuvânt pe 32 de biți, care este masca pentru bitul 25?

Este

**0b00000000001000000000000000000000**

această mască?

Este clar că în cazul cuvintelor mai late de 8 biți specificatorul 0b nu este potrivit. Nici specificatorul 0x nu este foarte bun. În cazul aceluiași bit 25 dintr-un cuvânt de 32 de biți, este **0x00200000** masca corectă?

Specificarea măștii pentru un singur bit, în limbajul C, se face prin intermediul operației de deplasare: masca pentru bitul  $i$  este  $1 \ll i$ . **Dacă  $i$  este o constantă**, expresia  $1 \ll i$  se calculează la compilare, adică ori că se scrie 0b00000000001000000000000000000000, ori 0x00200000, ori  $1 \ll 25$  codul mașină rezultat este același.

Astfel, pentru setarea bitului 3 din octetul B se va scrie următorul cod C:

```
B = B | 1<<3; //setare bit 3 în B
```

Mai concis, folosind atribuirea compusă, se poate scrie:

```
B |= 1<<3;
```

Acest stil se poate folosi și atunci când trebuie setați mai mulți biți. De exemplu, pentru a seta biții 3 și 5 din B, putem scrie:

```
B |= 1<<3 | 1<<5;
```

Acest stil nu este potrivit atunci când trebuie setați mulți biți. De exemplu dacă trebuie să setăm biții 5..0 din B cel mai potrivit cod este:

```
B |= 0b00111111; // sau B |= 0x3f;
```

### Resetarea biților dintr-un cuvânt

Pentru a reseta un bit într-un cuvânt lăsând ceilalți biți nemodificați, vom folosi două proprietăți din algebra booleană.

Fie  $x$  o variabilă booleană. Cele două proprietăți sunt:

**Agresivitatea lui ,0' pentru operația AND:**  $x \text{ AND } ,0' = ,0'$

**Neutralitatea lui ,1' pentru operația AND:**  $x \text{ AND } ,1' = x$

Pentru a reseta un bit vom folosi agresivitatea lui ,0' față de AND iar pentru a păstra valoarea unui bit vom folosi neutralitatea lui ,1' față de AND. Fie  $B$  octetul definit în paragraful anterior.

De exemplu, pentru a reseta bitul 3 din B trebuie construită masca  $m$  astfel încât:

bit	7	6	5	4	3	2	1	0
B	$v_7$	$v_6$	$v_5$	$v_4$	<b><math>v_3</math></b>	$v_2$	$v_1$	$v_0$
AND								
m	$m_7$	$m_6$	$m_5$	$m_4$	<b><math>m_3</math></b>	$m_2$	$m_1$	$m_0$
	$v_7$	$v_6$	$v_5$	$v_4$	<b>0</b>	$v_2$	$v_1$	$v_0$

Valoarea lui  $m_3$  pentru care

$$v_3 \text{ AND } m_3 = ,0'$$

este ,0' ( $v_3 \text{ AND } ,0' = ,0'$ ), conform teoremei agresivității lui ,0' față de operatorul AND.

Pentru ceilalți biți valoarea lui  $m_i$  pentru care

$$v_i \text{ AND } m_i = v_i$$

este ,1' ( $v_i \text{ AND } ,1' = v_i$ ) conform neutralității lui ,1' față de operatorul AND.

În concluzie resetarea bitului 3 din B se face prin intermediul operatorului AND iar masca are un ,0' în poziția 3 și ,1' în rest:

$$\begin{array}{rcccccccc}
 \text{bit} & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
 B = & v_7 & v_6 & v_5 & v_4 & \textcolor{red}{v_3} & v_2 & v_1 & v_0 & \text{AND} \\
 m = & \underline{1} & \underline{1} & \underline{1} & \underline{1} & \underline{0} & \underline{1} & \underline{1} & \underline{1} & = \\
 & v_7 & v_6 & v_5 & v_4 & \textcolor{red}{0} & v_2 & v_1 & v_0
 \end{array}$$

**Regulă: Resetarea bitului  $i$  într-un cuvânt  $W$  se face prin intermediul operației AND între  $W$  și masca  $M$ , unde  $M$  este o constantă care are valoarea ,0' în poziția  $i$  și ,1' în rest.**

În cazul în care cuvântul este un octet se pot reseta 8 biți și în consecință trebuie definite 8 măști:

Poziția bitului	Masca binară	Masca hexa
0	11111110	fe
1	11111101	fd
2	11111011	fb
3	11110111	f7
4	11101111	ef
5	11011111	df
6	10111111	bf
7	01111111	7f

În cazul resetării, specificarea în C a măștii prin deplasare este imposibilă în mod direct deoarece la deplasare se introduce doar ,0'. Însă dacă observăm că masca pentru resetare este masca de setare negată, putem scrie că masca de resetare pentru bitul  $i$  este  $\sim(1 \ll i)$ . Dacă  $i$  este o constantă, expresia  $\sim(1 \ll i)$  se calculează la compilare.

Astfel, pentru resetarea bitului 3 din octetul B se va scrie următorul cod C:

```
B = B & ~(1<<3); //resetare bit 3 în B
```

Mai concis, folosind atribuirea compusă, se poate scrie:

```
B &= ~(1<<3);
```

Acest stil se poate folosi și atunci când trebuie resetați mai mulți biți. De exemplu, pentru a reseta biții 3 și 5 din B, putem scrie:

```
B &= ~(1<<3 | 1<<5);
```

Acest stil nu este potrivit atunci când trebuie setați mulți biți. De exemplu dacă trebuie să resetam biții 5..0 din B cel mai potrivit cod este:

```
B &= 0b11000000; // sau B &= 0xC0;
```

### Testarea unui bit dintr-un cuvânt

În programele scrise pentru microcontrolere apare frecvent necesitatea de a executa o secvență de instrucțiuni sau alta în funcție de valoarea unui bit dintr-un cuvânt. Secvență tipică în care trebuie testată valoarea unui bit este:

```
if( bitul_i_din_w == ,1')
    secventa1
else
    secventa2
```

Deoarece în C standard nu există tipul bit, trebuie construit un cuvânt care să fie 0 (adică să aibă toți biții ,0') dacă bitul  $i$  este ,0' ori diferit de zero altfel.

În cazul bitul 3 dintr-un octet, acest cuvânt se construiește astfel:

$$\begin{array}{r}
 \text{bit } 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1 \ 0 \\
 B = v_7 \ v_6 \ v_5 \ v_4 \ \textcolor{red}{v_3} \ v_2 \ v_1 \ v_0 \quad \text{AND} \\
 m = \begin{array}{ccccccc} 0 & 0 & 0 & 0 & \textcolor{red}{1} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & \textcolor{red}{v_3} & 0 & 0 & 0 \end{array} =
 \end{array}$$

Dacă valoarea bitului 3 este ,0' rezultatul va fi 00000000 iar în caz contrar va fi 00001000.

**Testarea bitului  $i$  într-un cuvânt  $W$  se face prin intermediul operației **AND** între  $W$  și masca  $M$ , unde  $M$  este o constantă care are valoarea ,1' în poziția  $i$  și ,0' în rest.**

Astfel, pentru testarea bitului 3 din octetul  $B$  se va scrie următorul cod C:

```

if(B & 1<<3) //testare bit 3 din B
    //do something
else
    //do something else

```

### Inversarea

Pentru a inversa un bit se folosesc următoarele două proprietăți ale operatorului xor:

$$\text{bit xor ,1'} = \overline{\text{bit}} \quad \text{bit xor ,0'} = \text{bit}$$

Dacă vrem să inversăm anumiți biți dintr-un cuvânt masca se construiește astfel: dacă bitul  $i$  din cuvânt trebuie inversat, bitul  $i$  din mască va fi ,1' iar dacă bitul  $i$  din cuvânt nu trebuie modificat, bitul  $i$  din mască va fi ,0'.

### Chestiuni teoretice: porturile digitale la microcontrolerul ATmega16

**Este obligatoriu să citiți și sa înțelegeți capitolul 2 „Porturile A, B, C și D” din prelegerea 3. Altfel va fi imposibil să terminați acest laborator!**

### Desfășurarea lucrării

Se vor implementa 3 funcții booleene, conform următoarei table de adevăr:

X	$x_2$	$x_1$	$x_0$	$f_2$	$f_1$	$f_0$
0	0	0	0	0	1	0
1	0	0	1	0	1	1
2	0	1	0	1	1	1
3	0	1	1	1	0	0
4	1	0	0	0	0	1
5	1	0	1	1	0	0
6	1	1	0	0	0	0
7	1	1	1	1	0	1

Funcția  $f_0$  este ,1' când numărul de ,1'-uri din  $X=x_2x_1x_0$  este impar,  $f_1$  este ,1' când  $X<3$  iar  $f_2$  este ,1' când numărul  $X$  este un număr prim.

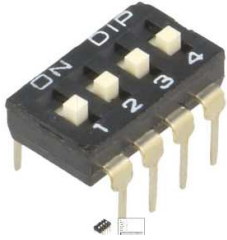
Variabilele de intrare  $x_{2:0}$  se vor conecta la portului PORTB iar funcțiilor de ieșire  $f_{2:0}$  se vor trimite în exterior prin portul PORTA, conform următoarei corespondențe:

- $x_0 \rightarrow$  portul B, bit 0 (pin PB0)
- $x_1 \rightarrow$  portul B, bit 1 (pin PB1)
- $x_2 \rightarrow$  portul B, bit 2 (pin PB2)
- $f_0 \rightarrow$  portul A, bit 0 (pin PA0)
- $f_1 \rightarrow$  portul A, bit 1 (pin PA1)
- $f_2 \rightarrow$  portul A, bit 2 (pin PA2)

## Pasul 1: Implementarea hardware

Se va porni de la montajul din laboratorul precedent (Blink). La montajul existent se vor adăuga 3 comutatoare, 3 LED-uri și rezistențe.

Fiecare variabilă de intrare va fi implementată prin intermediul unui comutator. Se dispune de 4 comutatoare grupate într-un singur componentă. Această componentă este un DIP SWITCH, prezentat în figura din stânga. Conform schemei, când un comutator este închis (poziție ON) pinul corespunzător este conectat la Vcc iar când comutatorul este deschis pinul este conectat la masă prin rezistență. Conectarea comutatoarelor la microcontroler se va detalia în prelegerea 4.



**Mai întâi se conectează la alimentare coloanele 5P și 2P prin intermediul liniilor 2H (vezi figura 2, lab. 2).**

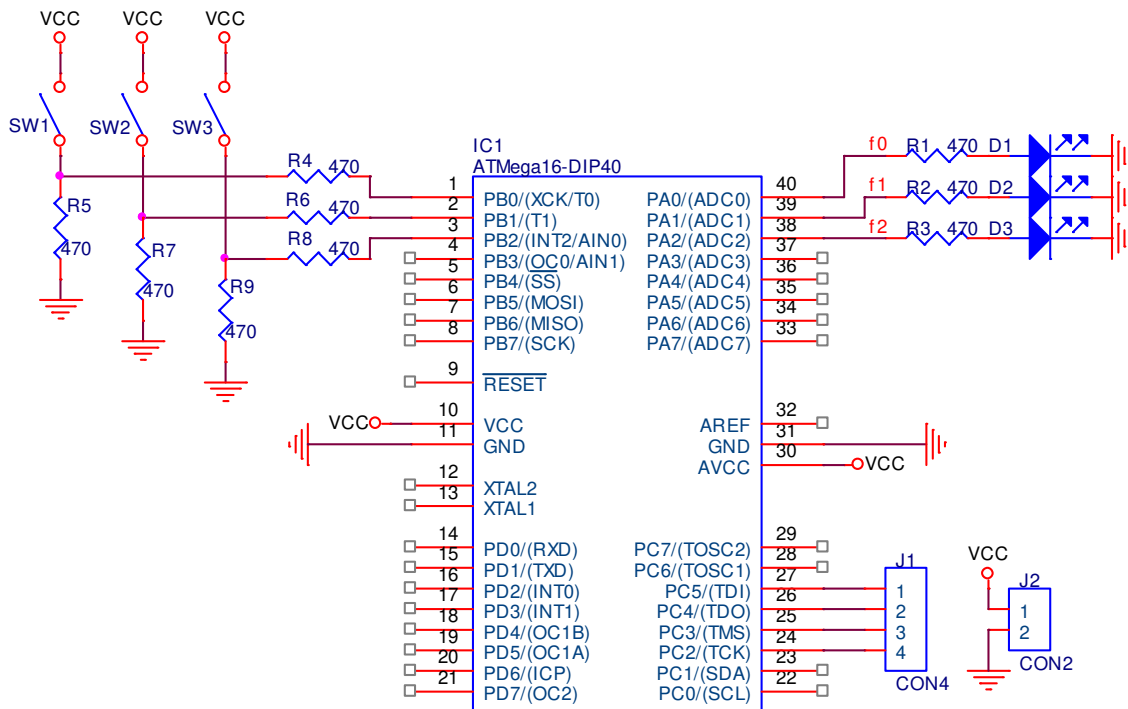


figura 2

Se va realiza montajul conform schemei din figura 2, iar amplasarea componentelor se va face ca în figura 3.

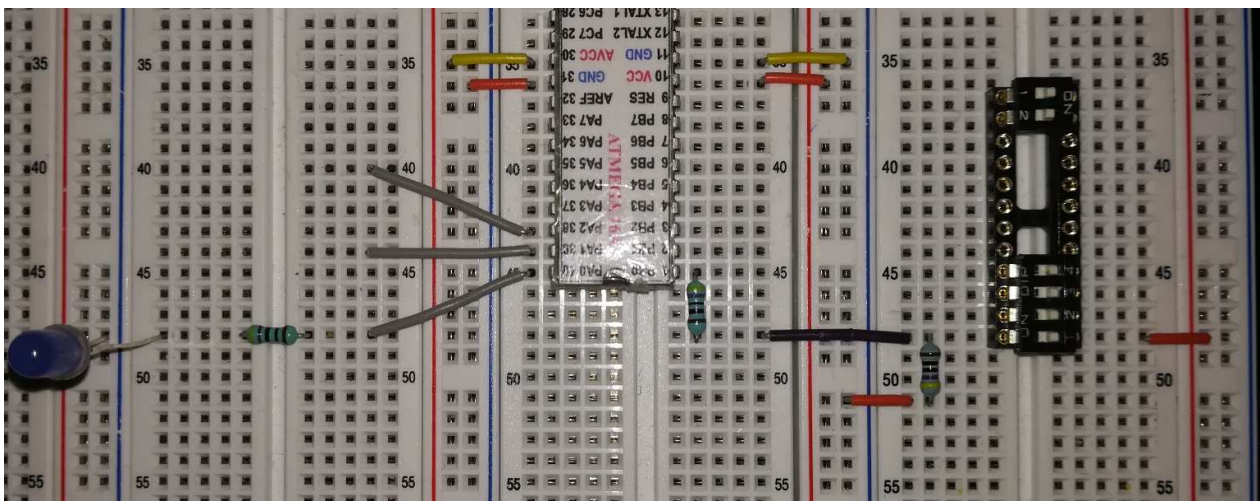


figura 3



În figura 3 este sugerată dispunerea componentelor. În figură sunt montate toate componentele și firele folosite pentru funcția  $f_0$ . Pentru  $f_1$  și  $f_2$  sunt montate doar firele care conectează rezistențele R2 și R3 cu microcontrolerul. Remarcați **dispunerea în evantai** a firelor gri, pentru ca LED-urile să aibă loc unul lângă altul. Între perechi trebuie să existe 2, 3 găuri libere. Montați rezistențele R2, R3 și LED-urile D2 și D3.

În figura 3 este implementată și intrarea  $x_0$ . În continuare montați firele și rezistențele pentru celelalte două intrări.

**Atenție!** Din cauza **deformărilor** datorate aparatului de fotografiat neperformant anumite găuri pot apărea dezaliniat față de pinii componentelor! Implementați după schema electrică din figura 2! Schema din figura 3 este doar pentru a sugera dispunerea componentelor și a firelor.

**Atenție!** Valoarea rezistențelor R5, R7 și R9 este prea mică dar este folosită în schemă pentru că este egală cu valoarea rezistențelor folosite la LED-uri. În acest fel în schemă toate rezistențele au aceeași valoare, 470Ω, și nu mai pot apare greșeli. Valoarea normală pentru o rezistență în serie cu comutatorul este 1- 5 KΩ.

Rezistențele R4, R6 și R8 au rolul de a proteja microcontrolerul în cazul în care direcția portului B este programată greșit (OUT în loc de IN). Dacă am fi siguri că direcția portului B este programată corect, aceste rezistențele ar putea fi eliminate și în locul lor ar trebui montate fire. Cum sunteți începătorii și începătorii fac multe greșeli, OBLIGATORIU montați aceste rezistențe.

După ce ați realizat montajul, **NU alimentați!** Dacă alimentați montajul fără validare prealabilă **veți plăti componentele distruse!**

În continuare scrieți codul C.

## Pasul 2: Crearea proiectului

Crearea proiectului se face conform pașilor prezentați în laboratorul anterior. Pe scurt, aceștia sunt:

1. Se lansează în execuție **AVR Studio4**.
2. Din fereastra **Welcome to AVR Studio 4** se alege opțiunea **New Project**.
3. În fereastra **Welcome...** alegeți proiect de tip **AVR GCC**, bifați checkbox-urile **Create initial file** și **Create folder**, stabiliți locația proiectului (evident pe D:\micro\...) și alegeți un nume pentru proiect. În această lucrare de laborator numele folosit în continuare va fi **functii**. În final apăsați butonul **Next**.
4. În următoarea fereastră alegeți drept platformă de debug **JTAG ICE**, tipul microcontrolerului **ATmega16**, lăsați opțiunea port pe **Auto** și apoi apăsați **Finish**.

## Pasul 3: Crearea sursei

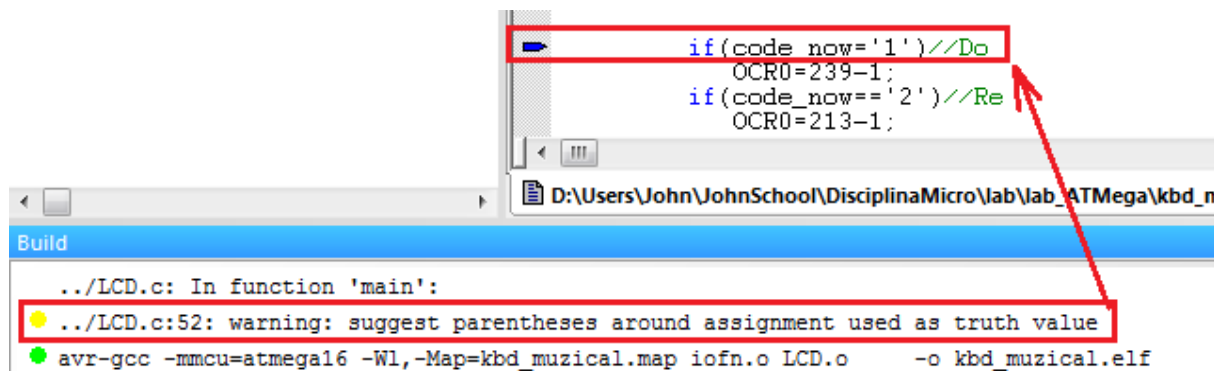
### Cerințe obligatorii:

1. Pentru o editare mai ușoară a sursei, folosiți Notepad++. **Editati același fișier** și cu editorul AVR, și cu Notepad++. Atunci când salvați cu unul din editoare, celălalt vă va întreba dacă vreți să actualizați. Răspundeți cu Da. Nu faceți modificări în modul Debug.
2. **Este OBLIGATORIU să indentați corect programul.** În AVR studio, fereastra **Tools** → **Options** → **Editor** se setează **Tab width** la 3 și se validează **Repace tab with spaces**. Aceleași setări se fac și în Notepad++. Dacă profesorul va observa că sursa nu este corect indentată, **veți fi**



**penalizați cu două puncte** și veți modifica codul pentru a respecta indentarea. Media penalizărilor sau o parte din ea se va scădea din nota obținută la laborator. Rezultatul scăderii nu va fi mai mic ca 5. Restul de puncte de penalizare, dacă există, se va scădea din nota examenului final. De exemplu, dacă media la laborator este 5,5 și media penalizărilor este 1, nota la laborator va deveni 5 și mai rămân 0,5 puncte de scăzut din nota de la examen. Atenție! **Există riscul să picați examenul din acest motiv!**

3. **Este OBLIGATORIU să eliminați toate avertizările** (warnings) înainte de a testa programul. De cele mai multe ori un warning ascunde o greșeală de programare. În următorul fragment de cod apare una din cele mai frecvente erori de programare în C: = în loc de ==:



În acest caz compilatorul generează avertizarea marcată cu cerc galben. Ignorarea avertizării ne face să pierdem foarte mult timp cu depanarea. Este mult mai eficient să examinăm toate avertizările și să eliminăm cauzele care le produc în 3 minute decât să pierdem ore cu depanarea.

**Înainte de executarea programului executați Rebuild All.** AVR Studio prezintă o comportare ciudată: dacă modificați codului și executați **Build**, AVR Studio afișează toate warningurile. Fără să modificați codul executați **Build** încă o dată: de data aceasta avertizările nu mai apar. Din acest motiv executați **Rebuild all** înainte de a programa microcontrolerul.

**Codul care trebuie adăugat în fișierul funcții.c este marcat cu verde.** Pe măsură ce citiți indicațiile ce urmează, adăugați codul necesar în sursă. Nu este obligatoriu să adăugați comentariile.

```
#include <avr/io.h>

int main() {
    // valorile memorate ale intrarilor
    unsigned char inputs;
    // Daca nu va place unsigned char puteti folosi uint8_t definit in stdint.h
    // uint8_t este definit ca: typedef unsigned char uint8_t
    // Variabila xi, (i=0,1,2) memoreaza in bitul i valoarea intrarii xi.
    // Ceilalti biti sunt 0.
    unsigned char x2, x1, x0;

    // in variabila outs se assembleaza iesirile astfel:
    // bit   7 6 5 4   3 2 1 0
    // outs= - - - -   - f2 f1 f0
    unsigned char outs;
    unsigned char temp;
```

Mai întâi vom configura porturile A și B.

**Citiți și înțelegeți capitolul 2 „Porturile A, B, C și D” din prelegerea 3.**

Configurați portul A pentru ca toți pinii portului să fie de ieșire:

```
DDRA=0b????????; //sau DDRA = 0x??;
```

```
DDRB=0b???????; //sau DDRB = 0x??;
```

```
while (1) {
```

*Nume\_variabila*=PINX; //X se înlocuiește cu A, B, C sau D;

```
//memoreaza valorile variabilelor de intrare
//bit          7654 3 2   1  0
//inputs =      x2 x1 x0
inputs = PINB;
```

```

bit      7654 3  2  1  0
inputs = ---- - x2 x1 x0

```

```

bit      7654 3  2  1  0
inputs = ---- - x2 x1 x0   operator (care?)
         = ??? ?  ?   ?  ?
-----
          0000 0  x2 x1 x0

```

```
inputs = ??;
```

```

      inputs
bit 7654 3  2  1  0
      0000 0 x2 x1 x0 →
      |
      |
      |-----→
      |
      |
      |-----→

      x0
bit 7654 321 0
      0000 000 x0

      x1
bit 7654 321 0
      0000 000 x1

      x2
bit 7654 321 0
      0000 000 x2

```

Se observă că variabila  $x_0$  se poate genera din variabila *inputs* dacă se resetează toți biții acesteia, mai puțin bitul 0:

```
// x0 = 0000 000x0; Variabila x0 conține numai intrarea x0
x0 = inputs & 1<<0;
```

În cazul lui  $x_1$  doar mascarea este insuficientă deoarece valoarea intrării  $x_1$  apare pe bitul 1. Pentru ca valoarea intrării  $x_1$  să apară pe bitul 0 trebuie ca să deplasăm dreaptă pe *inputs* și apoi să resetăm biții 7..1. **Atenție!** În procesul de calculare a lui  $x_1$  și  $x_2$  **NU** modificați pe *inputs*! Vom avea nevoie de el mai târziu.

```
// x1 = 0000 000x1
x1 = inputs...;
```

Pentru  $x_2$  procedați asemănător:

```
// x2 = 0000 000x2
x2 = inputs...;
```

Cele 3 funcții se vor asambla în variabila *outs*. Prima operație care o vom face este

```
outs = 0;
```

Rolul acestei operații va deveni evident în scurt timp.

Calculul celor 3 funcții începe cu  $f_0$ . Funcția  $f_0$  este ,1' când numărul de ,1'-uri din  $X=x_2x_1x_0$  este impar. Altfel spus  $f_0$  este bitul de paritate. Se știe că paritatea se calculează cu funcția *xor*. Astfel  $f_0 = x_0 \oplus x_1 \oplus x_2$ . Codul C pentru implementarea lui  $f_0$  este:

```
// f0 se calculeaza in temp
temp = x2 ^ x1 ^ x0;
```

Apoi *temp* este folosit pentru a stabili valoarea bitului 0 din *outs*:

```
if( temp & 1<<0)
    outs = ??;
```

Deoarece *outs* a fost inițializată cu 0, ramura *else* nu mai este necesară. Astfel programul devine mai scurt. **Nu uitați, avem la dispoziție doar 16 KB** iar o instrucțiune ocupă 2 octeți.

Operația **&1<<0** este necesară pentru ca rezultatul testului *temp != 0* să depindă numai de bitul 0 al lui *temp*. Dar biții 7:1 pot să ia valori diferite de ,0' în cursul calculării lui *temp*. Ca exemplu vom considera  $\sim x$ , unde  $x$  este o variabila de tip unsigned char:

```
// Acesta este un exemplu. Nu copiați acest cod!
temp=~x;
// bit      7654 3210
//      x = 0000 000v
//      ~x = 1111 111v
```

În acest caz se observă că *temp* nu mai are formatul 0000\_000bit, motiv pentru care **&1<<0** este obligatoriu. Pentru siguranță se recomandă să faceți **&1<<0** atunci când generați o nouă valoare de tip „bit reprezentat pe char”. Puteți încălca această recomandare dacă sunteți absolut siguri că operația nu este necesară. De exemplu pentru operația *temp*=~A & ~B; este necesar **&1<<0**, sau nu?

În continuare se va calcula funcția  $f_1$ . La prima vedere am putea proceda ca pentru  $f_0$ , adică să găsim forma minimă. Dacă faceți minimizarea rezultă forma  $f_1 = \bar{x}_2(\bar{x}_1 + \bar{x}_0)$ . Deși simplu, calculul expresiei necesită executarea a 5 instrucțiuni în cod mașină.

Forma minimă este cea mai bună soluție dacă implementarea se face cu porți logice deoarece oferă cel mai mic preț de cost. Dar emularea funcțiilor logice se face prin executarea unui program, nu cu porți logice. Din acest motiv **prețul de cost al emulării trebuie exprimat în număr de instrucțiuni cod mașină, nu în porți logice**. Prețul de cost al emulării nu este altceva decât **complexitatea** implementării.

În unele cazuri prețul de cost al implementării hardware coincide cu complexitatea exprimată în număr de instrucțiuni cod mașină. Un astfel de caz este funcția  $f_0$ : și în varianta hardware, și în varianta emulării software nu există nimic mai simplu decât un xor de trei variabile. În alte cazuri însă există soluții software care nu au echivalent hardware.

Chiar dacă implementăm funcții booleene nu înseamnă ca trebuie obligatoriu să folosim numai algebra booleană. Dispunem de un întreg ALU! De ce să nu-l folosim?

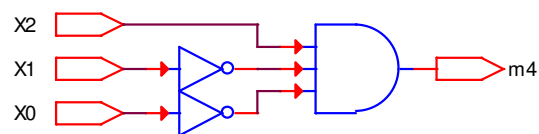
### Emularea funcțiilor logice

În cadrul cursului de proiectare logică s-au prezentat conceptele pe care se bazează implementarea funcțiilor logice: **mintermen**, **maxtermen**, **forma canonică** și **forma minimă**. Revedeți aceste concepte!

Vom începe analiza cu implementarea unui **mintermen**. Ca exemplul s-a ales mintermenul  $m_4$  de trei variabile. Tabela de adevăr pentru  $m_4$  este următoarea:

X	$x_2$	$x_1$	$x_0$	$m_4$	$M_4$	$f_1$
0	0	0	0	0	1	1
1	0	0	1	0	1	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	0	0
5	1	0	1	0	1	0
6	1	1	0	0	1	0
7	1	1	1	0	1	0

#### Implementare hardware pentru $m_4$



Forma lui  $m_4$  este  $m_4 = x_2 \bar{x}_1 \bar{x}_0$ . Implementarea hardware se face cu o poartă AND și două inversoare și este prezentată în partea dreaptă a tabelului.

În C emularea bazată pe algebra a mintermenul  $m_4$  booleană se face cu

$$m4 = (x2 \ \& \ \sim x1 \ \& \ \sim x0) \ \& \ 1 < 0;$$

Numărul de instrucțiuni în cod mașină pentru implementarea lui  $m_4$  este 5: două NOT-uri și trei AND-uri.

În C este posibilă o altă variantă de implementare, variantă care se bazează pe echivalentul zecimal al intrării. Intrarea X construită prin concatenarea celor 3 variabile de intrare este un număr care ia toate valorile între 0 și 7. Mintermenul  $m_4$  este ,1' când X este 4 și ,0' în rest. Deoarece în programul scris până în acest moment X este variabila *inputs* putem coda această observație după cum urmează:

```
if(inputs==4)
    m4=1;
else
    m4=0;
```

Operatorul relațional `==` din C se traduce în cod mașină prin instrucțiunea `CPI reg, 4`. CPI înseamnă compară imediat iar `reg` conține variabila `inputs`. Această variantă de calculare a lui `m4` necesită doar o instrucțiune în cod mașină față de 5 instrucțiuni necesare variantei bazate pe algebra booleană. Este evident că **varianta cu operator relațional este mult mai bună!**

La fel de simplu se pot implementa și funcțiile **maxtermen**. Maxtermenul 4 prezentat în tabelul anterior în coloana M4 se implementează după cum urmează:

```
if(inputs ==4)
    M4=0;
else
    M4=1;
```

Dacă emularea mintermenilor și maxtermenilor este atât de simplă cu operatori relaționali, merită să analizăm cum s-ar implementa **forma canonică** a funcțiilor booleene. Forma canonică disjunctivă a unei funcții este o sumă OR de mintermeni. Ca exemplu, vom scrie forma canonică a lui  $f_1$ . Funcția  $f_1$  este compusă din mintermenii 1, 2 și 3. Forma sa canonică este  $f_1 = \sum(0,1,2) = m_0 + m_1 + m_2$ . Dacă folosim echivalentul zecimal al intrării, forma canonică disjunctivă este echivalentă cu a spune că  $f_1$  este ,1' dacă X este egal cu 0 sau X este egal cu 1 sau X este egal cu 2. Implementarea formei canonice în C este

```
if(inputs == 0 || inputs == 1 || inputs == 2)
    f1=1;
else
    f1=0;
```

Implementarea se face cu 5 instrucțiuni, la fel ca implementarea formei minime  $\bar{x}_2(\bar{x}_1 + \bar{x}_0)$ . Avantajul ar fi că am evitat consumul de timp cu minimizarea (pe care oricum ați uitat-o).

Emularea  $f_1$  lui se poate face chiar cu mai puține instrucțiuni dacă facem următoarea observație: funcția este ,1' dacă echivalentul zecimal al intrării este mai mic decât 3. Emularea lui  $f_1$  se poate face cu:

```
if(inputs<3)
    outs ???; //seteaza bitul 1 din outs
```

Se observă că implementarea lui  $f_1$  s-a făcut cu o singură operație (exceptând atribuirea) în loc de 5 operații!

**În concluzie emularea funcțiilor booleene se poate face cu operatori logici sau aritmetic**, adică adunare, scădere, deplasări rotiri, comparații, operatori logici, etc. Totul este ca emularea nonbooleană să fie mai scurtă sau egală cu implementarea formei minime.

**Atenție:** operatorii logici au pret de cost diferit. Operatorii `==`, `!=`, `>`, `<` necesită pe majoritatea procesorelor o singură instrucțiune cod mașină. În schimb operatorii `>=`, `<=` necesită de regulă două instrucțiuni, așa că în loc de `var >= 2` folosiți `var>1`.

În final se calculează  $f_2$ :

- **Aflați forma minimă pentru  $f_2$ !**
- Găsiți o implementare cu operatori relaționali și logici.
- Alegeți varianta cea mai simplă. Dacă nu aveți decât o variantă din cele două, puteți lua nota 5 dar nu puteți continua pentru notă mai mare!

Apoi setați bitul corespunzător lui  $f_2$  (bitul 2) în `outs`. Adăugați doar unul din blocurile de cod următoare:

```
// dacă ați ales minimizarea, f2 se calculeaza in temp
temp = ...;
if( temp & 1<<0)
    outs = ???;

// dacă ați găsit o condiție:
if(conditie)
    outs = ???; //setează bitul 2 din outs
```

Ultima operație este scrierea în portul de ieșire. Scrierea unui port de ieșire se face simplu cu:

**PORTX= Nume\_variabila;**




unde *Nume\_variabila* este o variabilă de tip unsigned char. Observați diferența față de citire din port. Ultima instrucțiune va fi:


**PORTA=outs;**


În acest moment ar trebui ca fișierul sursă să fie complet.

```
    }//end main loop
} //end main
```

#### Pasul 4: Compilarea și execuția

1. Mai întâi compilați și linkeditați apăsând butonul **Build** .
2. **Chemați profesorul pentru a verifica indentarea sursei și corectitudinea montajului.**
3. Alimentați!
4. Apăsați butonul **Con.** În fereastra **Select Avr Programmer** nou apărută selectați platforma **JTAG ICE**. Pentru port selectați **Auto** și apoi apăsați butonul **Connect...**
5. Dacă se stabilește conexiunea cu  $\mu C$ , va apare fereastra **JTAG ICE in JTAG mode ...**. Dacă nu reușiți să stabiliți conexiunea, executați procedura detaliată în laboratorul precedent. Dacă nici după aceasta nu se stabilește legătura, **chemați profesorul!**
6. În fereastra JTAG ICE... apărută la pasul anterior selectați câmpul tab **Fuses**. Trebuie să aveți bifate opțiunile **OCDEN**, **JTAGEN** iar **SUT\_CKSEL** trebuie să fie 8MHz+64ms. **Dacă a fost necesar** să schimbați vreuna din setările, apăsați butonul **Program!**
7. Selectați câmpul tab **LockBits**. Verificați dacă toate opțiunile încep cu **No**. În caz contrar schimbați și programați.
8. Apoi apăsați butonul **Start debugging** . Ca urmare codul mașină obținut după Build va fi programat prin intermediul JTAG ICE în memoria de cod a uC. De asemenea programul se va opri la începutul lui main. Locul în care este oprit programul este marcat de o săgeată galbenă.
9. Pentru a executa codul mașină înscris anterior în flash apăsați butonul **Run** .

Pentru a opri execuția programului apăsați butonul **Break** . Puteți relua execuția apăsând din nou **Run**. Puteți opri și relua execuția de câte ori doriți.

Pentru a ieși din modul debug apăsați butonul **Stop Debugging** .

### Pasul 5: Testarea codului

Se fac **toate combinațiile posibile** ale intrărilor și se observă cele 3 LED-uri. În cazul în care nu funcționează, acesta fiind cazul cel mai frecvent, este necesară depanarea aplicației (vezi pasul 6).

**Atenție!** Dacă nu aveți decât o variantă de implementarea a lui f2 din cele două, puteți chema profesorul și puteți lua nota 5 dar nu puteți continua pentru notă mai mare!

După ce funcționează, **chemați profesorul pentru validare!**

Împreună cu profesorul, **faceți următorul experiment:**

- Îndepărtați cele 3 fire care conectează la masă rezistențele R4, R5 și R6.
- Deschideți toate comutatoarele, adică formați combinația 000. Astfel intrările x2, x1 și x0 vor rămâne neconectate (în aer).
- Apropiati mâna de comutatoare și de rezistențe. Puteți pune mâna pe ele. Observați ce se întâmplă.
- Discutați cu profesorul.

**Dacă ați ajuns aici aveți nota 5!**

### Pasul 6: Depanarea programelor (opțional)

**Cine știe cum se depanează programele poate trece la pasul 6.**

Deoarece codul nu funcționează de prima dată, este necesară depanarea.

Mijloacele clasice de depanare sunt:

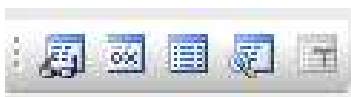
1. **Oprirea execuției programului și repornirea acestuia în orice moment.**
2. **Vizualizarea variabilelor (watch)**
3. **Execuția pas cu pas.**
4. **Oprirea execuției înainte de execuția unei linii de cod.**
5. **Inspectarea memoriei**

Toate aceste opțiuni de depanare sunt disponibile și în AVR Studio și sunt grupate în meniul **Debug**. Multe opțiuni din acest meniu apar și în bara de scule **Debug**. Dacă această bară nu este vizibilă, activați-o cu **View → Toolbars → Debug**. Această bară este prezentată în figura următoare:




Rolul primelor 4 butoane a fost deja explicat.

De asemenea activați (dacă nu este deja activată) bara **Debug Windows** cu **View → Toolbars → Debug Windows**. Această bară este prezentată în figura următoare:





## Vizualizarea stării variabilelor.

Dacă execuția programului este oprită, se poate vizualiza starea oricărei variabile dacă se plasează cursorul deasupra acesteia și se așteaptă aproximativ o secundă. Dacă se dorește afișarea mai multor variabile, se apasă butonul , ceea ce are ca efect activarea ferestrei **Watch**:

Watch				
Name	Value	Type	Location	
x0	0x01 ' '	char	0x045A [SRAM]	
x1	0x01 ' '	char	0x0459 [SRAM]	
Watch 1 Watch 2 Watch 3 Watch 4				

Pentru a adăuga o variabilă în fereastra **Watch**, faceți dublu clic în prima celulă liberă din coloana **Name** și apoi introduceți numele variabilei. Alternativ, selectați în editor numele unei variabile și apoi faceți **Drag&Drop** oriunde în fereastra **Watch**.

## Execuția pas cu pas.

Dacă programul este oprit, acesta se poate rula linie cu linie. **Atenție, linie cu linie, nu instrucțiune cu instrucțiune!**

### Foarte important:




Pentru a executa programul instrucțiune cu instrucțiune **este OBLIGATORIU** să scrieți o singură instrucțiune pe linie.

Teoretic, execuția unei linii urmată de oprire ar necesita un singur buton. Fie numele acestuia **Step**. Ce s-ar întâmpla dacă linia care urmează a fi executată este un apel către o funcție? După ce apăsăm **Step** există două posibilități:

1. Executăm întreaga funcție și ne oprim la linia ce urmează apelului de funcție
2. Executăm doar prima instrucțiune din funcție și ne oprim în funcție pe a doua instrucțiune

Altfel spus urmărim execuția funcției, sau sărim peste ea? Corect ar fi să facem un singur pas (step), adică să executăm următoarea instrucțiune. Cum următoarea instrucțiune este în funcție, corectă ar fi abordarea 2.


De multe ori însă acest lucru nu este de dorit pentru că știm că funcția este corectă deoarece am făcut deja depanarea sa sau este corectă pentru că este o funcție de bibliotecă. În concluzie uneori este de dorit să executăm funcția pas cu pas, alteori nu. Din acest motiv nu există un singur buton pentru execuția pas cu pas, ci trei:

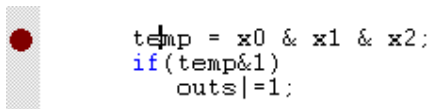
1. **Step into**  – execută prima instrucțiune din funcția apelată și apoi oprește-te.
2. **Step over**  – execută toată funcția ca și cum ar fi o singură instrucțiune și apoi oprește-te.
3. **Step out**  – părăsește funcția. Este activ numai dacă suntem cu execuția în corpul unei funcții diferită de *main*.

Dacă instrucțiunea care urmează a fi executată nu este un apel de funcție „**Step into**” și „**Step over**” au același efect: se execută instrucțiunea următoare.


### Pentru a opri execuția înainte de execuția unei linii de cod .

Se poate întâmpla să vrem ca un anumit grup de instrucțiuni să fie executat la viteză normală, deoarece știm că respectivul grup este corect. Execuția pas cu pas a respectivului grup ar consuma prea mult timp – de exemplu un ciclu „for” cu 1000 de iterații.

Pentru a ne opri înainte de execuția unei anumite linii (instrucțiuni) plasați cursorul pe respectiva linie iar apoi apăsați butonul **Run to Cursor**  . Același efect se obține dacă la începutul liniei pe care vrem să ne oprim este plasat un „breakpoint”. În figura de mai jos linia pe care s-a plasat „breakpoint”-ul este marcată cu un cerc roșu.



```
temp = x0 & x1 & x2;  
if(temp&1)  
    outs|=1;
```

Pentru a insera un breakpoint la începutul unei linii plasați cursorul pe respectiva linie și apăsați butonul  (Toggle Breakpoint). Aceeași procedură permite ștergerea unui breakpoint. Înserarea sau îndepărtarea unui breakpoint se poate face și cu click dreapta; din meniul contextual apărut sa va alege **Toggle Breakpoint**. Într-un program se pot plasa mai multe breakpoint-uri.

### Optimizări

Compilatorul **gcc** din mediului integrat AVR Studio, ca orice compilator, face optimizări. În consecință este posibil ca să nu se genereze cod mașina pentru orice instrucțiune din C. Depanarea codului optimizat este greoaie, deoarece în modul pas cu pas anumite zone din codul sursă C vor fi sărite. În Visual Studio sau în CodeBlocks sunt create două soluții: **Debug** și **Release**. Dezvoltarea se face pe varianta Debug deoarece în această varianta optimizările sunt dezactivate și se generează cod mașină pentru orice instrucțiune C. Evident, codul rezultat va fi mai mare decât în varianta cu optimizări dar depanarea se face cu ușurință. Varianta cu optimizări este varianta Release și este folosită în final, după ce dezvoltarea s-a terminat.

În AVR Studio nu există debug și release dar puteți dezactiva optimizările. Controlul optimizării este în fereastra **Project** → **Configurations Option** → **Optimization**. Codul nu se optimizează dacă se alege opțiunea **-O0**. În mod obișnuit folosiți opțiunea **-Os**.

**Important!** După ce schimbați nivelul de optimizare din meniul **Build** executați **Rebuild All**.

Dacă este ceva neclar, **chemați profesorul**.

### Pasul 7: Votare

Se cere implementarea unei noi funcții  $f_3$  care este ,1' când numărul de variabile de intrare care au valoarea ,1' este mai mare ca numărul de variabile de intrare care au valoarea ,0', adică dacă este majoritate.

**Analizați ambele variante de implementare** și alegeți-o pe cea mai simplă.

Adăugați o nouă pereche rezistența-LED pentru afișarea acestei funcții. **Funcțiile  $f_{2:0}$  deja implementate trebuie să funcționeze ca înainte**. Cu alte cuvinte, funcționalitatea deja implementată trebuie păstrată.

Pentru a modifica montajul, trebuie executată secvență exact în **ORDINEA DE MAI JOS**:

1. Se oprește depanatorul.

2. Se oprește JTAG ICE.
3. Se oprește sursa Hameg.

După ce ați efectuat schimbările, conectați în ordine inversă deconectării.

Când funcționează, **chemați profesorul pentru validare!!**

**Dacă ați ajuns aici veți primi între 1 și 2 puncte, în funcție de implementarea sau nu a variantei optime.**

### Pasul 8: A IV-a intrare

Se cere adăugarea unei noi variabile de intrare și anume  $x_3$ . Sistemul va calcula 4 funcții ce depind de 4 variabile:  $x_3x_2x_1x_0$ . Funcțiile  $f_{3:0}$  trebuie să funcționeze după aceeași logică ca mai înainte:

- $f_0$  este ,1' când numărul de ,1'-uri din  $X = x_3x_2x_1x_0$  este impar.
- $f_1$  este ,1' când  $X < 3$
- $f_2$  este ,1' când numărul  $X$  este un număr prim.
- $f_3$  este ,1' când numărul de variabile de intrare care au valoarea ,1' este mai mare decât numărul de variabile de intrare care au valoarea ,0'.

Când funcționează, **chemați profesorul pentru validare!!**

**Dacă ați ajuns aici veți mai primi între 1 și 3 puncte, în funcție de calitatea implementării.**

### Pasul 9: Deconectarea

La finalul activității de laborator, **executați secvență de deconectare:**

1. Se oprește execuția programului.
2. Se oprește depanatorul.
- 3. Se șterge programul din flash.**
4. Se închide AVR Studio.
5. Se oprește JTAG ICE.
6. Se oprește sursa Hameg
7. Se demontează **NUMAI** partea de montaj creată în cadrul acestui laborator.