Akademia
Górniczo-Hutnicza
w Krakowie
Katedra Elektroniki

Technika mikroprocesorowa

Ćwiczenie 4

Operacje logiczne

Autor: Paweł Russek
Tłumaczenie: Mariusz Sokołowski
http://www.fpga.agh.edu.pl/upt
wersja: 26.04.2016r.

1. Cel

Głównymi celami niniejszego ćwiczenia jest:

- zapoznanie się z trybem adresacji pośredniej,
- poznanie wybranych instrukcji, takich jak:
  - wykonujących operacje na bitach,
  - wykonujących podstawowe operacje logiczne,
  - wykonujących operacje przesuwania,
- poznanie algorytmu sumowania liczb bez znaku.

W/w tematy będą przedstawione na przykładzie programowania wyświetlacza LED modułu ZL3AVR.

Do testowania wszystkich programów będzie wykorzystywany sprzętowy system uruchomieniowy.

2. Wymagania wstępne

- Zaliczone ćwiczenia nr 1, 2 i 3.

3. Wbudowany symulator/debbuger

W przypadku większości aplikacji, do przeprowadzenia efektywnego procesu uruchamiania testowanego programu niezbędny jest systemowy symulator/debbuger .Programowy symulator może nie być na tyle wydajny, aby poradzić sobie z bardziej rozbudowanymi projektami. Poza tym, symulator takiego rodzaju, nie odzwierciedla prawdziwych reakcji sprzętu na działanie programu. Wobec powyższego, w toku niniejszych ćwiczeń, będzie wykorzystywany sprzętowy system uruchomieniowy (ang. debbuger), celem łatwiejszego przeprowadzenia procesu testowania, powstałych w trakcje zajęć, programów. Zaletą takiego systemu jest to, że instrukcje uruchamianego programu są wykonywane przez fizyczny mikrokontroler, pod nadzorem systemu nadrzędnego, pracującego np. na komputerze klasy PC, dając w rezultacie wrażenie wykonywania wszystkiego na komputerze. Sprzętowy debbuger pozwala na: pracę krokową (instrukcja po instrukcji), zakładanie pułapek, obserwowanie stanu zmiennych, rejestrów, portów i pamięci, itd. Na przykład, gdy instrukcja w programie odczytuje stan portu A, to na ekranie komputera wyświetla się faktyczny stan tego portu. Jeżeli instrukcja zapisuje wartość do portu A, to stan ten jest od razu widoczny na fizycznych końcówkach mikrokontrolera.

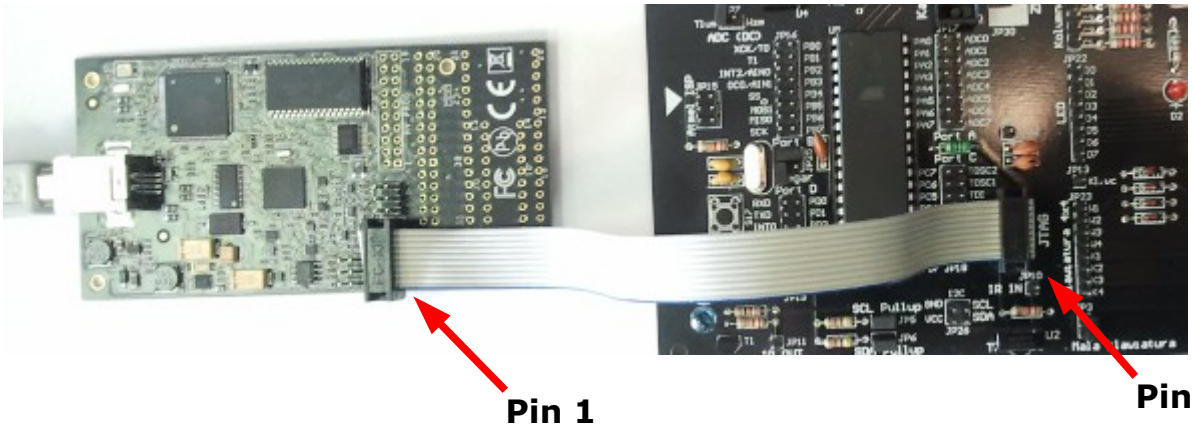
Dzisiejsze mikrokontrolery oferują złącze JTAG do implementacji sprzętowego debbuger'a. Z reguły potrzebny jest specjalny układ do realizacji debugger'a w oparciu o złącze JTAG. My skorzystamy z modułu AVR Dragon, który wraz z oprogramowaniem Atmel Studio, tworzy kompletny system uruchomieniowy, pozwalający równocześnie programować mikrokontrolery AVR.

3.1. Podłączanie modułu AVR Dragon

Moduł AVR Dragon ma możliwość współpracowania z portem JTAG lub ISP mikrokontrolera AVR. W naszym przypadku wykorzystamy złącze JTAG, celem zrealizowania sprzętowego

debbuger'a. Aby tego dokonać, należy wykonać następujące czynności:

1. Podłączyć port JTAG modułu AVR Dragon do złącza JTAG płytki ZL3AVR (JP21). Zwróć szczególną uwagę na pozycję pinu nr 1 (rys. 1).

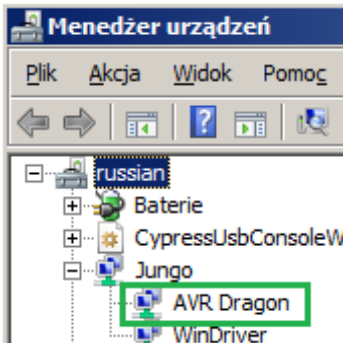


Rys. 1 . Podłączanie modułu AVR Dragon do płytki ZL3AVR.

UWAGA!

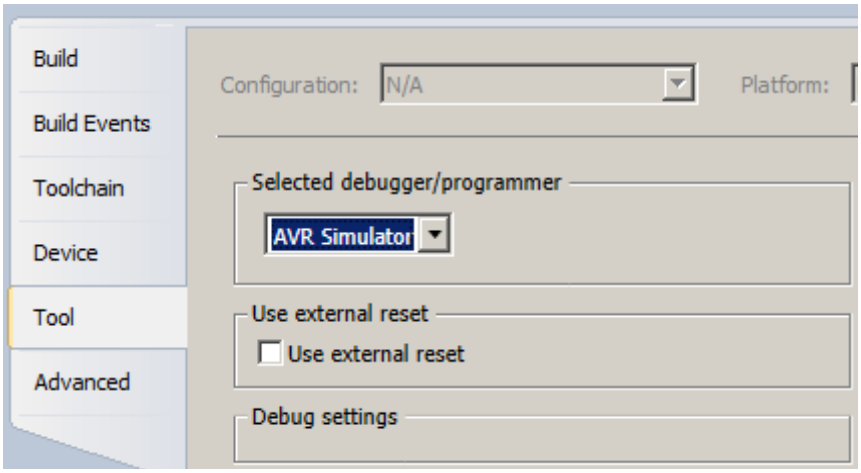
Należy używać kabla o długości około 20cm. Dłuższy kabel może zaburzyć poprawną pracę układów, ze względu na szkodliwy wpływ na zależności czasowe sygnałów.

2. Podłączyć moduł AVR Dragon do portu USB komputera. Jeżeli czynność ta jest przeprowadzana pierwszy raz, zgłosi się okno instalatora sprzętu systemu Windows. Aby zainstalować odpowiednie sterowniki, należy posiadać uprawnienia administratora w systemie komputera. Wszelkie problemy należy zgłaszać prowadzącemu zajęcia.
3. W Menedżerze Urządzeń należy sprawdzić poprawność instalacji modułu (rys. 2).



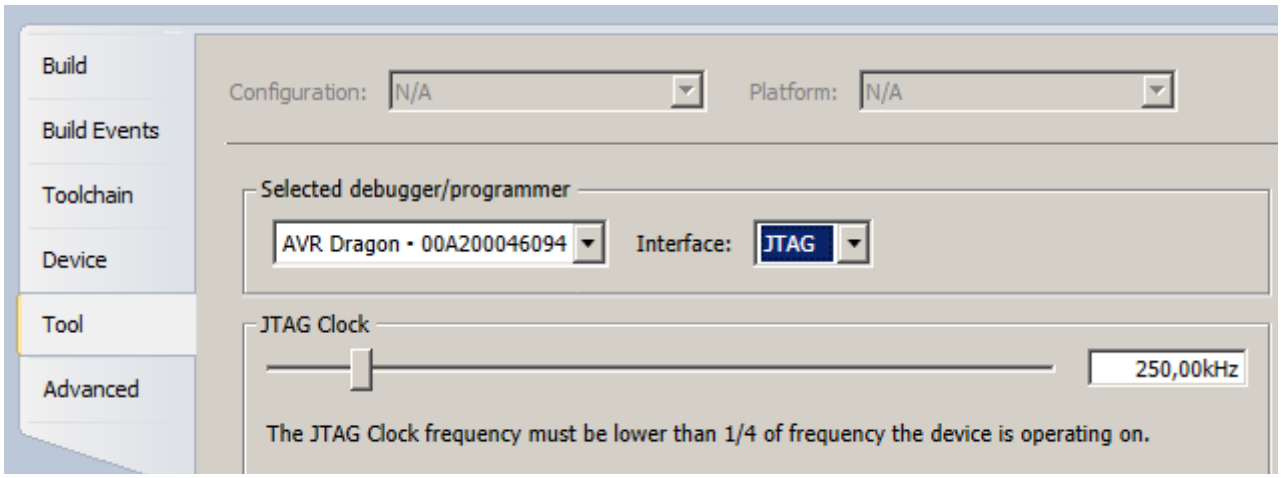
Rys. 2

4. Uruchomić program „AVR Simulator”.



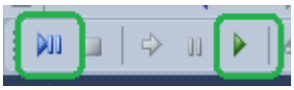
Rys. 3

5. W wyświetlonym oknie narzędziowym wybrać **AVR Dragon** w polu 'Selected debugger/programmer' i interfejs **JTAG** interface w polu 'Interface' – rys. 3 i 4.



Rys. 4

6. Podłączyć zasilanie do modułu ZL3AVR. Wybrać jedną z komend debugger'a (rys. 5).



Rys. 5

Interfejs debugger'a pozwala, podobnie jak programowy symulator, wybierać funkcje takie, jak: uruchom (ang. 'run'), wstrzymaj (ang. 'pause'), zatrzymaj (ang. 'stop), praca krokowa (ang. 'step'), załóż pułapki (ang. 'breakpoints'). Rożnica jest jednak taka, że efekty pracy symulatora widać tylko na ekranie komputera, a efekty pracy debuggr'a fizycznie na peryferiach modułu ZL3AVR.

4. Nowa dyrektywa assembler'a

Dyrektywa definicji stałych bajtowych – DB

Dyrektywa ta rezerwuje lokacje w pamięci programu lub pamięci EEPROM. Żeby mieć możliwość odwołania się do danej lokacji, dyrektywa DB powinna być poprzedzona etykietą, pozwalającą ustalić adres początku zarezerwowanej przestrzeni. Po dyrektywie DB zamieszcza się listę danych 8-bitowych, rozdzielonych przecinkami. Lista musi zawierać przynajmniej jedną daną z zakresu 0÷255 lub -128÷127 w przypadku liczb ze znakiem (w pamięci zostaje umieszczona liczba w kodzie U2). Jeżeli lista zawiera więcej niż jedną liczbę, każde dwie dane 8-bitowe zostaną upakowane w słowo 16-bitowe w pamięci programu. W przypadku nieparzystej ilości, ostania dana zostanie dopełniona zerem, aby stworzyć 16-bitowe słowo nawet, jeśli następną instrukcją jest dyrektywa DB.

Składnia:

```
.xSEG          ;.CSEG – pamięć programu, .ESEG – pamięć EEPROM
etykieta: .DB lista_danych_8-bitowych
```

Przykład:

```
.include"m32def.inc"
.org 0x00                              ;dyrektywa org nie jest konieczna
            rjmp prog_start            ;skok do programu głównego
.org 0x32                              ;adres początku listy danych dyrektywy DB
prime: .DB 2, 3, 5, 7, 11, 13, 17, 19, 23   ;stworzenie listy dziewięciu liczb pierwszych
                                       ;w przestrzeni pamięci programu
.org 0x100                             ;adres początku programu
prog_start:  ldi r30, low(2*prime)     ;mnożenie przez dwa, celem uzyskania adresu
             ldi r31, high(2*prime)    ;w przestrzeni bajtowej
             ...
```

Zawartość pamięci programu od adresu 0x32 pokazuje tabela tab. 1.

Adresacja 16-bitowa		**Adresacja 8-bitowa**	**Pamięć programu**	**Komentarz**
0x00		0x00	**0xff**	*rjmp prog_start*
		0x01	**0xc0**	
0x01		0x02		*niezdefiniowane*
		0x03		
...				
0x32		0x64	**0x02**	*Zawartość listy dyrektywy DB*
		0x65	**0x03**	
0x33		0x66	**0x05**	
		0x67	**0x07**	

0x34		0x68	**0x0b**	
		0x69	**0x0d**	
0x35		0x6a	**0x11**	
		0x6b	**0x13**	
0x36		0x6c	**0x17**	
		0x6d	**0x00**	*Dana uzupełniona wartością 00*
…		…	…	
0x100		0x200	????	*prog_start:* *ldi r30, low(2*prime)* *ldi r31, high(2*prime)*
		0x201	????	
0x101		0x202	????	
		0x203	????	
...				

Tab. 1

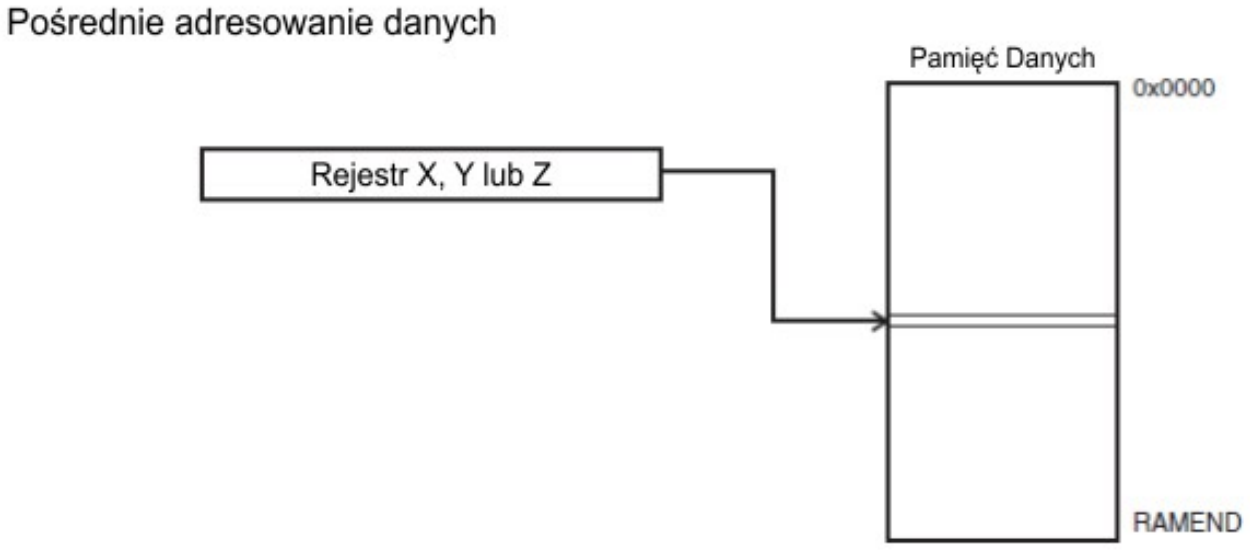
Zauważ, że dyrektywa .org używa adresacji 16-bitowej (dwa kolejne bajty). Natomiast etykieta *prime* jest zastąpiona wartością odpowiednią od adresacji 8-bitowej (bajtowej) – mnożenie przez 2. Dlatego wartości z listy DB są zamieszczone od adresu 0x64, podczas gdy dyrektywa .org wskazuje adres 0x32.

Ćwiczenie 4.1

Zweryfikuj zawartość tabeli tab. 1 używając symulatora. Wypełnij pola, zawierające ????, poprawnymi wartościami.

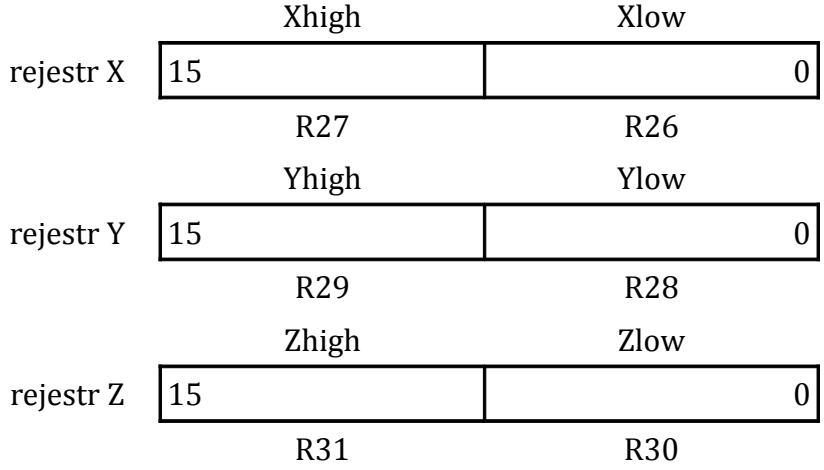
5. Adresacja pośrednia

W trybie pośredniej adresacji, do generacji adresu danej lokacji w pamięci, jest wykorzystywana zawartość rejestru. W rodzinie AVR, do w/w funkcji, są przeznaczone trzy 16-bitowe rejestry X, Y i Z. Pozwalają one na zaadresowanie całej przestrzeni adresowej procesora, czyli 65 536-iu bajtów (rys. 6).



Rys. 6

Każdy z rejestrów jest tworzony z zestawu dwóch, specyficznych rejestrów ogólnego przeznaczenia. Na przykład rejestr X składa się z pary rejestrów R26 i R27.



Rejestry X, Y i Z mogą być używane jako wskaźniki, wykorzystywane np. przez instrukcje ***ld*** i ***lpm***.

5.1. Instrukcja ld

Instrukcja ***ld*** ładuje jeden bajt z pamięci danych do rejestru Rn. Adres bajtu jest zawarty w rejestrze wskaźnikowym X, Y lub Z (16-bitowym), którego zasięg dotyczy bieżącego segmentu pamięci danych o rozmiarze 64KB.

Po wykonaniu transferu, rejestr wskaźnikowy może zostać w stanie niezmienionym lub jego wartość może być automatycznie zwiększona o jeden. Trzecią możliwością jest automatyczne zmniejszenie zawartości rejestru wskaźnikowego o jeden, przed wykonaniem operacji transferu. Właściwości te można wykorzystać m.in. do operacji na łańcuchach i tabelach.

ld Rn , X	*;0 ≤ n ≤ 31*	*ld Rn , Y*	*;0 ≤ n ≤ 31*	*ld Rn , Z*	*;0 ≤ n ≤ 31*
ld Rn, X+	*;0 ≤ n ≤ 31*	*ld Rn, Y+*	*;0 ≤ n ≤ 31*	*ld Rn, Z+*	*;0 ≤ n ≤ 31*
ld Rn, -X	*;0 ≤ n ≤ 31*	*ld Rn, -Y*	*;0 ≤ n ≤ 31*	*ld Rn, -Z*	*;0 ≤ n ≤ 31*

5.2. Instrukcja lpm

Instrukcja ***lpm*** (Load Program Memory) ładuje jeden bajt z pamięci programu do rejestru Rn. Adres bajtu jest zawarty w rejestrze wskaźnikowym Z. Instrukcja ta pozwala na efektywne wykorzystanie miejsca w pamięci, przy operowaniu na stałych bajtowych, mimo że pamięć programu ma organizację 16-bitową. Zapewnia to fakt, że rejestr Z zawiera adres bajtu a nie słowa 16-bitowego.

Po wykonaniu transferu, rejestr Z może zostać w stanie niezmienionym lub jego wartość może być automatycznie zwiększona o jeden.

lpm	*;domyślnie R0*	
lpm Rn, Z	*;0 ≤ n ≤ 31*	
lpm Rn, Z+	*;0 ≤ n ≤ 31*	*Z ← Z + 1*

Przykład:

```
.include"m32def.inc"
.org 0x00                          ;dyrektywa org nie jest konieczna
                rjmp prog_start    ;skok do programu głównego
.org 0x32                          ;adres początku listy danych dyrektywy DB
prime: .DB 2, 3, 5, 7, 11, 13, 17, 19, 23, 29   ;stworzenie listy dziesięciu liczb pierwszych
                                                ;w przestrzeni pamięci programu
.org 0x100                         ;adres początku programu
prog_start:     ldi zl, low(2*prime)    ;mnożenie przez dwa, celem uzyskania adresu
                ldi zh, high(2*prime)   ;w przestrzeni bajtowej
                lpm r16, z
                out porta, r16
```

UWAGA!

Instrukcja ***ld*** operuje w przestrzeni adresowej **pamięci danych**, natomiast instrukcja ***lpm*** w przestrzeni adresowej **pamięci programu**.

Instrukcją o odwrotnym działaniu w stosunku do ***ld*** jest instrukcja ***st*** (Store), która zapisuje daną z rejestru Rn do pamięci danych, adresowanej za pomocą rejestrów X, Y lub Z.

Pamięć programu może być zapisywana za pomocą instrukcji ***spm*** (Store Program Memory). Należy jedna pamiętać, że jest to pamięć typu Flash, która wymaga odpowiednich procedur kasowania stronicowego.

Alternatywą dla instrukcji ***ld*** jest instrukcja ***lds***, wykorzystująca tryb adresacji bezpośredniej, gdzie adres jest zawarty bezpośrednio za kodem instrukcji.

Przykład:

```
lds r2,$FF00      ;ładuje rejestr R2 zawartością komórki pamięci danych o adresie $FF00
```

Instrukcji tej używa się w przypadku, gdy adres jest typu statycznego, nie modyfikowany, jak np. w pętlach.

Ćwiczenie 4.2

Napisz program, który w sposób ciągły będzie wyświetlał dziesięć kolejnych liczb pierwszych na diodach LED modułu ZL3AVR. Zastosuj podprogram Opóźnienie o długości ok. 1s, pozwalający na obserwację zmieniających się liczb. Zapisz program pod nazwą 'PrimesLEDs.asm'.

6. Podstawowe operacje logiczne

Instrukcje operacji logicznych są najbardziej powszechnie używanymi instrukcjami. Realizują podstawowe działania Boole'a takie, jak: I (AND), LUB (OR), ALBO (XOR) i NIE (NOT) – negacja.

6.1. Instrukcje and i andi – funkcja logiczna AND

Instrukcja ***and*** wykonuje operację funkcji logicznej AND pomiędzy rejestrami Rd i Rr (każdy bit osobno) - wynik zachowuje w rejestrze Rd.

and Rd, Rr ;0 ≤ d ≤ 31, 0 ≤ r ≤ 31

Instrukcja ***andi*** wykonuje operację funkcji logicznej AND pomiędzy rejestrem Rd i daną natychmiastową, zawartą w kodzie instrukcji - wynik zachowuje w rejestrze Rd.

andi Rd, K 16 ≤ d ≤ 31, 0 ≤ K ≤ 255

Funkcja logiczna AND		
Wejście		Wyjście
X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

Powyższe instrukcje mają wpływ na wartość flagi Z w rejestrze stanu (Z=1, jeśli wynik jest równy 0). Są najczęściej wykorzystywane do ustawiania wartości 0 na wybranych bitach operandu.

Przykład:

```
andi r17,$0F  ; ustawia bity starszej połowy rejestru R17 na 0
andi r18,$10  ; wyizolowanie bitu nr 4 w rejestrze R18
andi r19,$AA  ; zeruje wszystkie parzyste bity rejestru R19
and r2,r3     ; funkcja AND pomiędzy rejestrami R2 i R3 – wynik w R2
ldi r16,1     ; ustawienie maski 0000 0001 w rejestrze R16
and r2,r16    ; wyizolowanie bitu nr 0 w rejestrze R2
```

6.2. Instrukcje or i ori – funkcja logiczna OR

Instrukcja ***or*** wykonuje operację funkcji logicznej OR pomiędzy rejestrami Rd i Rr (każdy bit osobno) - wynik zachowuje w rejestrze Rd.

or Rd,Rr ;0 ≤ d ≤ 31, 0 ≤ r ≤ 31

Instrukcja ***ori*** wykonuje operację funkcji logicznej OR pomiędzy rejestrem Rd i daną natychmiastową, zawartą w kodzie instrukcji - wynik zachowuje w rejestrze Rd.

ori Rd,K ;16 ≤ d ≤ 31, 0 ≤ K ≤ 255

Funkcja logiczna OR		
Wejście		Wyjście
X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

Powyższe instrukcje mają wpływ na wartość flagi Z w rejestrze stanu (Z=1, jeśli wynik jest równy 0). Są najczęściej wykorzystywane do ustawiania wartości 1 na wybranych bitach operandu.

Przykład:

```
ldi r20, 0x40  ; r20 = 0x04
ori r20,0x30   ;teraz r20 = 0x34
```

6.3. Instrukcja com – funkcja logiczna NOT

Instrukcja ***com*** wykonuje operację funkcji logicznej NOT na zawartości rejestru Rd (negacja każdego bitu osobno).

com Rd ;0 ≤ d ≤ 31

Funkcja logiczna NOT	
Wejście	Wyjście
X	NOT X
0	1
1	0

Powyższa instrukcja ma wpływ na wartość flagi Z w rejestrze stanu (Z=1, jeśli wynik jest równy 0).

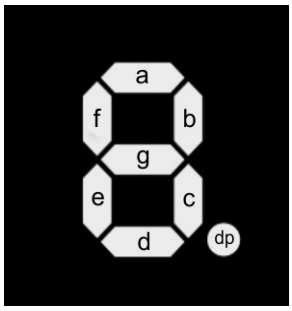
Przykład:

```
com r4      ;neguje zawartość rejestru R4
breq zero   ;skok, jeśli wynik równy 0
```

7. Dodatkowe informacje o module ZL2AVR

7.1. Wyświetlacz siedmiosegmentowy LED

Na rysunku rys. 7 przedstawiono topologię klasycznego wyświetlacza siedmiosegmentowego. Odpowiednia kombinacja wyboru segmentów pozwala na wyświetlanie cyfr, niektórych liter (potrzebnych np. do reprezentacji liczby szesnastkowej) oraz znaków definiowanych przez użytkownika. Każdy segment jest oznaczony literą, od a do g, a ósmy, opcjonalny, segment pozwala na wyświetlanie liczb z ułamkiem dziesiętnym (kropka) lub na przykład czasu (dwukropek).



Rys. 7

Tabela tab. 2 zawiera listę stanów poszczególnych segmentów, celem wizualizacji cyfr kodu szesnastkowego (cyfry od 0x0 do 0xF). Dane z tabeli pozwalają na transkodowanie reprezentacji szesnastkowej znaku na kod wyświetlacza.

	p7	**p6**	**p5**	**p4**	**p3**	**p2**	**p1**	**p0**	
Cyfra	**dp**	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**HEX**
0	Off	On	On	On	On	On	On	Off	0x7E
1	Off	Off	On	On	Off	Off	Off	Off	0x30
2	Off	On	On	Off	On	On	Off	On	0x6D
3	Off	On	On	On	On	On	Off	Off	0x79
4	Off	Off	On	On	Off	Off	Off	On	0x33
5	Off	On	Off	On	On	Off	On	On	0x5B
6	Off	On	Off	On	On	On	On	On	0x5F
7	Off	On	On	On	Off	Off	Off	Off	0x70
8	Off	On	On	On	On	On	On	On	0x7F

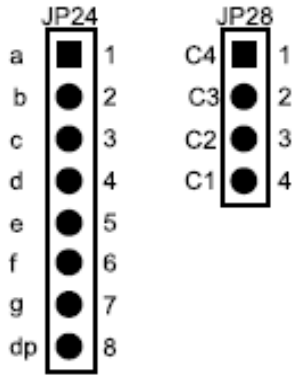
9	Off	On	On	On	On	Off	On	On	0x7B
A	Off	On	On	On	Off	On	On	On	0x77
B	Off	Off	Off	On	On	On	On	On	0x1F
C	Off	On	Off	Off	On	On	On	Off	0x4E
D	Off	Off	On	On	On	On	Off	On	0x3D
E	Off	On	Off	Off	On	On	On	On	0x4F
F	Off	On	Off	Off	Off	On	On	On	0x47

Tab. 2

W typowym module wyświetlacza siedmiosegmentowego wszystkie katody lud anody są ze sobą połączone i wyprowadzone jako jedna końcówka. Mówimy wtedy, że mamy do czynienia z wyświetlaczem ze wspólną katodą lub wspólną anodą. Zazwyczaj wspólną katodę dołącza się do masy, a wspólną anodę do dodatniego napięcia zasilającego.

7.2. Wyświetlacz LED modułu ZL3AVR

Moduł ZL3AVR posiada na swoim „pokładzie" cztery wyświetlacze siedmiosegmentowe ze wspólną anodą, połączone w strukturę multipleksowaną. Oznacza to, że w danym momencie tylko jeden, wybrany, wyświetlacz świeci. Jeżeli obsługa poszczególnych wyświetlaczy będzie odbywać się z częstotliwością co najmniej 25Hz, nasze oczy będą miały złudzenie, że wszystkie wyświetlacze świecą równocześnie (efekt podobny jak w kinie).
Sygnały sterujące wyświetlaczami są dostępne na złączach JP24 (segmenty) i JP28 (anody) – rys. 8.



Rys. 8

Bity C1÷C4 złącza JP28 służą do aktywacji pojedynczego wyświetlacza. Bity a, b, c, d, e, f, g i dp złącza JP24 są wykorzystywane do sterowania segmentami. Aktywnym poziomem sygnału, w obu przypadkach, jest „0".

Przykład: Wyświetlenie cyfry '5' na pierwszym wyświetlaczu. Do sterowania anodami wybrano port B, a segmentami port A.

```
ldi r16, 0xff
out ddra, r16       ;Ustawienie Port A jako wyjście
out ddrb, r16       ;Ustawienie Port B jako wyjście
ldi r16, 0x01       ;Wybór pierwszego wyświetlacza
```

```
        com r16          ;Negacja, celem uzyskania aktywnego zera sterującego
        out portb, r16   ;Aktywacja wyświetlacza poprzez port B
        ldi r16, 0x5B    ;Kod cyfry 5
        com r16          ;Negacja, celem uzyskania aktywnego zera sterującego
        out porta, r16   ;Zaświecenie segmentów odpowiedzialnych za „widzialność" cyfry 5
stop:   rjmp stop
```

Ćwiczenie 4.3

Przekopiuj kod programu z pliku 'PrimesLEDs.asm' i zmodyfikuj go tak, aby na ostatnim wyświetlaczu były wyświetlane cyfry, kodu szesnastkowego, od 0 do F. Zapisz program jako 'SevSegDisp.asm'

8. Operacje na połówkach bajtu

8.1. Instrukcja swap – zamiana miejscami połówek bajtu

Instrukcja ***swap*** wykonuje operację zamiany miejscami połówek bajtu rejestru Rd.

Przed:	d7-d4	d3-d0	Po:	d3-d0	d7-d4

Przed:	1110	1010	Po:	1010	1110

```
        swap Rd      ;0 ≤ d ≤ 31
```

Przykład: Wysłanie młodszej połówki rejestru R1 do młodszej części portu A (3:0) oraz starszej połówki rejestru R1 do młodszej części portu B (3:0).

```
        mov r2,r1    ;r2=r1
        andi r2,0x0f ;Izolacja młodszej połówki bajtu
        out porta, r2 ;Wysłanie do portu A
        mov r2,r1    ;r2=r1
        swap r2      ;Zamiana miejscami połówek bajtu rejestru R2
        andi r2, 0x0f ;Izolacja młodszej połówki bajtu
        out portb, r2 ;Wysłanie do portu B
```

9. Dodawanie liczb bez znaku

Liczby bez znaku reprezentują dane z zakresu 0÷255 (szesnastkowo 0x00÷0xFF).

Do wykonywania operacji dodawania, w mikrokontrolerach AVR, służą głównie instrukcje ***add*** i ***adc***.

add Rd, Rr ;Rd=Rd+Rr

adc Rd, Rr ;Rd=Rd+Rr+C (carry)

Operacja dodawania zmienia m.in. stan flagi C i Z. Flaga C jest ustawiana, jeśli wystąpi przeniesienie (wynik nie mieści się w rejestrze przeznaczenia) z bitu D7 wyniku. Natomiast flaga Z jest ustawiana, gdy wynik operacji wynosi 0.

W przypadku dodawania liczb n-bajtowych (tzw. wielokrotna precyzja), aby wynik był poprawny, musi być uwzględniony stan flagi C na każdym etapie (oprócz pierwszego) dodawania poszczególnych bajtów. Do tego rodzaju operacji jest stworzona instrukcja ***adc***.

9.1. Instrukcja add – dodawanie bez bitu przeniesienia C

Instrukcja ***add*** wykonuje operację dodawania dwóch rejestrów Rd i Rr. Wynik jest umieszczany w rejestrze Rd. Flaga C nie jest uwzględniana.

add Rd,Rr ;0 ≤ d ≤ 31, 0 ≤ r ≤ 31

9.2. Instrukcja adc – dodawanie z uwzględnianiem bitu przeniesienia C

Instrukcja ***adc*** wykonuje operację dodawania dwóch rejestrów Rd i Rr. Wynik jest umieszczany w rejestrze Rd. Flaga C jest dodawana do wyniku.

adc Rd,Rr ;0 ≤ d ≤ 31, 0 ≤ r ≤ 31

Przykład: Dodawanie 16-bitowej liczby 0x1f55 do rejestru Z

```
ldi r16,low($1f55)     ;r16=55
ldi r17, high($1f55)   ;r17=1f
add ZL, r16
adc ZH, r17
```

10. Procedura programowania wyświetlacza LED

Ćwiczenie 4.4

Napisz program realizujący 16-bitowy, programowy licznik, którego bieżąca wartość będzie wyświetlana na czterech wyświetlaczach siedmiosegmentowych.

1. Licznik zrealizuj na dwóch 8-bitowych rejestrach.

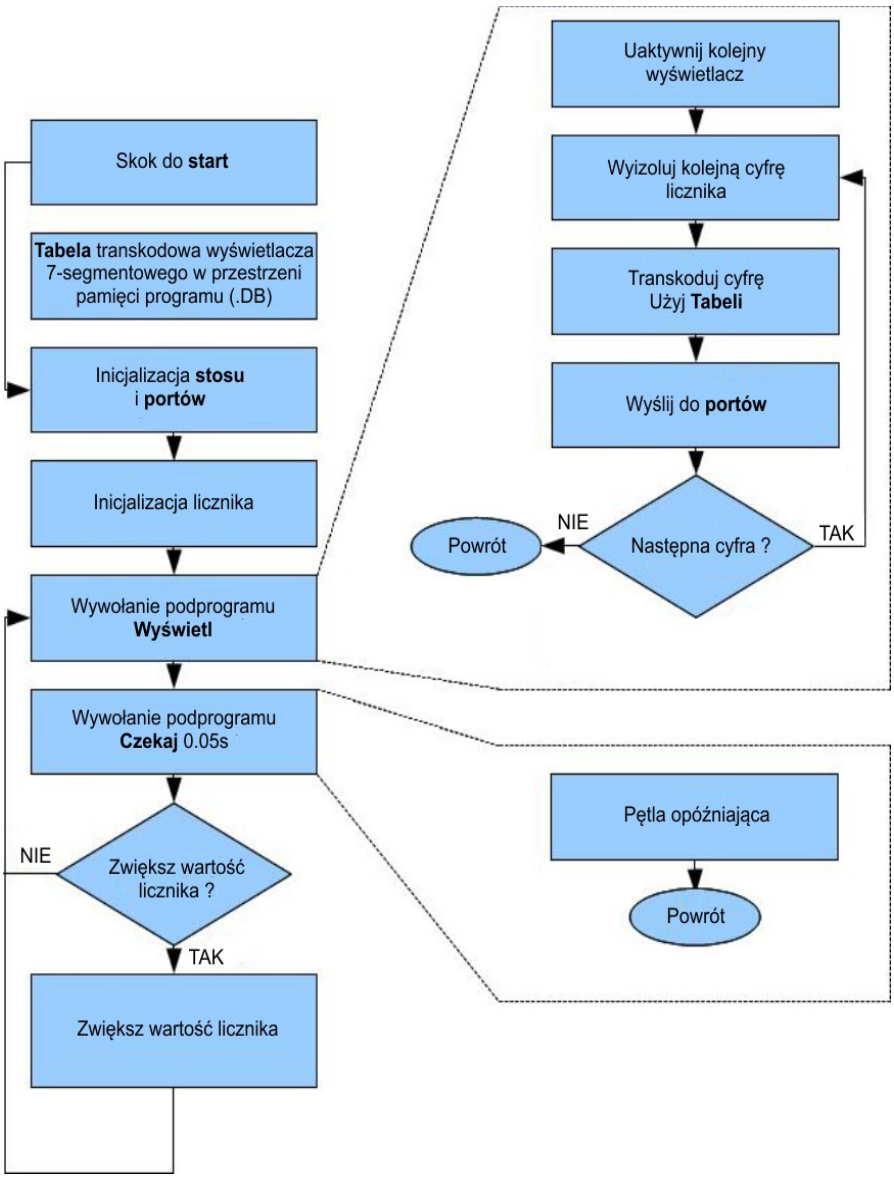
2. Zastosuj podprogram opóźniający.

3. Wyświetlanie wartości licznika napisz w formie podprogramu.

4. Częstotliwość odświeżania wyświetlacza LED - 1/20 sekundy.

5. Częstotliwość modyfikacji licznika - 1/5 sekundy (cztery okresy odświeżające LED).

Proponowany schemat blokowy programu jest zamieszczony na rysunku rys. 9.



Rys. 9

Zapisz program jako 'SevSegDisp4.asm'.