

## 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  $f_1$ . Din tabelă se observa că funcția 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 puțință 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”.

Să presupunem că se dispune de o memorie cu organizarea 8x3 în care se înscrie tabela de adevăr din tabelul de mai sus. Dacă variabilele de intrare  $x_{2:0}$  se conectează la adresele  $A_{2:0}$ , la ieșirile  $D_{2:0}$  se vor obține cele 3 funcții.

Această idee poate fi folosită în cazul emulării circuitelor logice combinaționale: se generează în memorie tabela de adevăr 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[]={

    0b110, //0
    0b111, //1
    0b111, //2
    0b100, //3
    //-----
    0b001, //4
    0b100, //5
    0b000, //6
    0b001  //7
};

int main(){
    unsigned char inputs;

    DDRA=0xff;
    DDRB=0;

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

        // secventa anterioara este echivalenta cu:
        // outs=segLUT[PINB & 0b00000111];
        // dar este mai greu de depanat
        // și codul generat este la fel de lung
    }
}

```

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  $6 \cdot 8 = 48$  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.

Metoda tabeli 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 tabeli de căutare ar fi de 1024 de octeți. În schimb metoda analitică necesită numai 18 octeți.

**În principiu metoda LUT se aplică dacă numărul de intrări este relativ mic sau dacă timpul de calcul a forme analitice este prea mare.**

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 importanța 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 implementare 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!

### Scopul lucrării

Se va implementa un decodificator 7 segmente octal. Multipolul primește la intrare un număr binar de 3 biți. Configurația segmentelor pentru cele 7 cifre este prezentată în tabelul următor:

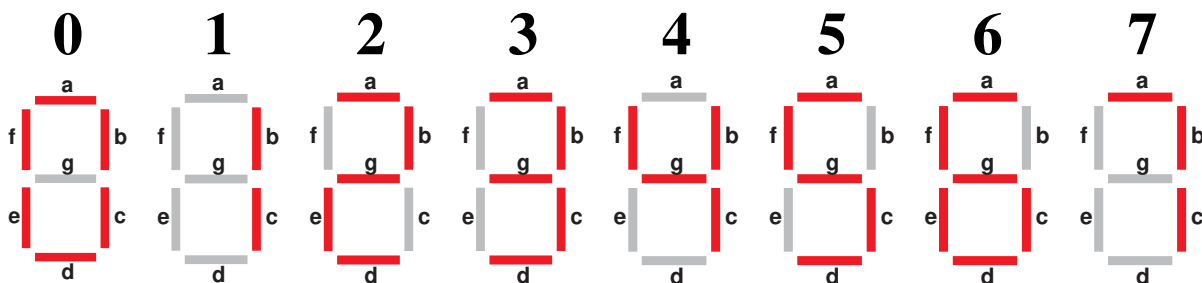


figura 1

Decodificatorul trebuie să funcționeze conform următoarelor tabele de adevăr:

Table 1

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

Se realizează montajul din figura următoare:

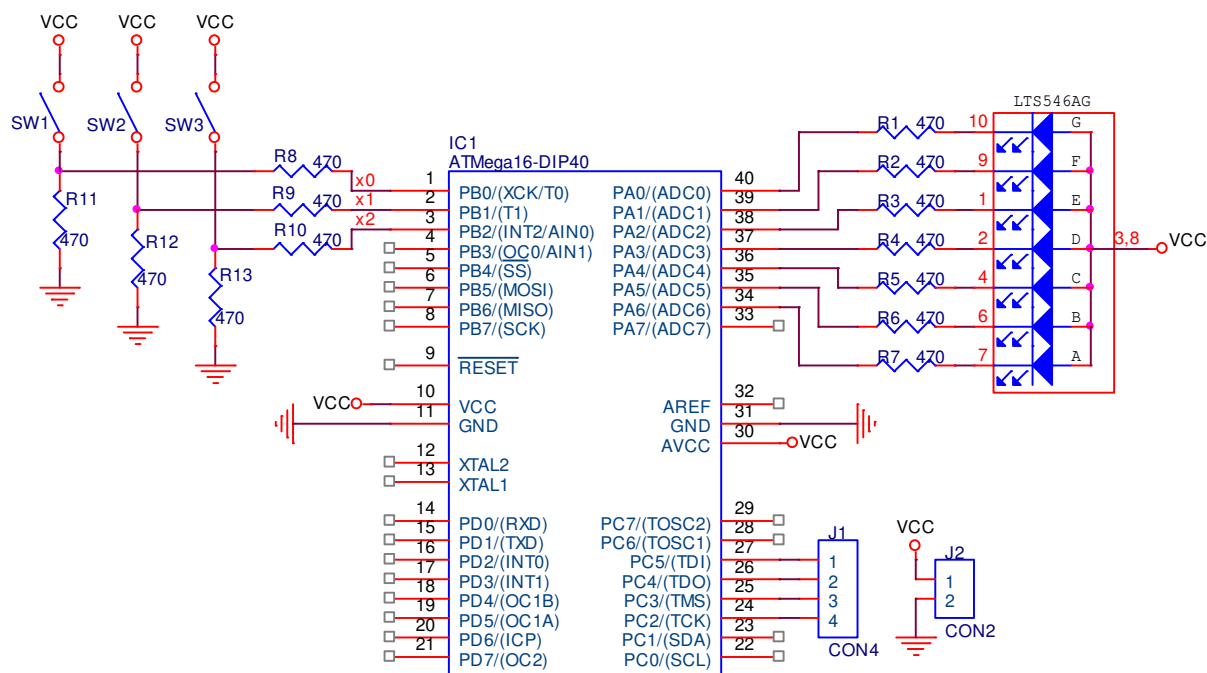


figura 2

Afișorul folosit este cu anod comun și are codul LTS546AG. Acest afișor a fost folosit în primul laborator (acolo găsiți datele de catalog). Tot în primul laborator se cerea proiectarea unei scheme care să permită afișarea oricărei combinații de segmente cu aceeași intensitate. Soluția este înserierea unei rezistențe pe fiecare segment, așa cum se observă în figura 2.

Intrările  $x_0$ ,  $x_1$ ,  $x_2$  se vor citi prin intermediul portului B. **Portul B** este 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 0-6. **Portul A** este configurat ca port de **ieșire**.

Amplasarea fizică și conectarea segmentului g este prezentată în figura următoare:

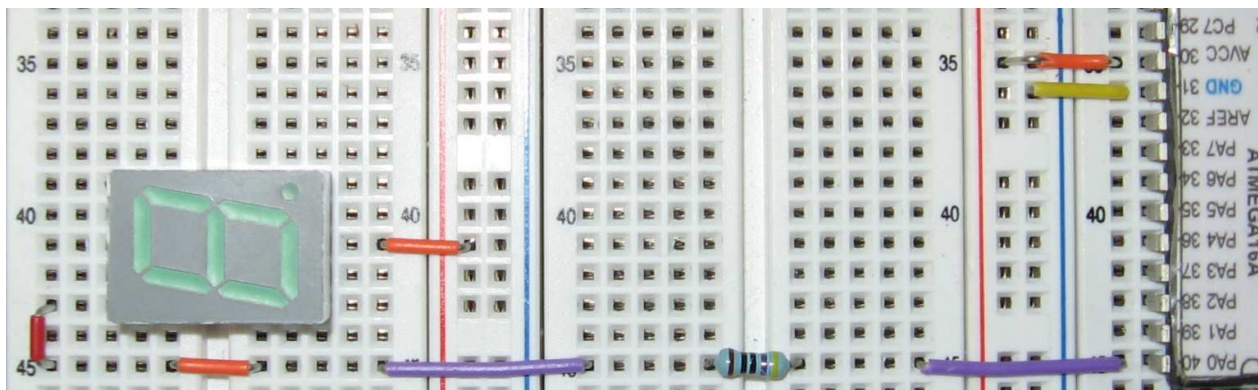


figura 3

**Implementați și restul schemei** din figura 2. Nu uitați, realizarea montajului se face fără să încălecați fire.

**NU ALIMENTAȚI PANA NU A FOST VALIDATA CORECTITUDINEA MONTAJULUI ! IN CAZ CONTRAR VETI SUPORTA EVENTALELE PAGUBE !**

## 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:

```
AVR Memory Usage
-----
Device: atmega16

Program:      124 bytes (0.8% Full)
(.text + .data + .bootloader)

Data:         0 bytes (0.0% Full)
(.data + .bss + .noinit)
```

Test va fi folosit în continuare ca termen de comparație.

Construiți Table 2 pe hârtie sau creați un fișier text. Resursele utilizate pentru codul anterior a fost înscris în Table 2, linia **Test**.

Table 2

Proiect \ Resurse	Program	Data	$\Delta(\text{Proiect} - \text{Test})$	
			$\Delta \text{ Program}$	$\Delta \text{ Data}$
Test	124	0	$124 - 124 = 0$	$0 - 0 = 0$
LUT_RAM	p_ram	d_ram	$p\_ram - 124$	$d\_ram - 0$
LUT_ROM	p_rom	d_rom	$p\_rom - 124$	$d\_rom - 0$
SWITCH	p_sw	d_sw	$p\_sw - 124$	$d\_sw - 0$

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 structură 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. Mascăți la ,0' biți nefolosiți.

**Când sunteți gata, chemați profesorul pentru verificarea montajului.**

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

Notați cu **p\_ram** lungimea în octeți a secțiunii Program și cu **d\_ram** lungimea în octeți a secțiunii Data. Completați linia LUT\_RAM din Table 2,

Dacă funcționează conform tablei de adevăr din Table 1 **chemați profesorul pentru validare!**

3. Varianta de la punctul 2 folosește 8 octeți din memoria 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 tablei 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 această variantă este foarte asemănător cu cel de la punctul 2, 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 tablei de căutare prin adăugarea specificatorului **PROGMEM**:

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

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 cu **Help → avr-libc Reference → Library Reference → [<avr/pgmspace.h>: Program Space Utilities](#)**.

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. Cu aceste valori completați linia LUT\_ROM în Table 2.

Apoi se verifică funcționarea: se fac toate combinațiile posibile ale intrărilor și se observă afișorul. Chiar dacă funcționează conform tablei de adevăr din Table 1, **Deocamdată NU chemați profesorul** pentru validare!

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

Faceți **Build**. Notați cu **p\_sw** lungimea în octeți a secțiunii Program și cu **d\_sw** lungimea în octeți a secțiunii Data. Cu aceste valori completați linia SWITCH în Table 2.

Comparați cele 3 variante de implementare! Dacă varianta cea mai scurtă ia nota 10, **ce notă ar trebui să ia o implementare cu switch sau if?**

**Nu** mai verificați funcționarea.

Dacă implementarea LUT\_ROM de la punctul 3 este funcțională și Table 2 este completat, reîncărcați proiectul LUT\_ROM și **chemați profesorul pentru validare.**

**Dacă ați ajuns aici aveți nota 5.**

Secțiunea opțională (se va executa în ordinea crescătoare a pașilor):

### **Pasul 3: Decodificator zecimal**

**Creați un nou proiect.** Se cere implementarea unui decodificator 7 segmente ale cărui intrări reprezintă o cifră zecimală codificată BCD8421. Pentru implementare se va folosi metoda tabeli de căutare (LUT) rezidentă în memoria de cod.

Decodificatorul zecimal furnizează la ieșire funcțiile a,b,c,d,e,f,g în așa fel încât să se aprindă cifrele 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Dacă la intrare se formează o combinație mai mare ca nouă, decodificatorul va afișa ,–' (numai segmentul g aprins).

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

**Dacă ați ajuns aici aveți între 1 și 2 puncte în plus, în funcție de calitatea implementării.**

### **Pasul 4: Decodificator zecimal cu verificarea afișării**

La proiectul de la pasul 3 adăugați o nouă intrare, numită TEST. Dacă această intrare este ,0', montajul va funcționa ca la pasul 3. Dacă această nouă intrare este ,1', montajul va afișa în buclă cifrele 0-9. Afișarea unei cifre va dura aproximativ 1 secundă. Folosiți ideea din laboratorul 2 – blink.

La comutarea din regimul normal în regimul test, numărarea va porni de la zero.

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

**Dacă ați ajuns aici aveți între 1 și 3 puncte în plus, în funcție de calitatea implementării.**

### **Pasul 5: Deconectare**

La terminare executați următoarele operații:

1. Se oprește execuția programului.
2. Se oprește depanatorul.
3. **Se șterge memoria de program.**
4. Se închide AVR Studio.
5. Se oprește alimentarea JTAG ICE.
6. Se oprește sursa Hameg
7. Se desface montajul, numai componentele adăugate în acest laborator.