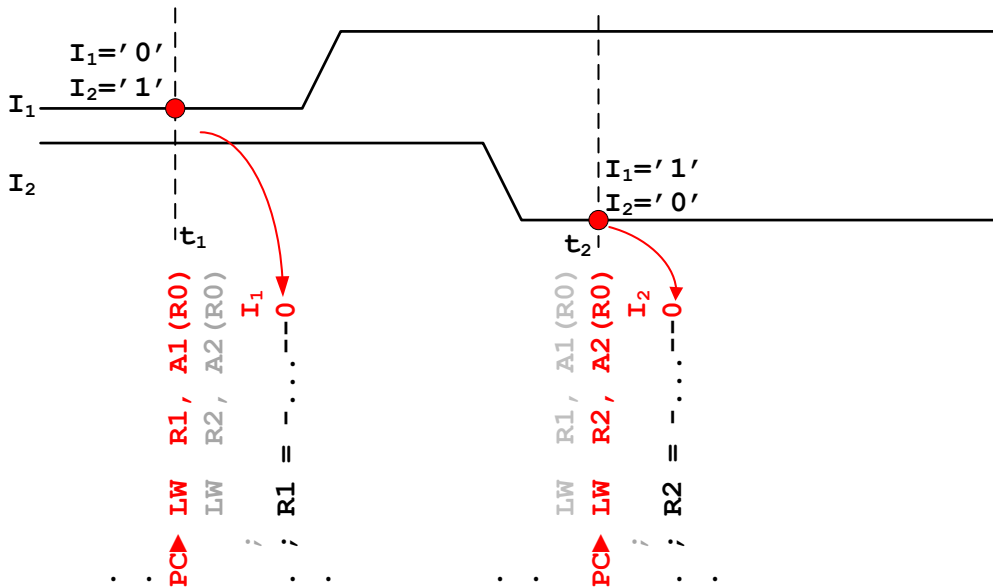


LABORATOR 3 - Emularea software a SLC-urilor

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

Pe bitul 0 al portul de intrare de 1 bit cu adresa A1 este conectată intrarea I1 iar pe bitul 0 al portului de intrare de 1 bit cu adresa A2 este conectată intrarea I2. Deoarece cele două intrări sunt conectate fiecare la portul propriu, sunt necesare două citiri pentru memorarea valorilor lui I1 și I2:

```
LW    R1, A1(R0)
LW    R2, A2(R0)
```

La momentul t1 intrările au valorile I1='0' și I2='1'. Ca urmare pe bitul 0 din R1 se va memora valoarea logică a lui I1, adică ,0'. Până se execută al doilea LW, I1 și apoi I2 evoluează ca în figura de mai sus.

La momentul t2 când se execută LW R2, A2(R0) intrările au valorile logice I1='1' și I2='0'. Ca urmare pe bitul 0 din R2 se va memora valoarea logică a lui I2, adică ,0'. Restul programului „va crede” că I1='0' și I2='0', deși intrările nu au avut niciodată aceste valori în intervalul de timp din figură.

Soluția este ca toate intrările să fie citite în același moment. Cel mai simplu este ca I1 și I2 să fie citite prin intermediul aceluiași port. Această soluție complică un pic programul de emulare, așa cum se va vedea în continuare, dar micșorează numărul de porturi.

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 și execută un program următoarea structură:

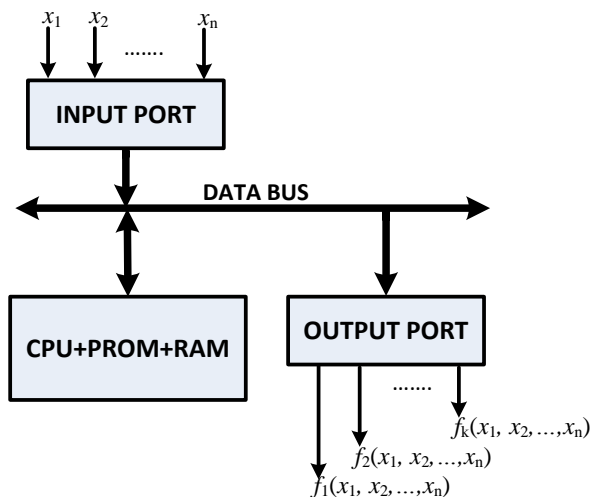


figura 1

La nesfârșit

- Citește valoarea variabilelor de intrare x_1, x_2, \dots, x_n prin intermediul portului de intrare și memorează-le.
- 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: Metoda analitică 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

bit 7 6 5 4 3 2 1 0
 $B = v_7 v_6 v_5 v_4 v_3 v_2 v_1 v_0$

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

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

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:

bit 7 6 5 4 3 2 1 0
 $B = v_7 v_6 v_5 v_4 \text{ } \mathbf{v_3} v_2 v_1 v_0 \quad \text{OR}$
 $M = \underline{0 \ 0 \ 0 \ 0} \text{ } \mathbf{1} \underline{0 \ 0 \ 0} \quad =$
 $\quad \quad \quad v_7 v_6 v_5 v_4 \text{ } \mathbf{1} v_2 v_1 v_0$

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 **0b000000000010000000000000000000** 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 0b000000000010000000000000000000, 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	v_3	v_2	v_1	v_0
						AND			
M	=	m_7	m_6	m_5	m_4	m_3	m_2	m_1	m_0
		v_7	v_6	v_5	v_4	0	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:

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	AND
M =	1	1	1	1	0	1	1	1	=
	v ₇	v ₆	v ₅	v ₄	0	v ₂	v ₁	v ₀	

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 reșetați 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 resetăm 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:

bit	7	6	5	4	3	2	1	0	
B =	v ₇	v ₆	v ₅	v ₄	v₃	v ₂	v ₁	v ₀	AND
M =	0	0	0	0	1	0	0	0	=
	0	0	0	0	v₃	0	0	0	

Dacă 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'

Recomandare:

Deoarece frecvent pot apare confuzii între operațiile de setare, resetare, inversare și testare bit (ce operator? ce mască) se recomandă utilizare macrodefinițiilor. De exemplu, dacă am declara următoarele macro:

```
#define setbit(byte, bit_n0) ...
#define resetbit(byte, bit_n0) ...
#define testbit(byte, bit_n0) ...
```

În loc de

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

se scrie

```
resetbit(B, 3);
```

Ar fi mult mai clar, mai ales dacă este nevoie de resetare în mai multe locuri din program.

Despre macrodefiniții citiți la

<https://www.cprogramming.com/tutorial/cpreprocessor.html>

capitolul Macros.

Nu este obligatoriu să folosiți macrodefiniții dar este recomandabil.

La examenul final dacă folosiți setbit, resetbit, testbit fără să le definiți veți fi depunctați.

Chestiuni teoretice: porturile digitale la microcontrolerul ATmega16

Este obligatoriu să citiți și să 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 funcții booleene notate f_0, f_1 și f_2 specificate 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 B iar funcțiilor de ieșire $f_{2:0}$ se vor trimite în exterior prin portul A, 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

Creați în simulIDE circuitul din figura 2.

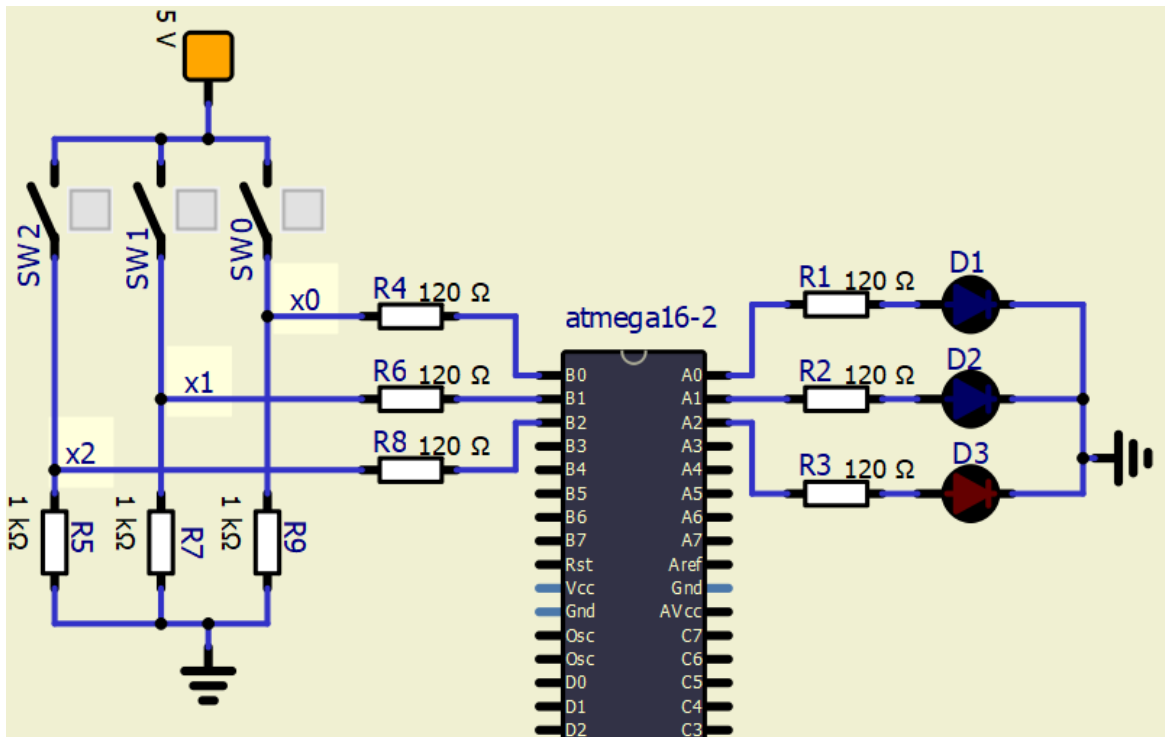


figura 2

Fiecare variabilă de intrare va fi implementată prin intermediul unui comutator. Conform schemei, când un comutator este închis 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.

În figura toate componentele au identificatorii vizibili. Identificatorii sunt R1, R2, R3,...R9, SW1, SW2, SW3, etc. Identificatorii sunt necesari pentru ca în continuarea laboratorului să putem referi cu ușurință componentele. De exemplu, pentru a ne referi la a doua rezistență din dreapta, de sus în jos vom folosi ID-ul sau și anume R2. Id-urile sunt necesare numai în platforma de laborator. **Studentii nu trebuie să adauge ID-uri** pe schemele pe care le fac în timpul laboratorului.

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 rezistențe ar putea fi eliminate și în locul lor ar trebui montate fire. Cum sunteți începătorii și începătorii fac multe greșeli, OBLIGATORIU montați aceste rezistențe. Se poate argumenta că în simulare nu contează și așa și este, dar dacă vreodată vom folosi schema din figura 2 ca referință pentru un montaj real, atunci este bine să ne obișnuim să montăm rezistențele R4, R6 și R8.

Alegeți următoarele culori pentru LED-uri: D1 – roșu, D2 – portocaliu, D3 – verde.

În continuare scrieți codul C.

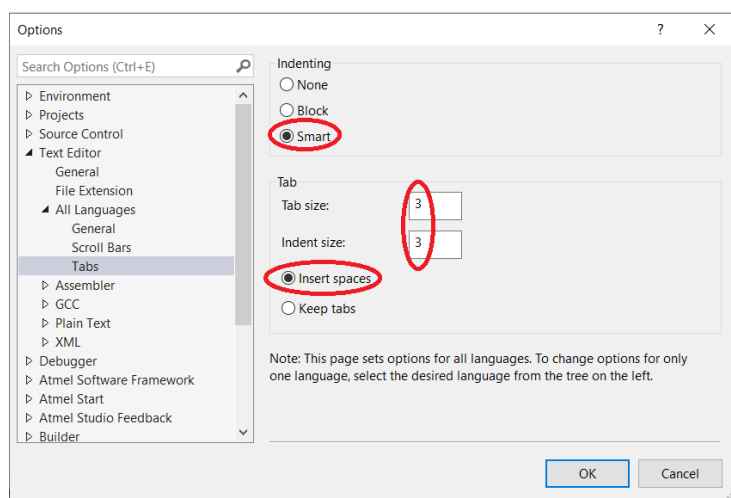
Pasul 2: Crearea proiectului

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

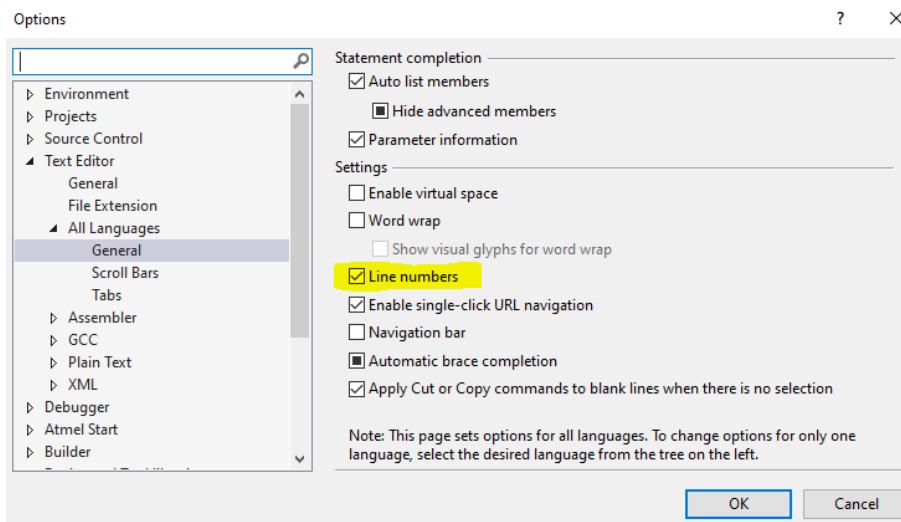
1. Se lansează în execuție **Atmel Studio 7**.
2. Din pagina de start executați **New Project...**
3. În fereastra New Project alegeți proiect din categoria **C/C++** de tip „**GCC C Executable Project**”, bifați checkbox-ul **Create directory for solution**, stabiliți locația proiectului și numele proiectului. În această lucrare de laborator numele proiectului va fi **funcții**. În final apăsați butonul **OK**; va apare fereastra **Device Selection**.
4. În fereastra **Device Selection** selectați **ATmega16A** și apoi apăsați **OK**.
5. Redenumiți **main.c** ca **funcții.c**.

Pasul 3: Crearea codului sursă

Pentru o editare mai ușoară a sursei, în fereastra **Tools → Options → Text Editor → All Languages → Tabs** faceți următoarele setări:



Pentru a referii mai ușor o anumită linie din codul sursa, în fereastra **Tools → Options → Text Editor → All Languages → General** activați afișarea numărului liniei:



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

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

int main() {
    // memoreaza valorile semnalelor de intrare
    unsigned char inputs;

    // Variabila vxi, (i=0,1,2) memoreaza in bitul i valoarea intrarii xi.
    // Ceilalti biti sunt 0.
    unsigned char vx2, vx1, vx0;

    // 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, în special paginile 6-10, începând cu figura 8.

Configurați toți pinii A ca ieșiri:

```
DDRA = 0b????????;
```

Configurați toți pinii B ca intrări:

```
DDRB = 0b????????;
```

Atenție mare la stabilirea direcției porturilor! Dacă nu sunt montate rezistențe de protecție în serie cu intrarea (R, R6, R8 în figura 2) microcontrolerul **se poate distruge** dacă un port este folosit ca IN de hardware dar este declarat ca OUT în software.

```
while(1) {
```

Citirea pinilor unui port se face cu instrucțiunea:

```
Nume_variabila = PINX; //X se înlocuiește cu A, B, C sau D;
```

unde *Nume_variabila* este o variabilă de tip `unsigned char`. Cine este portul intrare? Oricine ar fi, toate porturile sunt definite în headerul **io.h**, motiv pentru care acesta a fost inclus în fișierul sursă. Valorile semnalelor de intrare x_2, x_1, x_0 se citesc **simultan** din portul de intrare.

```
//memorează valorile semnalelor de intrare
//bit      7654 3 2 1 0
//inputs =      x2 x1 x0
inputs = PINX;
```

Mascarea la ,0' a biților cu valoare nedeterminată. Atenție la următorul aspect: citirea portului de intrare generează o valoare pe 8 biți, dar numai 3 biți au o valoare bine determinată. Numai ultimii 3 biți se folosesc. Aceștia sunt biții 2:0 care sunt conectați fie la ,0' fie la ,1' prin comutatoare. Valoarea celorlalți 5 biți este nedeterminată. Astfel citirea portului de intrare generează valoarea:

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

Caracterul ,-, înseamnă valoare nedeterminată. Pentru a elimina influența biților cu valoare nedeterminată, aceștia trebuie **resetați** (forțați la ,0' sau mascați la ,0'). Resetați biții cu valoare nedeterminată din variabila *inputs*. Operația de resetare a fost descrisă anterior:

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

Mai sus s-au mascat biții 7:3

```
inputs = inputs operator(care ?) masca(ce valoare?);
```

Deoarece în C standard nu există tipul bit sau boolean, ca în VHDL, pentru fiecare variabilă care ar trebui să fie de tip boolean sau bit se va alocă un caracter. Bitul zero al acestui caracter va memora variabila booleană iar ceilalți 7 biți se vor forța la zero. Această reprezentare o vom numi bit pe byte și **are formatul 0000_000bit**.

Conform celor spuse mai sus, valorile semnalelor x_2, x_1, x_0 se vor memora în variabile vx_2, vx_1, vx_0 . Variabilele vx_2, vx_1, vx_0 se creează din variabila *inputs* după cum urmează:

```
inputs
bit 7654 3 2 1 0
0000 0 x2 x1 x0 →
    |
    |
    |
    L-----→
    |
    |
    |
    L-----→
                                vx0
                                bit 7654 321 0
                                0000 000 x0
                                vx1
                                bit 7654 321 0
                                0000 000 x1
                                vx2
                                bit 7654 321 0
                                0000 000 x2
```

Se observă că variabila vx_0 se poate genera din variabila *inputs* dacă se resetează toți biții acesteia, mai puțin bitul 0:

```
// vx0 = 0000 000x0; Variabila vx0 conține numai intrarea x0
vx0 = 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 dreapta pe *inputs* și apoi să resetăm biții 7..1.

În general, **pentru a crea variabila vx_i din bitul x_i** este nevoie de i deplasări la dreapta urmate de resetarea biților 7..1.

Atenție! În procesul de calculare a lui v_{x1} și v_{x2} **NU** modificați pe `inputs`! Vom avea nevoie de ea mai târziu.

```
// vx1 = 0000 000x1  
vx1 = inputs...;
```

Pentru v_{x2} procedați asemănător:

```
// vx2 = 0000 000x2  
vx2 = 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 (oare?) 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 foarte simplu:

```
// f0 se calculeaza in temp  
temp = vx2 ^ vx1 ^ vx0;
```

Apoi `temp` este folosit pentru a seta bitului 0 din `outs`:

```
if( temp & 1<<0) // if( testbit(temp, 0) dacaati definit testbit  
    outs = ??; //seteaza bitul 0 din 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` deoarece 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 variabilă 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. 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 operatori booleeni. Dispunem de un întreg ALU! De ce să nu-l folosim?

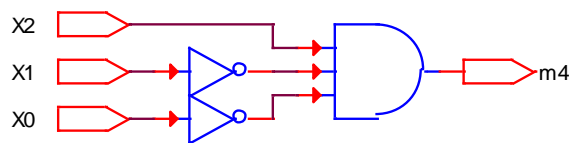
Chestiuni teoretice: Emularea funcțiilor logice

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 se face cu

```
m4= (x2 & ~x1 & ~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 (4== inputs)
    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 m_4 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**.

Dacă putem emularea mintermenii, la fel de simplu se poate implementa **forma canonică**. Funcția f_1 este compusă din mintermenii 1, 2 și 3. Forma canonică disjunctivă a lui f_1 este $f_1 = \sum(0,1,2) = m_0 + m_1 + m_2$. Forma canonică disjunctivă este echivalentă cu a spune că f_1 este '1' dacă $X = 0$ sau $X = 1$ sau $X = 2$. 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 = ...; //setează 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 aritmetici, 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 preț de cost diferit. Operatorii ==, !=, >, < necesită pe majoritatea procesoarelor o singură instrucțiune cod mașină. În schimb operatorii >=, <= necesită de regulă două instrucțiuni, așa că în loc de `var >= 2` folosiți `var>1`.

Ultima funcție care se implementează este f_2 . Procedați după cum urmează:

- Aflați forma minimă pentru f_2 .
- Găsiți o implementare cu operatori relaționali și logici.
- Alegeți dintre forma minimă și varianta cu operatori relaționali și logici varianta cea mai simplă. **Veți explica profesorului** care variantă este mai simplă, ceea ce înseamnă că va trebui să aveți ambele variante.

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

```

// dacă ați ales minimizarea
temp = ...;
if( temp & 1<<0)
    outs = ???; //setează bitul 2 din outs

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

```

În final, scrierea variabilei `outs` în portul la ieșire care sunt conectate LED-urile se face cu:

```

PORTA = outs;

```

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

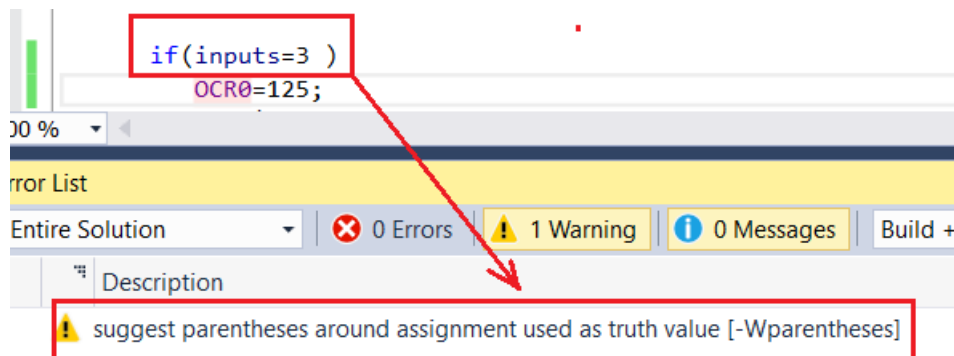
```

    } //end main loop
} //end main

```

Pasul 4: Build

1. **Executați Rebuild Solution** din meniul **Build**. Atmel Studio prezintă o comportare ciudată: dacă modificați codului și executați **Build**, Atmel Studio 7 afișează toate warningurile. Fără să se modifice codul se execută Build încă o dată: de data aceasta avertizările nu mai apar. Din acest motiv executați întotdeauna **Rebuild Solution**.
2. **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 semnul exclamării galben. Ignorarea avertizării ne face să pierdem foarte mult timp cu depănarea. Este mult mai eficient să examinăm toate avertizările și să eliminăm cauzele care le produc în 3 minute decât să pierdem zeci de minute cu depănarea.

Pasul 5: Testarea codului

Încărcați codul din fișierul .hex în microcontrolerul din simulIDE. Procedura a fost explicată în laboratorul precedent

Apoi se fac **toate combinațiile posibile** ale intrărilor și se observă LED-urile. În cazul în care nu funcționează, acesta fiind cazul cel mai frecvent, este necesară depănarea aplicației. Pentru depănare vezi **Anexa 1**.

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

Pasul 6: Bonus

Profesorul va crea tema pentru bonus în Google Classroom după ce primul student va termina secțiunea opțională. **Puteți primi bonus 0.5 puncte. Vezi regulile pentru bonus în pagina cursului.**

Anexa 1: Depanarea programelor

În cazul **simulIDE** depanarea este greoaie. În acest caz mijloacele de depanare sunt:

1. Oprirea și pornirea manuală (fără breakpoint) a programului.
2. Vizualizarea registrelor IO și a variabilelor

Pentru a opri execuția programului

Oprirea execuției se face cu butonul **Pause Simulation** din figura următoare:

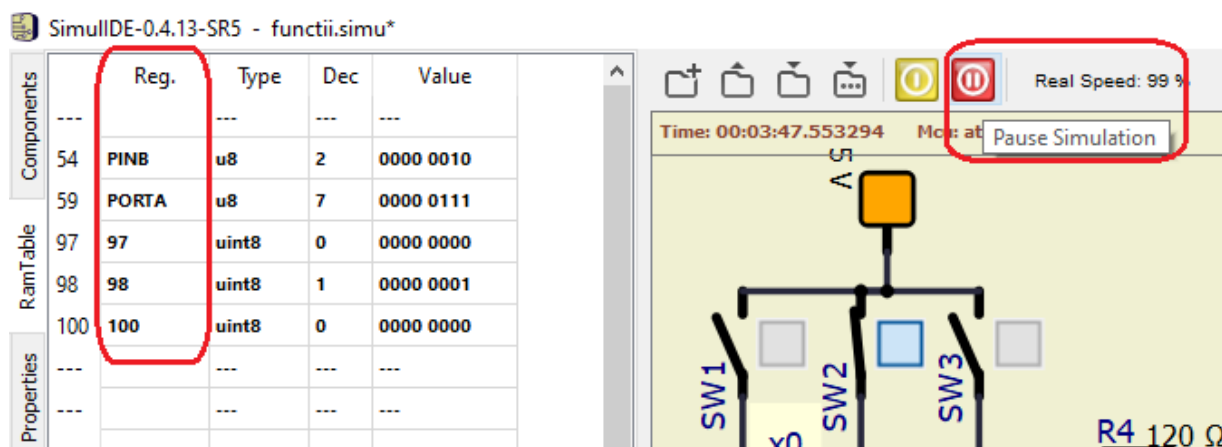


figura 3

Pentru a reporni simularea se apasă din nou butonul **Pause Simulation**.

Evident, oprirea și repornirea simulării se face numai cu circuitul alimentat.

Vizualizarea stării registrelor IO și a variabilelor.

Dacă execuția programului este oprită, se poate vizualiza starea oricărui registru IO. În coloana Reg. din figura 3 introduceți numele registrului a cărui valoare vreți să o aflați: DDRA, PORTA, DDRB, etc.

Se poate vizualiza și starea variabilelor dacă sunt îndeplinite anumite condiții. Prima condiție este ca variabila să nu fie eliminată de compilator ca urmare a optimizărilor. Citiți în continuare pentru a înțelege cum se modifică nivelul de optimizare.

Optimizările făcute de compilator


Compilatorul **gcc** din mediului integrat AVR Studio face optimizări. În consecință este posibil ca să nu se genereze cod mașină pentru orice instrucțiune din C. Depanarea codului optimizat este greoaie, aproape imposibilă, deoarece anumite zone din codul sursă C vor lipsi.

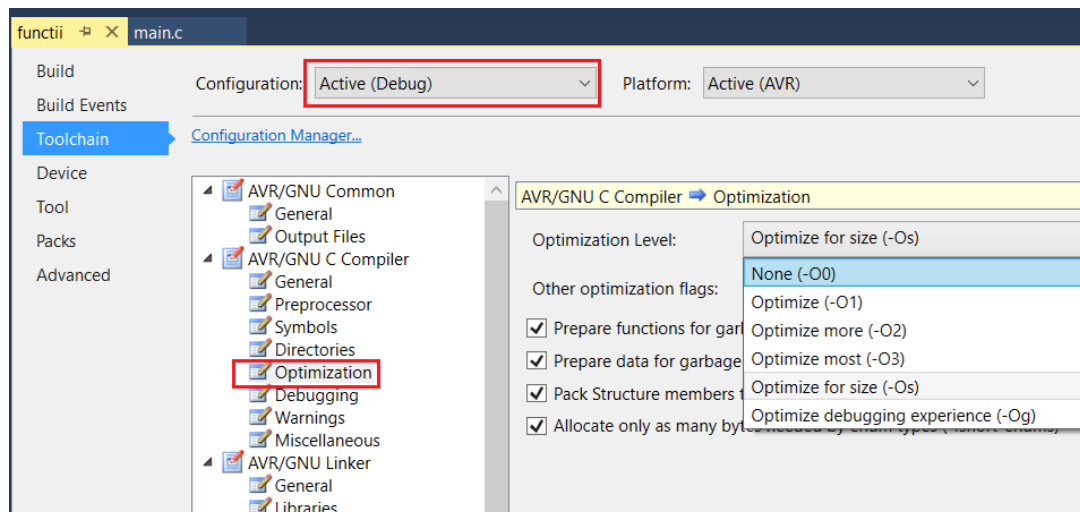
În Studio sunt create două soluții: **Debug** și **Release**. Pentru fiecare soluție se poate seta nivelul de optimizare. De regulă, pentru **Debug** optimizările sunt dezactivate iar pentru **Release** sunt setate la cel mai înalt nivel.

Dezvoltarea se va 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 **Release** este folosită în final, după ce dezvoltarea s-a terminat.

Deși pentru **Debug** optimizările ar trebui dezactivate iar pentru **Release** ar trebui setate la cel mai mare nivel, nivelul de optimizare se poate seta pentru fiecare soluție în parte. La crearea proiectului nivelul implicit de optimizare este mare și pentru Debug, și pentru Release.

Pentru a seta nivelul de optimizare, în fereastra **Solution Explorer** selectați proiectul (funcții), vizualizați proprietățile acestuia cu  și în fereastra de proprietăți selectați **Toolchain**, ca în figura următoare:



Pentru configurația **Debug** setați nivelul de optimizare la **None (-O0)**. Acum puteți vedea adresele variabilelor.

Adresele variabilelor se pot afla numai dacă variabilele sunt **globale**, altfel alocarea se face pe stivă și nu este fixă.

Adresele variabilelor se găsesc în fișierul **funcții.map**. Acesta se află în același folder ca fișierul .hex. În acest fișier căutați numele variabilei/variabilelor pe care vreți la afișați. În continuare este prezentată secțiunea din fișier în care se află adresele variabilelor vx0, vx1, vx2, etc.

```
...
COMMON      0x00800060      0x6 main.o
              0x00800060      outs
              0x00800061      vx0
              0x00800062      vx1
              0x00800063      temp
              0x00800064      vx2
              0x00800065      inputs
              0x00800066      PROVIDE ( __bss_end, .)
....
```

Ultimele două cifre din coloana a doua reprezintă adresa hexazecimală a variabilei din coloana a treia: vx0 are adresa 0x61, vx1 are adresa 0x62, etc. Aceste adrese se convertesc în zecimal și sunt introduse în coloana Reg. din figura 3: 0x61=97, 0x62=98, etc.

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