LABORATOR 4- Implementarea SLC-urilor, varianta LUT

Implementarea tip ROM a funcţiilor booleene

În laboratorul precedent s-au implementat 3 funcţii 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 funcţiei f_1. Din tabelă se observa că funcţia 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 instrucţiuni în cod maşină şi ocupă 6 octeţi în memoria de cod. Fără nici un dubiu, este cea mai scurtă implementare cu putinţă pentru f_1 prin metoda analitică.

În laboratorul precedent pentru emularea software a funcţiilor logice s-a folosit şi minimizarea. Minimizarea a fost studiată în cadrul cursului BLPC (DSD). Această metodă nu este însă singura; o altă metodă de implementare a funcţiilor logice se bazează pe memorii ROM. Această metodă nu necesită minimizare şi 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 ieşirile $D_{2:0}$ se vor obţine cele 3 funcţii.

Această idee poate fi folosită în cazul emulării circuitelor logice combinaţionale: 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 funcţii 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
      // şi codul generat este la fel de lung
   }
}
```

Se observă că valorile funcţiilor $f_{2:0}$ au fost memorate folosind un octet pentru fiecare valoare a intrării, dar din fiecare octet s-au folosit doar 3 biţi. Fără a creşte spaţiul de memorare, folosind mai mulţi biţi din cei 8 disponibili, se pot implementa până la 8 funcţii de 3 variabile. Este evident că această variantă este mult mai avantajoasă: pentru 8 funcţii de 3 variabile sunt necesari 8 octeţi pentru tabelă şi 8 octeţi pentru operaţia de indexare. Cei 16 octeţi sunt alocaţi în spaţiul de cod. Metoda analitică, la un consum de 6 octeţi per funcţie, ar necesita 6*8=48 de octeţi. Un alt avantaj îl constituie faptul că funcţiile nu mai trebuie minimizate; a minimiza 8 funcţii necesită un timp neneglijabil, dacă această operaţie se face manual.

Metodă tabelei de căutare (LUT) funcţionează pentru un număr relativ mic de variabile de intrare. Dacă metoda s-ar aplica pentru funcţia AND de 10 variabile dimensiunea tabelei de căutare ar fi de 1024 de octeţi. În schimb metoda analitică necesită numai 18 octeţi.

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

Să presupunem că un anumit proiect necesită calculul funcţiei sin(x), unde x este o intrarea reprezentată pe 8 biţi. Funcţia sin(x) se calculează prin dezvoltare în serie Taylor şi 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 operaţii. Biblioteca de virgulă mobilă are o dimensiune considerabilă! În acest caz este mult mai simplu să se precalculeze sin(x) şi 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 câţiva kiloocteţi pentru biblioteca de virgulă mobilă.

Alegerea metodei de implementare a funcţiilor booleene depinde de numărul de variabile de intrare, de numărul de funcţii de implementat şi de complexitatea acestora. NU există reţete prestabilite şi numai proiectantul (adică dumneavoastră) poate să facă alegerea.

Atenţie: chiar dacă prezentarea din acest laborator şi din cel precedent accentuează importanţa obţinerii celui mai scurt cod, în practică mai există un factor de care trebuie ținut seama: respectarea timpului alocat dezvoltării – **time to market**. Degeaba am obţinut 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 diferenţă din punct de vedere al prețului între o implementare de 10KB şi una de 11KB. Va conta însa o diferenţă între 16KB şi 17KB!

Scopul lucrării

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

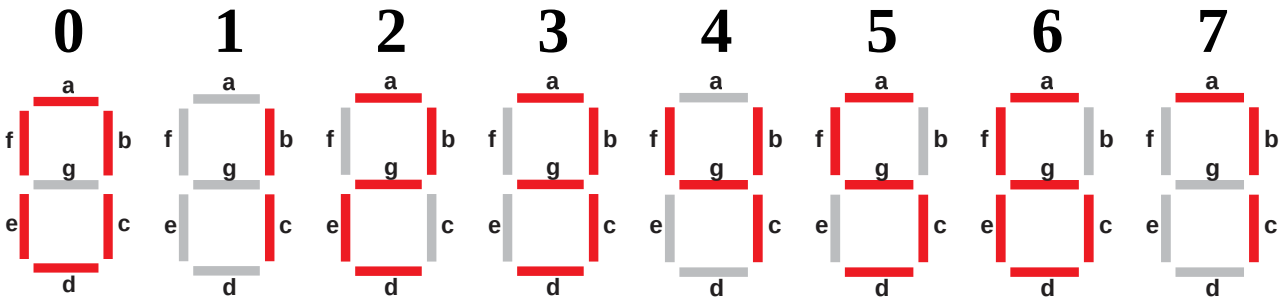

figura 1

Decodificatorul trebuie să funcţioneze conform următoarei 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**								

Completaţi tabela de mai sus.

Desfăşurarea lucrării

Pasul 1: Realizarea montajului

Se realizează montajul din figura următoare:

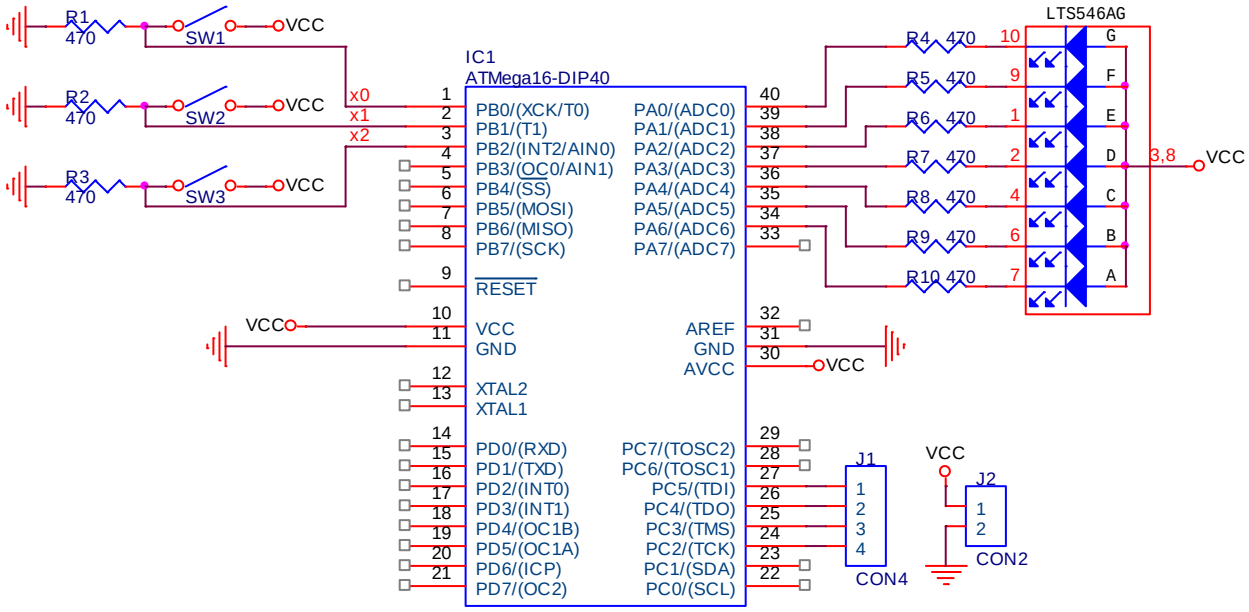

figura 2

Afişorul folosit este cu anod comun şi are codul LTS546AG. Acest afişor a fost folosit în primul laborator (acolo găsiți datele de catalog). Tot în primul laborator se cerea proiectarea unei scheme care să permită afişarea oricărei combinaţii de segmente cu aceeaşi intensitate. Soluţia este înserierea unei rezistenţe pe fiecare segment, aşa 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 funcţiilor de ieşire a..g din Table 1 se vor înscrie în PORTA, biţii 0-6. **Portul A** este configurat ca port de **ieşire**.

Amplasarea fizică şi conectarea segmentului *g* este prezentată în figura următoare:

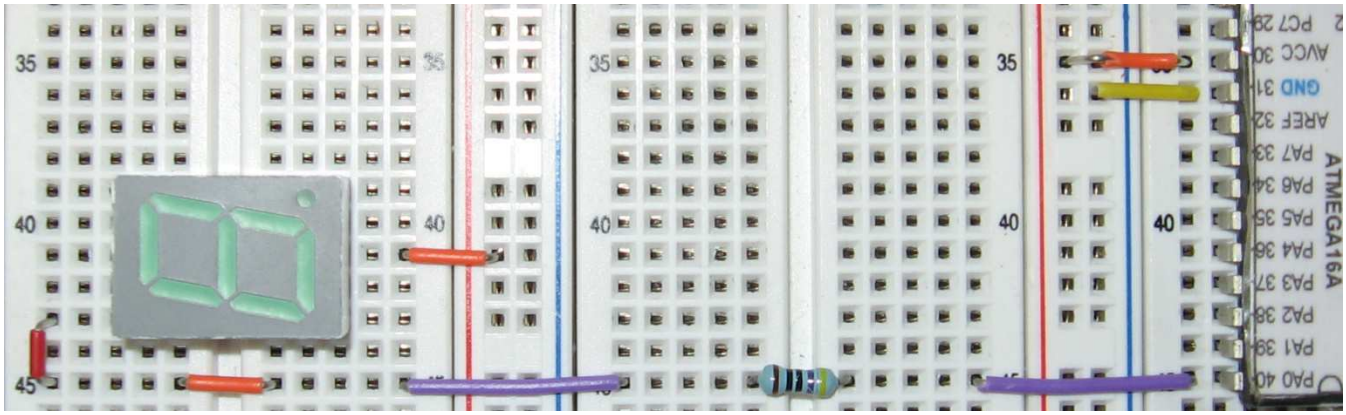

figura 3

Implementaţi şi restul schemei din figura 2. Nu uitaţi, realizarea montajului se face fără să încălecați fire.

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

Pasul 2: Crearea codului sursă şi execuţia

Pentru a evidenția cum depinde lungimea codului maşină şi 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 funcţional**:

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

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

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

Nu este nevoie să creaţi sau să executaţi 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 comparaţie.

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

Table 2

Resurse / Proiect	Program	Data	Δ(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 - 0
LUT_ROM	p_rom	d_rom	p_rom - 124	d_rom - 0
SWITCH	p_sw	d_sw	p_sw -124	d_sw - 0

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

2. **Creaţi un nou proiect cu numele LUT_RAM** conform paşilor prezentaţi în laboratoarele anterioare.

Implementați decodificatorul octal conform indicaţiilor din secţiunea **Implementarea tip ROM a funcţiilor booleene**. Codul va avea aceeaşi structura ca exemplul de la pagina 2. Tabela de căutare se va numi segLUT în loc de `fnLUT`.

Nu uitaţi! Citirea portului de intrare generează o valoare pe 8 biţi, dar numai 3 biţi au o valoare bine determinată. Mascaţi la ‚0’ biţi nefolosiți.

Când sunteți gata, chemaţi profesorul pentru verificarea montajului.

Verificaţi dacă funcționează! Se fac toate combinaţiile posibile ale intrărilor şi se observă afişorul.

Notaţi cu **p_ram** lungimea în octeți a secțiunii Program şi cu **d_ram** lungimea în octeți a secțiunii Data. Completaţi linia LUT_RAM din Table 2,

Dacă funcţionează conform tabelei de adevăr din Table 1 **chemaţi profesorul pentru validare!**

3. Varianta de la punctul 2 foloseşte 8 octeţi din memoria RAM pentru a memora tabela de căutare. Această soluţie este rar admisă deoarece risipeşte RAM valoros: avem 16KB de ROM şi numai 1KB de RAM. Mai mult, tabela se află iniţial în ROM şi este transferată în RAM în faza de iniţializare. Faza de iniţializare este adăugată de compilator. Nu uitaţi, nu avem sistem de operare care să facă încărcarea programului.

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

   Creaţi un nou proiect cu numele LUT_ROM. Codul această variantă este foarte asemănător cu cel de la punctul 2, cu trei diferenţe. Pentru ca tabela de căutare să fie creată în ROM vom include şi headerul **pgmspace** cu:

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

   Apoi vom modifica declaraţia tabelei 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 sunteţi curioşi, informaţii suplimentare despre funcţiile din biblioteca pgmspace se află cu **Help → avr-libc Reference → Library Reference → <avr/pgmspace.h>: Program Space Utilities**.

   Faceți **Build**. Notaţi cu **p_rom** lungimea în octeți a secțiunii Program şi cu **d_rom** lungimea în octeți a secțiunii Data. Ar trebui ca lungimea zonei **Data** să fie 0 octeţi. Cu aceste valori completaţi linia LUT_ROM în Table 2.

   Apoi se verifică funcţionarea: se fac toate combinaţiile posibile ale intrărilor şi se observă afişorul. Chiar dacă funcţionează conform tabelei de adevăr din Table 1, **Deocamdată NU chemaţi profesorul** pentru validare!

4. Există tendinţă printre studenți ca în loc de LUT sa să emuleze decodificatorul cu `if` sau `switch`. **Creaţi proiectului cu numele PSWITCH**. Implementaţi 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. Cu aceste valori completaţi linia SWITCH în Table 2.

   Comparaţi 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 verificaţi funcţionarea.

   Dacă implementarea LUT_ROM de la punctul 3 este funcţională şi Table 2 este completat, reîncărcați proiectul LUT_ROM şi **chemaţi profesorul pentru validare.**

   Dacă aţi ajuns aici aveți nota 5.

Secţiunea opţională (se va executa în ordinea crescătoare a paşilor):

Pasul 3: Decodificator zecimal

Creaţi 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 tabelei de căutare (LUT) rezidentă în memoria de cod.

Decodificatorul zecimal furnizează la ieşire funcţiile a,b,c,d,e,f,g în aşa 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 funcţionează, chemaţi profesorul pentru validare.

Dacă aţi ajuns aici aveți între 1 şi 2 puncte în plus, în funcție de calitatea implementării.

Pasul 4: Decodificator zecimal cu verificarea afişării

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. Afişarea unei cifre va dura aproximativ 1 secundă. Folosiţi ideea din laboratorul 2 – blink.

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

Când funcţionează, chemaţi profesorul pentru validare.

Dacă aţi ajuns aici aveți între 1 şi 3 puncte în plus, în funcție de calitatea implementării.

Pasul 5: Deconectare

La terminare executați următoarele operaţii:

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