LABORATOR 4- Implementarea SLC-urilor, varianta LUT

Implementarea tip ROM a funcțiilor booleene

În laboratorul precedent s-au implementat 3 funcții booleene, conform următoarei tabele 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

Vom discuta modul de implementare a funcției f_1. Funcția f_1 este ‚1' pentru mintermenii cu echivalentul zecimal mai mic decât 3. Implementarea C plus codul în asamblare rezultat în urma compilării este:

```
+00000044:   2729        EOR       R18,R25        Exclusive OR
25:             if(inputs<3 )
+00000045:   3033        CPI       R19,0x03       Compare with immediate
+00000046:   F408        BRCC      PC+0x02        Branch if carry cleared
26:                  outs|=1<<1;
+00000047:   6022        ORI       R18,0x02       Logical OR with immediate
28:             if(inputs...)
```

Implementarea necesită 3 instrucțiuni în cod mașină și ocupă 6 octeți în memoria de cod. Fără nici un dubiu, este cea mai scurtă implementare cu putință pentru f_1 prin metoda analitică.

În laboratorul precedent pentru emularea software a funcțiilor logice s-a folosit și minimizarea. Minimizarea a fost studiată în cadrul cursului BLPC (DSD). Această metodă nu este însă singura; o altă metodă de implementare a funcțiilor logice se bazează pe memorii ROM. Această metodă nu necesită minimizare și varianta sa hardware s-a studiat la „Organizarea calculatoarelor".

Această idee poate fi folosită în cazul emulării circuitelor logice combinaționale: se scrie într-un vector tabela de adevăr a funcției/funcțiilor care trebuie implementate iar apoi intrările se folosesc ca **index** în această tabelă. Astfel de tabele se numesc tabele de căutare (lookup table =LUT). Implementarea LUT pentru calculul celor 3 funcții din laboratorul precedent este:

```
//Acesta este un exemplu! Nu copiați acest cod.
#include <avr/io.h>
const unsigned char fnLUT[]={

   0b010, //0
   0b011, //1
   0b111, //2
   0b100, //3
//------------
   0b001, //4
   0b100, //5
   0b000, //6
   0b101  //7
};

int main(){
   unsigned char inputs;

   DDRA=0xff;
   DDRB=0;

   while(1){
      inputs = PINB & 0b00000111;
      PORTA  = fnLUT[inputs];
   }
}
```

Se observă că valorile funcțiilor $f_{2:0}$ au fost memorate folosind un octet pentru fiecare valoare a intrării, dar din fiecare octet s-au folosit doar 3 biți. Fără a crește spațiul de memorare, folosind mai mulți biți din cei 8 disponibili, se pot implementa până la 8 funcții de 3 variabile. Este evident că această variantă este mult mai avantajoasă: pentru 8 funcții de 3 variabile sunt necesari 8 octeți pentru tabelă și 8 octeți pentru operația de indexare. Cei 16 octeți sunt alocați în spațiul de cod. Metoda analitică, la un consum de 6 octeți per funcție, ar necesita 6*8=48 de octeți. Un alt avantaj îl constituie faptul că funcțiile nu mai trebuie minimizate; a minimiza 8 funcții necesită un timp neneglijabil, dacă această operație se face manual.

Metodă tabelei de căutare (LUT) funcționează pentru un număr relativ mic de variabile de intrare. Dacă metoda s-ar aplica pentru funcția AND de 10 variabile dimensiunea tabelei de căutare ar fi de 1024 de octeți. În schimb metoda analitică necesită numai 20 octeți.

În principiu metoda LUT se aplică dacă numărul de intrări este relativ mic.

Să presupunem că un anumit proiect necesită calculul funcției sin(x), unde x este o intrarea reprezentată pe 8 biți. Funcția sin(x) se calculează prin dezvoltare în serie Taylor și evident necesită calcule în virgulă mobilă. Cum ATmega nu dispune de hardware specializat pentru calcule în virgulă mobilă, va fi necesară includerea unei biblioteci pentru astfel de operații. Biblioteca de virgulă mobilă are o dimensiune considerabilă! În acest caz este mult mai simplu să se precalculeze sin(x) și să se memoreze valorile într-o tabelă de căutare implementată cu un vector. În acest caz 256 sau 512 octeți pentru tabelă este mult mai bine decât câțiva kiloocteți pentru biblioteca de virgulă mobilă.

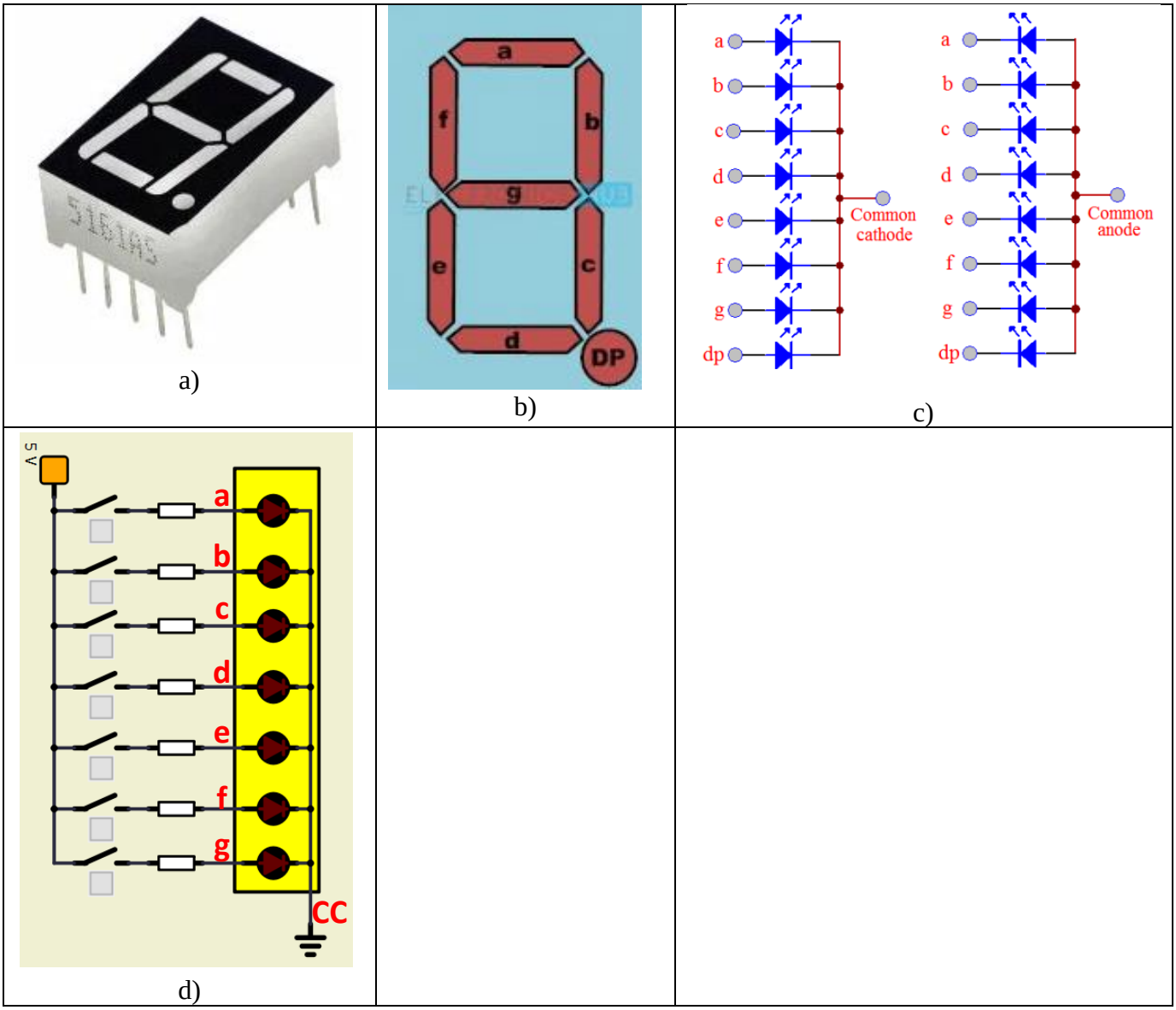
Alegerea metodei de implementare a funcțiilor booleene depinde de numărul de variabile de intrare, de numărul de funcții de implementat și de complexitatea acestora. NU există rețete prestabilite și numai proiectantul (adică dumneavoastră) poate să facă alegerea.

Atenție: chiar dacă prezentarea din acest laborator și din cel precedent accentuează importanta obținerii celui mai scurt cod, în practică mai există un factor de care trebuie ținut seama: respectarea timpului alocat dezvoltării – **time to market**. Degeaba am obținut codul cel mai scurt dacă am ratat termenul de predare al proiectului! În practică o implementarea cu 10% mai lungă decât implementarea perfectă este admisă. De exemplu, la ATmega16 dimensiunea memoriei ROM este 16KB; nu va exista

nici o diferență din punct de vedere al prețului între o implementare de 10KB și una de 11KB. Va conta însă o diferență între 16KB și 17KB

Afișor 7 segmente cu LED

În acest laborator se va folosi un afișor 7 segmente cu LED. Afișorul 7-segmente cu LED este alcătuit din șapte sau opt LED-uri aranjate ca în figura 1a:



a) b) c) d)

figura 1

Fiecare dintre cele șapte LED-uri se numește segment, deoarece atunci când este aprins segmentul face parte din cifra care trebuie afișată. De cele mai multe ori afișorul este prevăzut cu un al optulea LED ce permite indicarea punctului zecimal (DP). Punctul zecimal este folosit atunci când două sau mai multe afișaje cu 7 segmente sunt folosite împreună pentru a afișa numere cu virgulă.

Fiecare segment este identificat printr-o literă între **a** și **g**, ca în figura 1b. Al optulea segment este notat **DP**.

Deoarece este alcătuit din 8 LED-uri afișorul ar trebui să aibă 2x8=16 pini. Pentru a micșora numărul de pini 8 pini sunt conectați împreună, ca în figura 1c. In funcție modul în care pinii LED-urilor sunt conectate intern afișoarele 7 segmente sunt de două tipuri:

- Tipul de afișor la care anozii tuturor celor 8 LED-uri sunt conectați împreună se numește „Cu anod comun” (Common Anode - **CA**)

- Tipul de afișor la care catozii tuturor celor 8 LED-uri sunt conectați împreună se numește „Cu catod comun” (Common Cathode - **CC**)

Pentru fiecare tip de afișor există o schema de conectare. In cazul afișorului 7 segmente cu catod comun schema de conectare este prezentată în figura 1d. Acest tip de control a fost folosit pentru un singur LED în laboratorul 1 și pentru 3 LED-uri în laboratorul 2. În figura 1d repetăm de 7 ori schema de control folosită pentru un singur LED.

Scopul lucrării

Se va implementa un decodificator 7 segmente octal. Decodificatorul primește la intrare un număr binar de 3 biți notați x_2 x_1 x_0. Echivalentul zecimal al numărului binar x_2 x_1 x_0 se notează cu X, la fel ca în laboratorul precedent. Afișorul va afișa pe X. Deoarece X are 7 valori, afișorul va afișă una din cifrele 0-7. Configurația segmentelor pentru cele 7 cifre este prezentată în tabelul următor:

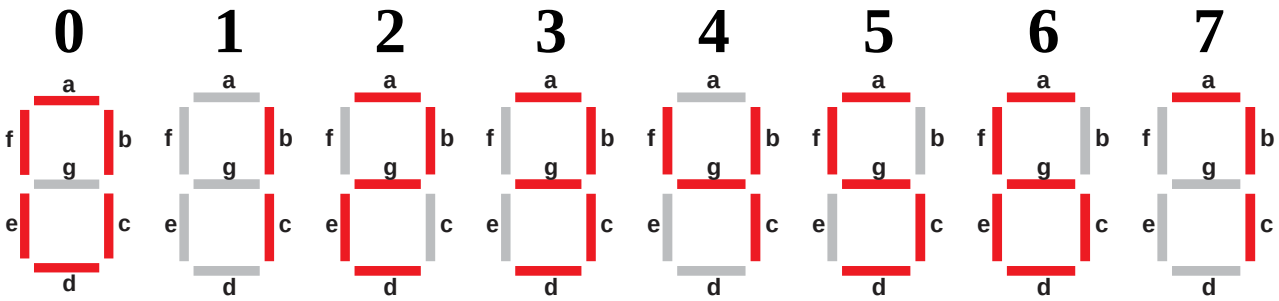


figura 2

Decodificatorul trebuie să funcționeze conform următoarei tabele de adevăr:

Table 1

Intrarea X în baza 10	**Intrarea X în baza 2** x_2 x_1 x_0	**Valori segmente:**							
		Bit 7	6	5	4	3	2	1	0
		-	g	f	e	d	c	b	a
0	**0 0 0**	-	**0**	**1**	**1**	**1**	**1**	**1**	**1**
1	**0 0 1**	-	**0**	**0**	**0**	**0**	**1**	**1**	**0**
2	**0 1 0**	…							
3	**0 1 1**								
4	**1 0 0**								
5	**1 0 1**								
6	**1 1 0**								
7	**1 1 1**								

Completați tabela de mai sus.

Desfășurarea lucrării

Pasul 1: Realizarea montajului

Pentru afișarea cifrelor din Table 1 în **simulIDE** se va folosi componenta „7 segment” din grupul „Outputs”. Selectarea acestei componente este prezentată în figura 3a.

Numele pinilor afișorului este prezentată în figura 3b cu litere roșii. Numele nu sunt atașate simbolului și nu apar pe planșa de desenare. Pinul „Common” poate fi CA sau CC în funcție de tipul afișorului. Mai multe despre componenta „7 segment” se găsește la SimulIDE: Seven Segment Display

Tipul afișorului, culoarea și numărul de afișoare sunt proprietăți ale afișorului și în acest laborator se vor seta ca în figura 3c.

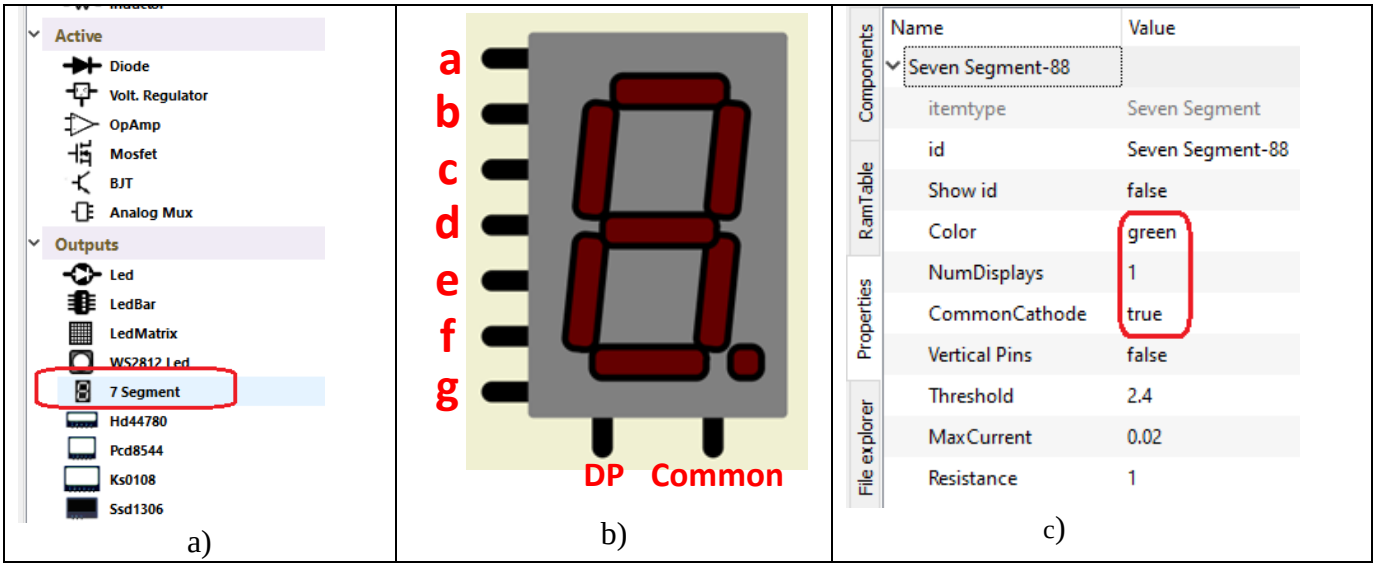


figura 3

Se realizează în simulIDE montajul din figura următoare:

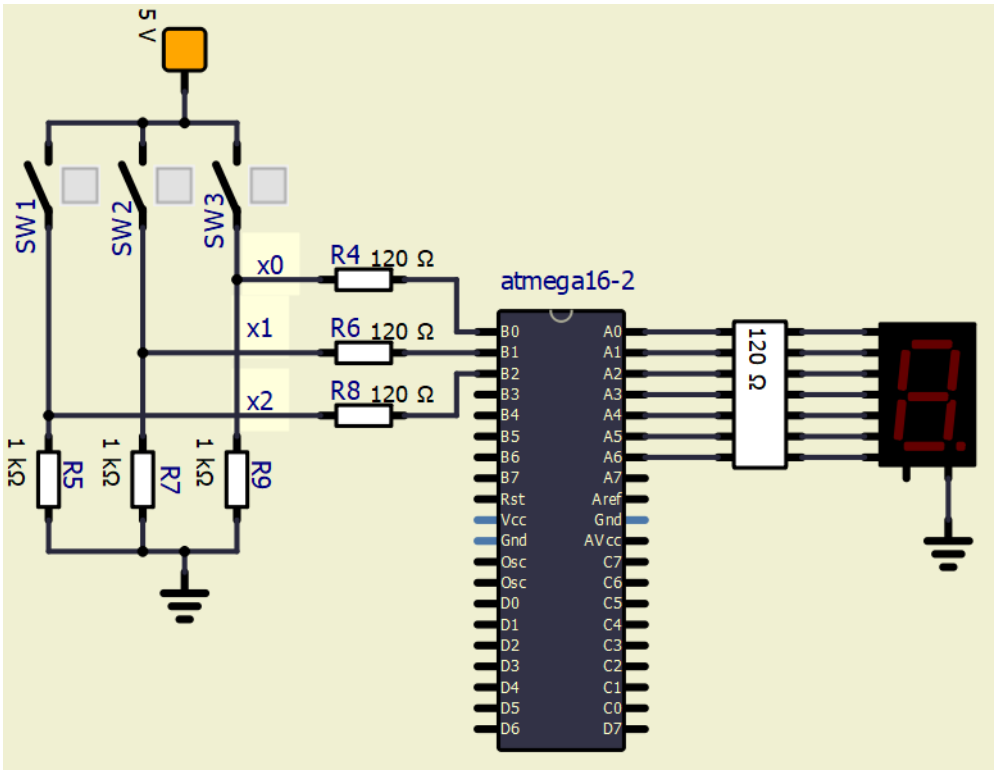


figura 4

Copiați schema din laboratorul „Funcții”, redenumiți-o ca „FuncțiiLUT” și păstrați doar comutatoarele. In acest fel ați desenat jumătate de schemă!

Intrările x_0, x_1, x_2 se vor citi prin intermediul portului B. **Portul B** va fi configurat ca port de **intrare**. Citirea intrărilor $x_{2:0}$ se face ca în laboratorul precedent.

Valorile funcțiilor de ieșire a..g din Table 1 se vor înscrie în PORTA, biții 6..0. **Portul A** va fi configurat ca port de **ieșire**.

Componenta RN1 este alcătuită din 7 rezistențe și este echivalentă cu cele 7 rezistențe din figura 1d. Folosirea rețelei rezistive simplifica atât schema cât si montajul.

Nu uitați, afișorul 7 segmente este de tip **CC** (Comon Cathode).

Pasul 2: Crearea codului sursă și execuția

Pentru a evidenția cum depinde lungimea codului mașină și necesarul de RAM de modul în care este scris codul C, vom crea 3 proiecte:

1. Codul de mai jos reprezintă **cel mai mic proiect funcțional**:

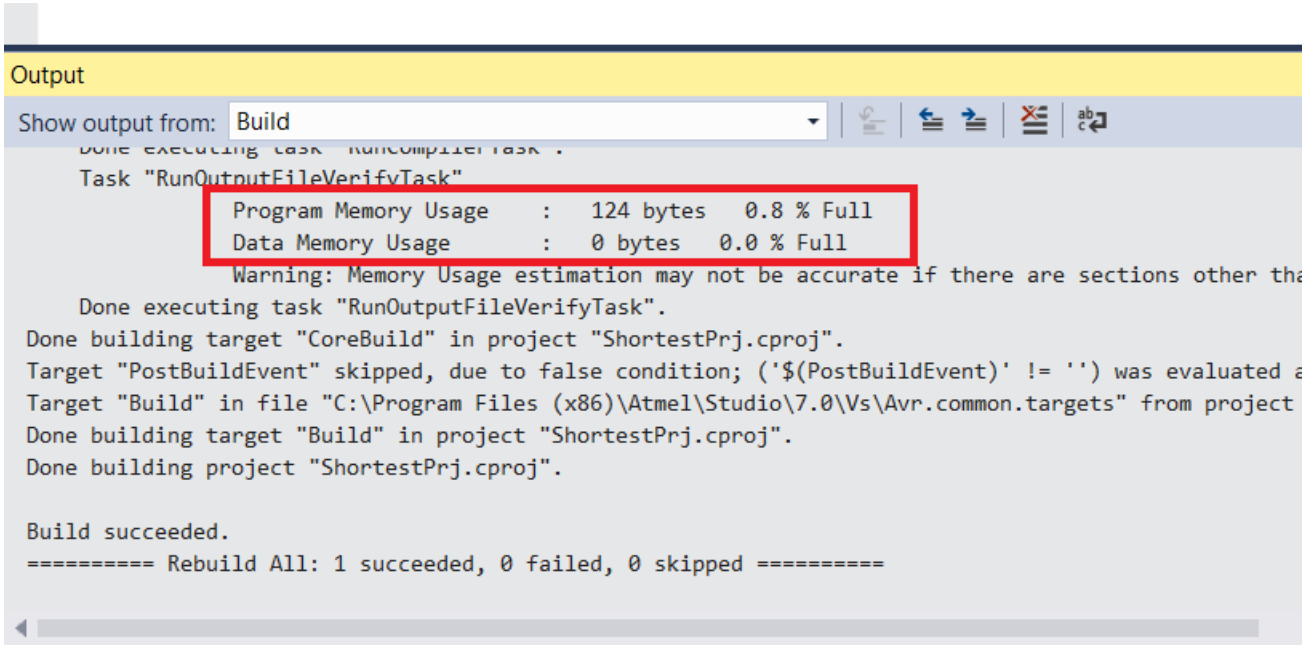
```
#include <avr/io.h>

int main(){
   DDRA=0xFF;
   DDRB=0;

   while(1){
      PORTA = PINB;
   }
}
```

Nu este nevoie să creați sau să executați acest cod.

Dacă se face Build pentru acest cod, vor fi raportate resursele utilizate:



Test va fi folosit în continuare ca termen de comparație.

Construiți Table 2 cu un editor de text (Notepad, Wordpad, MS Word etc.). Resursele utilizate pentru codul anterior a fost înscris în Table 2, linia **Test**.

Table 2

Resurse / Proiect	Program	Data	Δ(Proiect curent– Proiect Test)	
			Δ Program	Δ Data
Test	124	0	124 - 124=0	0 - 0=0
LUT_RAM	p_ram	d_ram	p_ram-124	d_ram
LUT_ROM	p_rom	d_rom	p_rom-124	d_rom
SWITCH	p_sw	d_sw	p_sw-124	d_sw

Deși programul este foarte scurt, se observă că secțiunea **Program** este destul de mare. Acest fapt se datorează inițializărilor pe care le adaugă compilatorul pentru orice program.

2. **Creați un nou proiect cu numele LUT_RAM** conform pașilor prezentați în laboratoarele anterioare.

Implementați decodificatorul octal conform indicațiilor din secțiunea **Implementarea tip ROM a funcțiilor booleene**. Codul va avea aceeași structura ca exemplul de la pagina 2. Tabela de căutare se va numi segLUT în loc de `fnLUT`.

Nu uitați! Citirea portului de intrare generează o valoare pe 8 biți, dar numai 3 biți au o valoare bine determinată. Mascați la ‚0' biți nefolosiți.

Verificați dacă funcționează! Se fac toate combinațiile posibile ale intrărilor și se observă afișorul.

Notăm cu **p_ram** lungimea în octeți a secțiunii Program și cu **d_ram** lungimea în octeți a secțiunii Data. Înlocuiți **p_ram** și **d_ram** în Table 2.

Dacă funcționează conform tabelei de adevăr din Table 1 **apelați profesorul pentru înregistrarea progresului!**

3. Varianta de la punctul 2 folosește memorie RAM pentru a memora tabela de căutare. Această soluție este rar admisă deoarece risipește RAM valoros: avem 16KB de ROM și numai 1KB de RAM. Mai mult, tabela se află inițial în ROM și este transferată în RAM în faza de inițializare. Faza de inițializare este adăugată de compilator. Nu uitați, nu avem sistem de operare care să facă încărcarea programului.

   În concluzie varianta RAM necesită cod suplimentar pentru transferul tabelei și irosește resurse RAM, deși codul se află deja în ROM. Soluția evidentă este să lucrăm cu tabela din ROM. În continuare vom crea o astfel de variantă.

   Creați un nou proiect cu numele LUT_ROM. Codul C pentru această variantă este foarte asemănător cu cel al variantei LUT_RAM, cu trei diferențe. Pentru ca tabela de căutare să fie creată în ROM vom include și headerul **pgmspace** cu:

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

   Apoi vom modifica declarația tabelei de căutare prin adăugarea specificatorului **PROGMEM**:

   ```
   const unsigned char segLUT[] PROGMEM ={…….
   ```

   Atenție: `segLUT[]` trebuie să fie globală.

   În final vom modifica codul de la punctul 2:

   In loc de:

   ```
   PORTA = segLUT[inputs];
   ```

   se va folosi

   ```
   PORTA = pgm_read_byte (&segLUT[inputs]);
   ```

   Dacă sunteți curioși, informații suplimentare despre funcțiile din biblioteca pgmspace se află la **https://teslabs.com/openplayer/docs/docs/prognotes/Progmem%20Tutorial.pdf**

   Faceți **Build**. Notați cu **p_rom** lungimea în octeți a secțiunii Program și cu **d_rom** lungimea în octeți a secțiunii Data. Ar trebui ca lungimea zonei **Data** să fie 0 octeți. Înlocuiți **p_rom** și **d_rom** în Table 2.

   Apoi se verifică funcționarea: se fac toate combinațiile posibile ale intrărilor și se observă afișorul. Verificați dacă funcționează conform tabelei de adevăr din Table 1.

   Deocamdată **NU** chemați profesorul pentru validare!

4. Există tendință printre studenți ca în loc de LUT sa să emuleze decodificatorul cu `if` sau `switch`. **Creați proiectului cu numele PSWITCH**. Implementați decodificatorul cu `switch`, sau if.

   Faceți **Build**. Notaţi cu **p_sw** lungimea în octeți a secțiunii Program şi cu **d_sw** lungimea în octeți a secțiunii Data. Înlocuiți **p_sw** și **d_sw** în Table 2.

   Comparați cele 3 variante de implementare! Care este cea mai bună? Dacă varianta cea mai scurtă ia punctajul maxim, **ce notă ar trebui să ia o implementare cu switch sau if? Evident, a patra parte din punctajul maxim!**

   Nu mai verificați funcționarea.

   Dacă implementarea LUT_ROM de la punctul 3 este funcțională și Table 2 este completat, reîncărcați proiectul LUT_ROM și **apelați profesorul pentru validare.**