Użycie portów

cp
cpi
nop
inc
dec
brne
breq

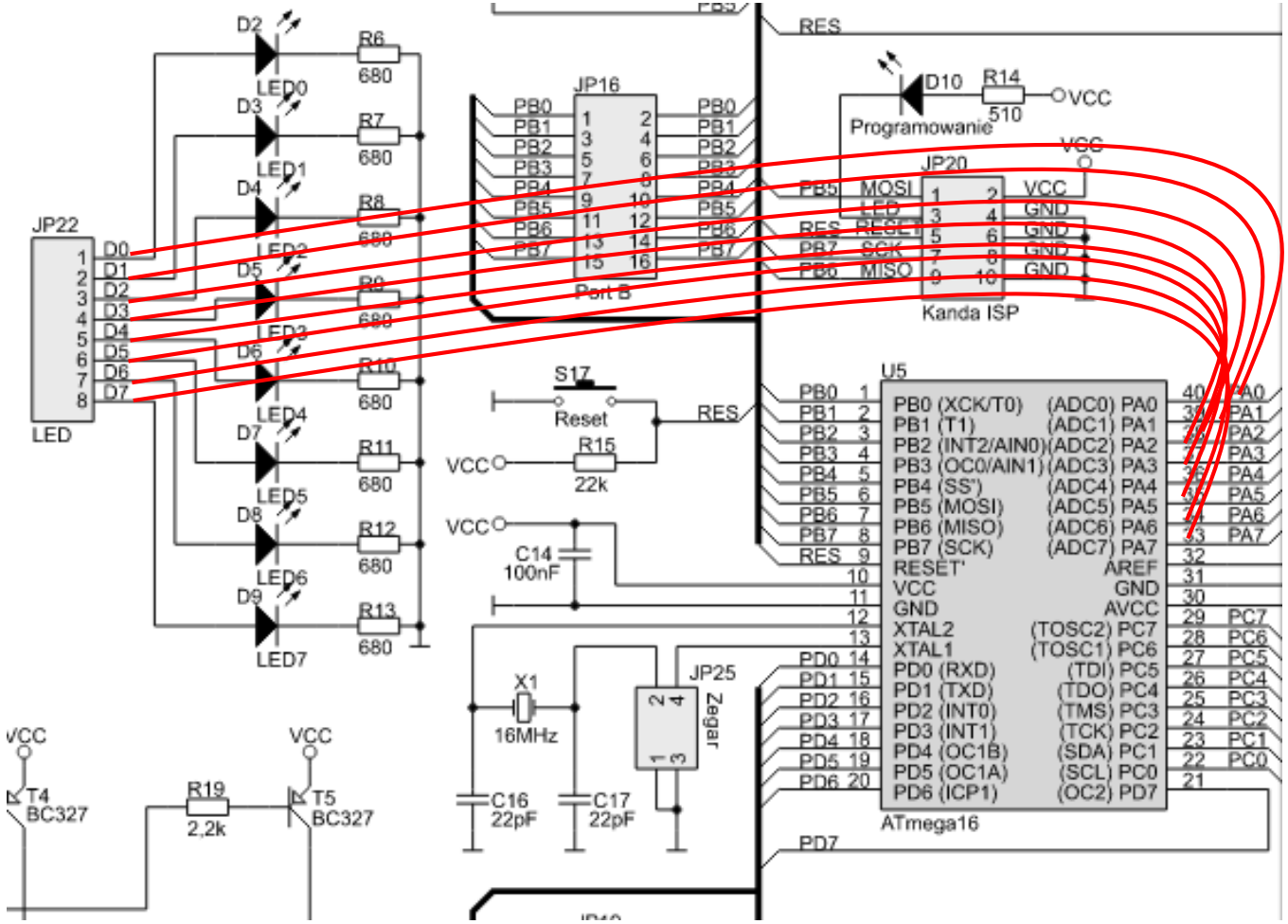
Program – włączenie trzech diod

CELE

1. **Podstawowe informacje o portach**
2. **Programowanie portów jako wejście/wyjście**
3. **Program włączający trzy diody LED**
4. **Zapoznanie się ze środowiskiem programatora USBASP oraz On-Chip Debug emulator JTAG AVR**

WYDZIAŁ FIZYKI
i INFORMATYKI STOSOWANEJ
Uniwersytet Łódzki

Operacje na portach – schemat połączeń



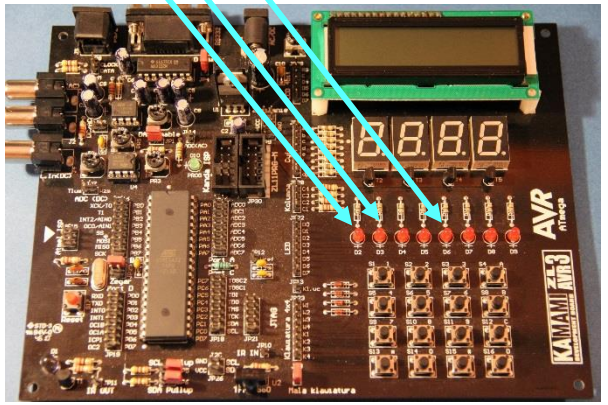
WYDZIAŁ FIZYKI
i INFORMATYKI STOSOWANEJ
Uniwersytet Łódzki

Pierwszy program z użyciem portów

Program – włączenie trzech diod

```
.nolist
.include"m32def.inc"
.list
.cseg           ; dalsza część programu dotyczy pamięci programu
.org 0x0000     ; ustal licznik adresu w pamięci programu na 0

ldi             r16,high(ramend) ; deklaracja stosu ( w tym programie stos nie jest wykorzystany)
out             SPH,r16
ldi             r16, low(ramend)
out             SPL,r16

ldi             r16,0b11111111 ; załaduj do rejestru R16 wartość 11111111
out             ddra,r16       ; Przesłanie zawartości rejestru r16 do rejestru ddra. Cały port A jako port wyjściowy (output)

ldi             r16,0b00010011 ; załaduj do rejestru R16 wartość 00010011
out             porta,r16      ; Przesłanie zawartości rejestru r16 do rejestru porta.
                               ; na wyjściach mikrokontrolera PA0,PA1,PA4 pojawi się „1”
```

włącz diody D2,D3,D6



Interfejsy programowania i uruchomieniowe

Programowania pamięci FLASH

ISP
(In System Programming)

SPI (szeregowy)
Złącze KANDA

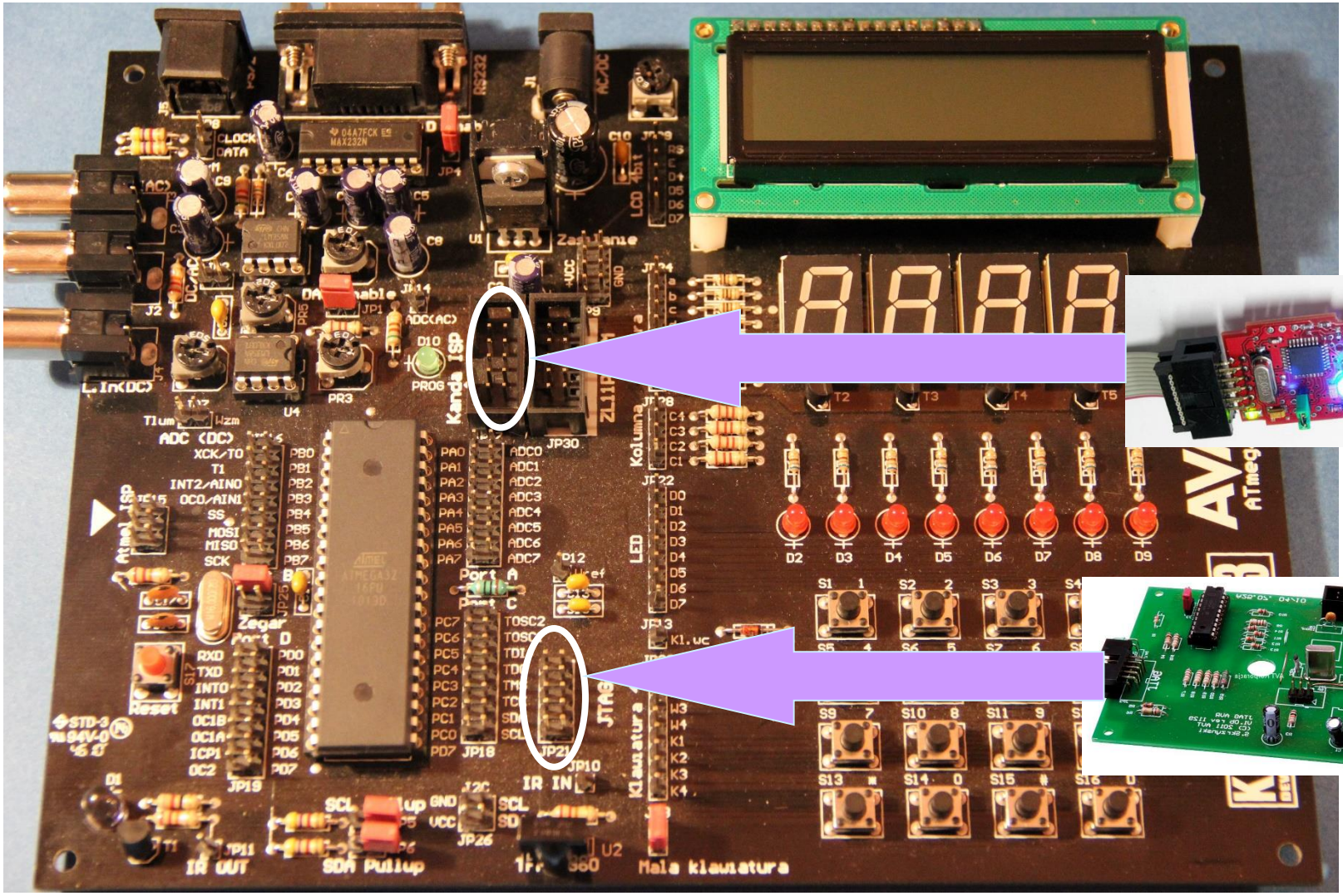
Interfejs JTAG

Poza systemem

Skomplikowane układy
z transmisją równoległą

WYDZIAŁ FIZYKI
i INFORMATYKI STOSOWANEJ
Uniwersytet Łódzki

Zestaw ZL3AVR



Pierwszy program z użyciem portów

Program – włączenie trzech diod

Mikrokontroler wykona wszystkie rozkazy kolejno, a każdy z nich zabierze mu czas 1 ms (poprzez nastawę bitów konfiguracyjnych wybraliśmy na zegar systemowy wewnętrzny oscylator 1 MHz). Co jednak stanie się, gdy układ skończy wykonywanie wszystkich zleconych mu czynności? Otóż zacznie on interpretować wartości umieszczone za ostatnim zaprogramowanym rozkazem jako dalsze instrukcje programu. Wartościami tymi będą liczby 0xFF (odpowiadające niezaprogramowanym komórkom pamięci programu, co wynika z właściwości pamięci typu flash) i to one zostaną wzięte za rozkazy. Dobrze, tylko co właściwie oznacza rozkaz 0xFFFF (jak pamiętamy, rozkazy AVR mają długość dwóch bajtów)? W asemblerze AVR instrukcja o takim kodzie nie istnieje – mikrokontroler zatem może zachować się w sposób nieokreślony (w praktyce rozkaz ten będzie traktowany jak instrukcja nop). Aby temu zapobiec, należy zatrzymać dalsze wykonywanie programu. Dokonamy tego poprzez wprowadzenie instrukcji skoku bezwarunkowego, adresowanego do niego samego:

```
Petla:
   rjmp Petla
```

Mikrokontroler będzie teraz ładował licznik programu wciąż tą samą wartością (przyporządkowaną przez translator etykiecie „Petla”), odpowiadającą położeniu instrukcji rjmp w pamięci programu. Układ wykonywać będzie ciągle skoki nie będzie wykonywał dalszych (w tym przypadku nieprawidłowych) rozkazów – program „stanie”.

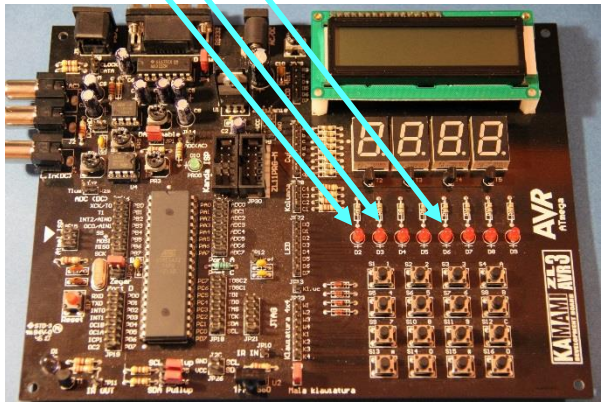
Dla poprawnej kompilacji program kończymy dyrektywą EXIT
.EXIT to dyrektywa powodująca zakończenie asemblacji pliku źródłowego w miejscu, w którym zostaje napotkana. Następujące po niej instrukcje i dyrektywy są przez translator ignorowane. Jeśli dyrektywa ta znajdzie się w pliku dołączanym za sprawą .INCLUDE, translator powróci do przetwarzania pliku głównego.

Pierwszy program z użyciem portów

Program – włączenie trzech diod

```
.nolist
.include"m32def.inc"
.list
.cseg
.org 0x0000

        ldi r16,high(ramend) ; deklaracja stosu ( w tym programie stos nie jest wykorzystany)
        out SPH,r16
        ldi r16, low(ramend)
        out SPL,r16

        ldi r16,0b11111111 ; załaduj do rejestru R16 wartość 11111111
        out ddra,r16       ; Port A jako port wyjściowy (output)

        ldi r16,0b00010011  ; załaduj do rejestru R16 wartość 00010011
        out porta,r16       ; na wyjściach mikrokontrolera PA0,PA1,PA4 pojawi się „1”

Petla:                      ; nieskończona pętla
   rjmp Petla

.exit
```

włącz diody D2,D3,D6



Zadania

Napisać program, który naprzemiennie WŁACZY I WYŁACZY diody D2 i D9 Należy zastosować opóźnienia używając pętli i instrukcji NOP.

Potrzebna będzie instrukcję pętli ze zliczaniem cykli (instrukcja FOR) Wykorzystanie mnemonik inkrementacji dekrementacji oraz mnemoniki wykorzystujących skoki warunkowe.

Porównywanie liczb 1 bajtowych

Relacja: równe
Relacja: nie równe

Relacja: mniejsze
Relacja: większe

Relacja: mniejsze lub równe
Relacja: większe lub równe

WYDZIAŁ FIZYKI
i INFORMATYKI STOSOWANEJ
Uniwersytet Łódzki

Instrukcja cp, cpi

cp, cpc i cpi porównują wartość wybranego rejestru z odpowiednio: drugim rejestrem, drugim rejestrem z przeniesieniem, wartością bezpośrednią. Również te instrukcje nie zmieniają wartości przechowywanych w rejestrach roboczych, operując jedynie na rejestrze SREG.

Mnemonik	Rozwinięcie	Operacja	SREG I T H S V N Z C
cp	**ComPare** porównaj	Rs1-Rs2 PC ← PC+1	- - \ \ \ \ \ \

Mnemonik	Rozwinięcie	Operacja	SREG I T H S V N Z C
cpi	**ComPare with Immediate** porównaj ze stałą	Rh-c256 PC ← PC+1	- - \ \ \ \ \ \

Opisywane rozkazy wykorzystywane są najczęściej do obliczenia warunków dla rozgałęzień. Z użyciem instrukcji testująco-porównujących i odpowiednich rozkazów sterujących możliwe jest kierowanie przepływem programu w zależności od zawartości wybranego rejestru.

Porównywanie liczb 1 bajtowych - Przykłady

Relacja: mniejsze

5<10

```
ldi     r16, 5
ldi     r17, 10
cp      r16,r17
```

r16-r17
5-10=-5
111111011

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I	T	H	S	V	N	Z	C
0	0	1	1	0	1	0	1

Relacja: większe

10>5

```
ldi     r16, 10
ldi     r17, 5
cp      r16,r17
```

r16-r17
10-5=5
00000101

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I	T	H	S	V	N	Z	C
0	0	0	0	0	0	0	0

Relacja: równe

5=5

```
ldi     r16, 5
ldi     r17, 5
cp      r16,r17
```

r16-r17
5-5=0
00000000

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I	T	H	S	V	N	Z	C
0	0	0	0	0	0	1	0

Instrukcja nop

Mnemonik	Rozwinięcie	Operacja	SREG I T H S V N Z C
nop	**No Operation** Brak operacji, instrukcja pusta	PC ← PC+1	- - - - - - - -

składnia
nop

Przykłady

```
ldi r16,0x25
nop
nop
```

Instrukcja inc, dec

Mnemonik	Rozwinięcie	Operacja	SREG I T H S V N Z C
inc	**INCrement** Zwiększa o jeden	PC ← PC+1 Rd←Rd+1	- - - \ \ \ \ -

Mnemonik	Rozwinięcie	Operacja	SREG I T H S V N Z C
dec	**DECrement** Zmniejsza o jeden	PC ← PC+1 Rd←Rd-1	- - - \ \ \ \ -

składnia
inc Rd
dec Rd

Przykłady

```
ldi r16, 255
inc r16
dec r16
```

WYDZIAŁ FIZYKI
i INFORMATYKI STOSOWANEJ
Uniwersytet Łódzki

Instrukcja brne, breq

Mnemoniki z prefiksem br- (BRBS, BRBC, BREQ, BRNE, BRCS, BRCC, BRSH, BRLO, BRMI, BRPL, BRGE, BRLT, BRHS, BRHC, BRTS, BRTC, BRVS, BRVC, BRIE, BRID)
pozwalają na wykonanie skoku pod warunkiem, że wybrany znacznik z rejestru SREG znajduje się w odpowiednio wysokim lub niskim stanie logicznym. Jeśli warunek ten jest spełniony, następuje skok do miejsca wyznaczonego przez argument bezpośredni rozkazu. W przeciwnym razie rozkaz rozgałęzienia działa jak instrukcja nop. Omawiane rozkazy nie zmieniają zawartości rejestru SREG.

Adresowanie skoków warunkowych (rozgałęzień) zachodzi względnie – bezpośredni argument rozkazu musi być wartością w kodzie U2, jaka ma być dodana do licznika programu (PC). Zakres rozgałęzień jest mocno ograniczony – możliwe są przesunięcia do 63 słów wstecz i 64 słów w przód programu.

Mnemonik	Rozwinięcie	Operacja	SREG I T H S V N Z C
brne	**BRanch if Not Equal** skocz, jeśli Z równe 0 w rejestrze SREG	Z=1 ⇒ PC ← PC + 1 Z=0 ⇒ PC ← PC + + adr64 + 1	- - - - - - - - Skoki nie zmieniają rejestru SREG

Mnemonik	Rozwinięcie	Operacja	SREG I T H S V N Z C
breq	**BRanch if EQual** skocz, jeśli Z równe 1 w rejestrze SREG	Z=0 ⇒ PC ← PC + 1 Z=1 ⇒ PC ← PC + + adr64 + 1	- - - - - - - - Skoki nie zmieniają rejestru SREG

breq

Porównywanie liczb 1 bajtowych

Relacja: równe

Relacja: nie równe

```
.include"m32def.inc"
.cseg

loop:
ldi     r16,0x02
ldi     r17,0x02
cp      r16,r17
breq    loop

nop

.exit
```

R16=r17

skocz, jeśli Z równe 1 w rejestrze SREG

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I	T	H	S	V	N	Z	C
0	0	0	0	0	0	1	0

```
.include"m32def.inc"
.cseg

loop:
ldi     r16,0x06
ldi     r17,0x02
cp      r16,r17
breq    loop

nop

.exit
```

R16>r17

```
.include"m32def.inc"
.cseg

loop:
ldi     r16,0x02
ldi     r17,0x06
cp      r16,r17
breq    loop

nop

.exit
```

R16<r17

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I	T	H	S	V	N	Z	C
0	0	0	0	0	0	0	0

brne

Porównywanie liczb 1 bajtowych

Relacja: równe

Relacja: nie równe

```
.include"m32def.inc"
.cseg

loop:
ldi     r16,0x02
ldi     r17,0x02
cp      r16,r17
brne    loop

nop

.exit
```

R16=r17

dalej, jeśli Z równe 1 w rejestrze SREG

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I	T	H	S	V	N	Z	C
0	0	0	0	0	0	1	0

```
.include"m32def.inc"
.cseg

loop:
ldi     r16,0x06
ldi     r17,0x02
cp      r16,r17
brne    loop

nop

.exit
```

R16>r17

```
.include"m32def.inc"
.cseg

loop:
ldi     r16,0x02
ldi     r17,0x06
cp      r16,r17
brne    loop

nop

.exit
```

R16<r17

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I	T	H	S	V	N	Z	C
0	0	0	0	0	0	0	0

brne → Skoki warunkowe **Pętli iteracyjnej - for**

Istnieje oczywiście więcej rozkazów, których interpretacja jest zależna od aktualnej zawartości rejestru SREG. Należą do nich instrukcje skoków warunkowych (ich mnemoniki charakteryzują się prefiksem br-), których działanie jest zależne od stanu wybranego znacznika ze SREG. Skoki warunkowe będziemy wykorzystywać niezwykle często. Dla przykładu pokażemy sposób implementacji pętli iteracyjnej (**na podobieństwo konstrukcji for**) z wykorzystaniem instrukcji brne, testującej znacznik Z (jeśli Z=0, rozkaz ten powoduje wykonanie skoku do wybranego punktu programu):

Stosując rozkaz dec (a także inc) należy pamiętać o pułapce, jaka czyha na początkujących – instrukcje te nie aktualizują znacznika przeniesienia (C). W ich przypadku należy więc testować znacznik wartości zerowej (Z).

brne Pętla for wykonana 255 razy

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I	T	H	S	V	N	Z	C
0	0	0	0	0	0	0	0

skok do etykiety go_1: jeśli rejestr R17 jest niezerowy,
brne testuje bit Z rejestru SREG Z=0 wykonany jest skok.

```
ldi r17,255        ; załaduj do rejestru r17 wartość, 255 dla pętli for i=0 to 255
go_1:              ; etykieta pętli na podobieństwo konstrukcji for
   nop             ; brak operacji trwający jeden takt zegara
   nop

 dec r17           ; zmniejszenie rejestru r17 o jeden, instrukcja ta dokonuje zmian w rejestrze SREG

 brne go_1
```

R17≠0

R17=0

nop

Dalej, wyjście z pętli, jeśli Z równe 1
w rejestrze SREG

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I	T	H	S	V	N	Z	C
0	0	0	0	0	0	1	0

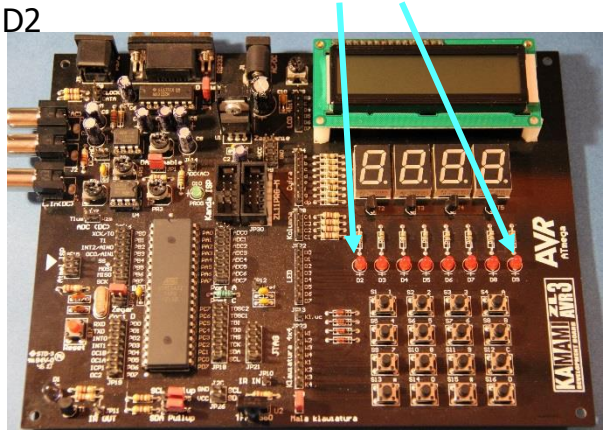
WYDZIAŁ FIZYKI
i INFORMATYKI STOSOWANEJ
Uniwersytet Łódzki

Zadanie program z użyciem portów

Program, który naprzemiennie WŁACZY I WYŁACZY diody D2 i D9

```
.nolist
.include"m32def.inc"
.list
.cseg
.org 0x0000

        ldi  r16,high(ramend) ; deklaracja stosu ( w tym programie stos nie jest wykorzystany)
        out  SPH,r16
        ldi  r16, low(ramend)
        out  SPL,r16

        ldi  r16,0b11111111    ; cały porta A jako wyjście
        out ddra,r16
Petla:
        ldi r16,0b00000001
        out porta,r16          ; włączenie diody D2 i wyłączenie diody D9
Tu musi być opóźnienie

        ldi r16,0b10000000     ; włączenie diody D9 i wyłączenie diody D2
        out porta,r16
Tu musi być opóźnienie

        jmp petla

.exit
```

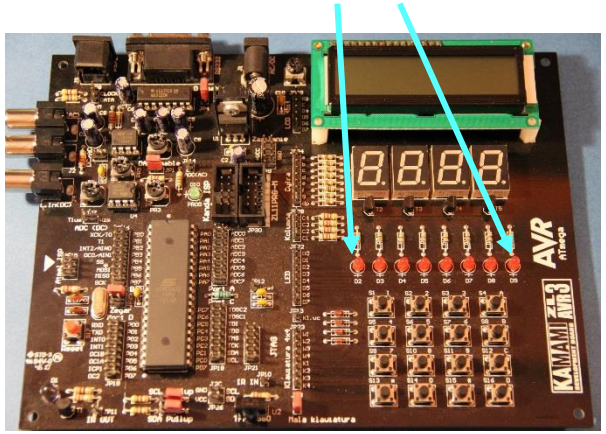
Wł/wy diody D2,D9



Program, który naprzemiennie WŁACZY I WYŁACZY diody D2 i D9 z opóźnieniem przy użyciu pętli for

```
.nolist
.include"m32def.inc"
.list
.cseg
.org 0x0000
        ldi r16,high(ramend)
        out SPH,r16
        ldi r16, low(ramend)
        out SPL,r16

        ldi r16,0b11111111              ; porta A jako wyjście
        out ddra,r16
Petla:
        ldi r16,0b00000001              ; włącz diodę D2
        out porta,r16

        ldi r17,255       ; załaduj do rejestru r17 wartość 255
        go_1:             ; etykieta pętli na podobieństwo konstrukcji for
          nop             ; brak operacji trwający jeden takt zegara
          nop
          dec r17         ; zmniejszenie rejestru r17 o jeden
        brne go_1         ; skok do etykiety go_1: jeśli rejestr R17 jest niezerowy, brne testuje bit Z
                            rejestru SREG Z=0 wykonany jest skok.

        ldi r16,0b10000000 ; włącz diodę D9
        out porta,r16

        ldi r17,225
        go_2:
          nop
          nop
          nop
          nop
          dec r17
        brne go_2

        jmp Petla
.exit
```

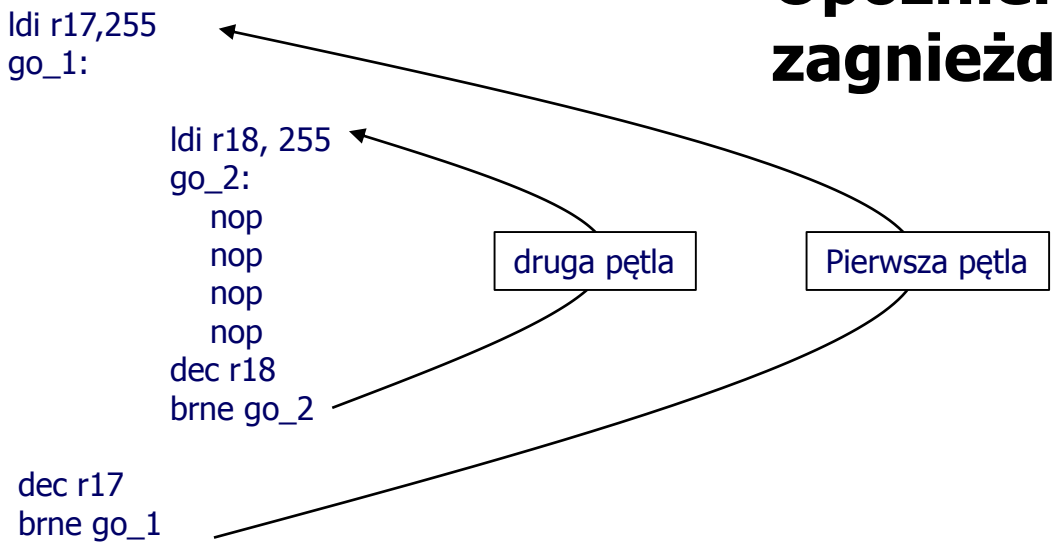
Wł/wy diody D2,D9



Zadania

Napisać program który naprzemiennie WŁĄCZY I WYŁACZY diody D2 i D9 z opóźnieniami używając zagnieżdżonych pętli i instrukcji NOP do generujących opóźnienia.

Opóźnienie, które używa zagnieżdżonych pętli



Zadania

Napisać program który naprzemiennie WŁĄCZY I WYŁĄCZY diody D2 i D9 z opóźnieniami używając procedury wykonującej opóźnienie.

Jak wykonać kilku milisekundowe opóźnienie w kodzie programu ?

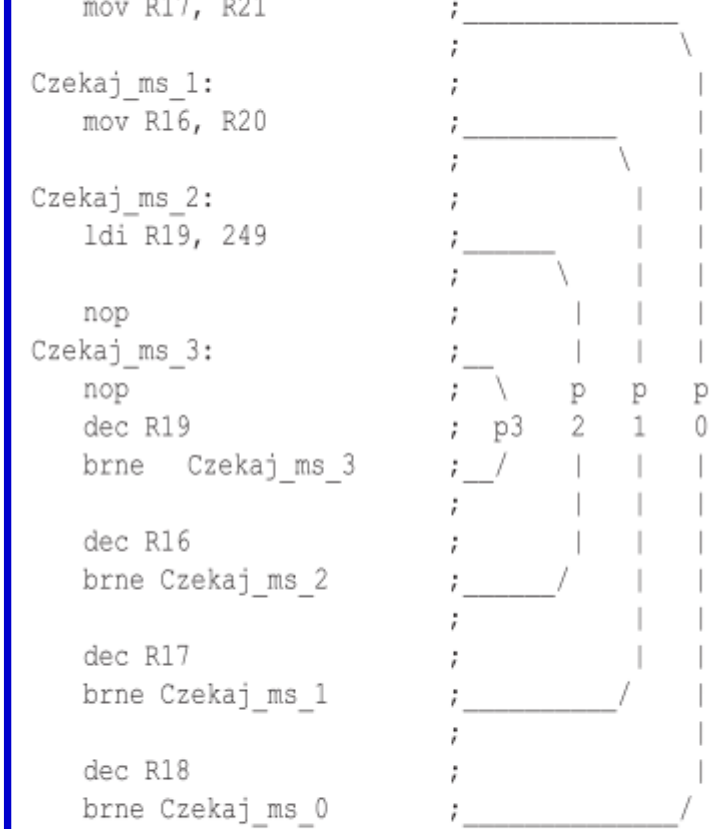
Musimy czekać!!!

ale jak zmusić nasz pracowity mikrokontroler do bezczynności ?

Jednym ze sposobów będzie zajęcie mikrokontroler czymś zupełnie nonsensownym. Możemy przykładowo zwiększać zawartości pewnego rejestru do jego przepełnienia. Jeśli dany rejestr będziemy w ten sposób przepełniać wielokrotnie (np. tyle razy, ile potrzeba do przepełnienia innego rejestru – pętle przepełniające możemy zagnieździć), to opóźnienie będzie się wydłużać (liczba operacji potrzebnych do zakończenia pętli będzie równa iloczynowi operacji potrzebnych do przepełnienia jednego i drugiego rejestru). Procedura opóźniająca mogłaby wyglądać zatem następująco (wprowadzono aż cztery rejestry, których wartość jest zmniejszana w zagnieżdżonych pętlach):

Zadania

```
;************************************************************
;      MILISEKUNDOWA PROCEDURA OPÓŹNIAJĄCA
;
; Parametry:
;   R16 - wartość opóźnienia w ms (dla zera - 256 ms)
;   R17 - mnożnik opóźnienia (dla zera - 256x)
;
; Wymagane stałe:
;   SYS_FREQ - częstotliwość pracy w MHz
;

.EQU SYS_FREQ = 1              ; częstotliwość pracy w MHz

Czekaj_ms:
   push R18
   push R19
   push R20
   push R21
   mov R20, R16
   mov R21, R17
   ldi R18, SYS_FREQ

Czekaj_ms_0:
   mov R17, R21
Czekaj_ms_1:
   mov R16, R20
Czekaj_ms_2:
   ldi R19, 249
   nop
Czekaj_ms_3:
   nop
   dec R19
   brne   Czekaj_ms_3
   dec R16
   brne Czekaj_ms_2
   dec R17
   brne Czekaj_ms_1
   dec R18
   brne Czekaj_ms_0

   pop R21
   pop R20
   pop R19
   pop R18
   ret
```



Zadania

Przed wywołaniem powyższego programu należy określić długość wymaganego opóźnienia – będzie to tzw. parametr wywołania. W asemblerze oczywiście nie istnieje nic na podobieństwo procedur wywoływanych z parametrami, które znamy z języków wysokiego poziomu, ale możemy stworzyć ich namiastkę. Sposobów można byłoby wymienić kilka: przekazywanie przez stos (nieco kłopotliwe), poprzez zmienne w pamięci SRAM, przez rejestry robocze (rozwiązanie najprostsze). W przypadku powyższej procedury zastosowano wariant ostatni – poprzez rejestr R16 należy przekazać liczbę, określającą długość wymaganego opóźnienia w milisekundach (1...255, zeru odpowiada 256 ms). Dodatkowo, do rejestru R17 należy załadować mnożnik, który zwiększa zakres dostępnych opóźnień. Szybkość wykonywania procedury jest zależna oczywiście od prędkości zegara systemowego. Aby ją od tego czynnika uniezależnić, wprowadzono stałą SYS_FREQ. Etykiecie tej należy przypisać wyrażoną w megahercach częstotliwość pracy mikrokontrolera (założono, że minimalną częstotliwością systemową jest 1 MHz). Wartość ta jest ładowana do rejestru R18 i determinuje ona liczbę przebiegów głównej pętli opóźniającej. Najgłębiej zagnieżdżony rejestr procedury (R19) jest ładowany zawsze tą samą wartością początkową, równą 249. Wykonywanie pętli przepełniającej trwa w jego przypadku dokładnie 1000 cykli zegarowych (250 razy czas wykonywania instrukcji nop, dec i brne przy spełnionym warunku), co przy odpowiednio dobranej wartości SYS_FREQ powinno odpowiadać opóźnieniu o długości 1 ms.

Program, który naprzemiennie WŁĄCZY I WYŁĄCZY diody D2 i D9 z opóźnieniami używając procedury wykonującej opóźnienie.

```
.nolist
.include"m32def.inc„
.EQU SYS_FREQ = 1
.list
.cseg
.org 0x0000

        ldi  r16,high(ramend) ; deklaracja stosu ( w tym programie stos będzie używany przez
        out  SPH,r16          ; procedurę Czekaj_ms )
        ldi  r16, low(ramend)
        out  SPL,r16

        ldi  r16,0b11111111   ; cały porta A jako wyjście
        out ddra,r16
Petla:
        ldi r16,0b00000001
        out porta,r16         ; włączenie diody D2 i wyłączenie diody D9

    ldi R16, 20
    ldi R17, 100
    Call czekaj_ms

        ldi r16,0b10000000    ; włączenie diody D9 i wyłączenie diody D2
        out porta,r16

    ldi R16, 20
    ldi R17, 100
    Call czekaj_ms

jmp petla


.INCLUDE ”czekaj_ms.inc”

.exit
```

wywołaj procedurę opóźniającą (t=100*20ms = 1s)

W tym miejscu możemy umieścić kod procedury opóźniającą Czekaj_ms:

lub

Makra i podprogramy przechowywać będziemy zazwyczaj w dodatkowych plikach, które dołącza się dyrektywami .INCLUDE. Dyrektywy te w trakcie asemblacji zastąpione są zawartością dołączanych plików (procedurami lub makrami). W aktualnym ćwiczeniu możemy uprościć kod aby był bardziej czytelny i procedurę opóźniającą – zapiszemy w osobnym pliku z rozszerzeniem .inc (Czekaj_ms.ini)