Programowanie niskopoziomowe

Instrukcja do ćwiczeń laboratoryjnych.

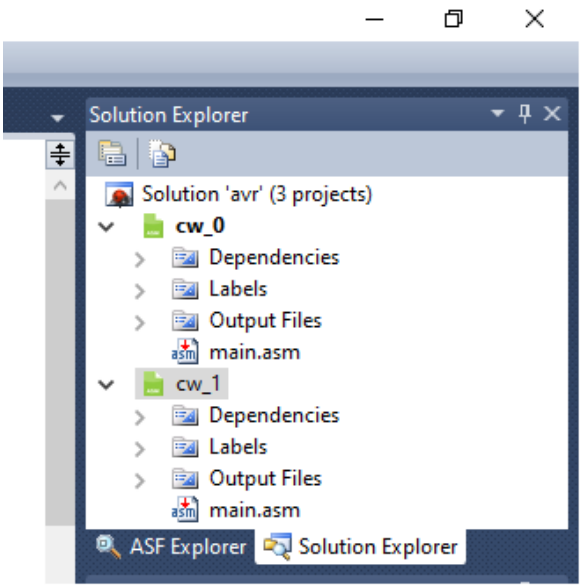
Architektura AVR

2020.09.30

Spis treści

1. Obsługa środowiska Atmel Studio

W środowisku AtmelStudio poszczególne Projekty (ilustracja: cw_0 i cw_1) wchodzą w skład Solucji (poniżej AVR).

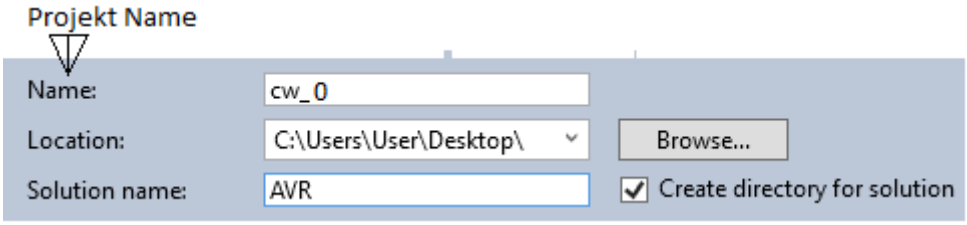


Jednym z elementów Projektu może być kod programu napisanego np. w Assemblerze (ilustracja main.asm).

Programy poszczególnych ćwiczeń (znajdujące się w plikach *.asm) będą przechowywane w oddzielnych projektach
(jedno ćwiczenie – jeden projekt - jeden plik *.asm).

Nazwy projektów powinny odpowiadać numerom ćwiczeń.
Plik źródłowy każdego projektu powinien nazywać się main.asm

1. Uruchomić *Atmel Studio*, wybrać *New Project* (lewy górny róg) a następnie ustawić *Assembler* (lewy górny róg) oraz inne parametry jak poniżej i przejść do następnego okna (*OK*).



2. Wybrać rodzinę (*Device Family*) „tinnyAVR, 8-bit” oraz mikrokontroler „Atinny2313”, przejść do następnego okna.
3. Ustawić symulator jako narzędzie do debugowania:
   - *Menu\Project\Properties\Tool\Selected Debuger|Programmer*: „Simulator”

     UWAGA: jeżeli opcja *Simulator* nie jest dostępna należy skopiować plik:
     C:\Program Files (x86)\Atmel\Atmel USB Drivers\Jungo\usb64\wdapi1010.dll
     do katalogu:
     C:\Program Files (x86)\Atmel\Atmel Studio 6.2\atbackend
     a następnie rozpocząć wykonywanie instrukcji od początku, pamiętając o uprzednim usunięciu stworzonego katalogu.

   - Zamknąć aktualną zakładkę
   - Zmienić nazwę pliku asm aktualnego projektu na „main.asm”
   - Zapisać projekt (*Ctrl+S* lub ikona z dyskietką).

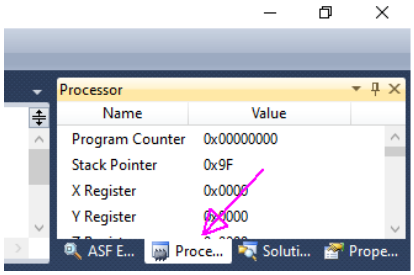
4. W zakładce z plikiem *.asm wstawić, zamiast aktualnej zawartości, poniższy kod (poprawić formatowanie, jeśli się zepsuło).

```
      ldi R16, 5
Loop: nop
      nop
      nop
      dec R16
      rjmp Loop
```

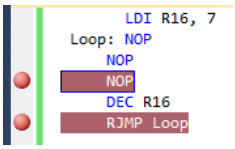
5. Skompilować program (*F7*), a następnie sprawdzić czy wynik kompilacji jest poprawny (Okienko *Output* na samym dole, ostatni wiersz).

6. Krokować program (*F10*, klawisz automatycznie uruchamia tryb debugowania).
W trakcie krokowania, w oknie *Processor*, obserwować zachowanie:
Licznika Programu, Licznika Cykli oraz rejestru `R16` w trakcie kilku iteracji pętli.

   UWAGA: Jeżeli z jakiegoś powodu okno *Processor* nie jest widoczne, można go wyświetlić klikając na odpowiednią zakładkę.



7. Zresetować *debuger* (*Shift+F5*), a następnie ustawić *breakpointy* jak poniżej (Podwójne klikniecie na szarym pasku po lewej stronie instrukcji).



   Wyzwalać symulację bez krokowania (praca ciągła) (*F5*), obserwując działanie symulatora.

2. Rejestry

Każdy mikrokontroler AVR posiada min. 16 8-bitowych rejestrów roboczych (`R0-R15`). Służą one do przechowywania danych wejściowych oraz wyników wykonania niektórych instrukcji. Przykład instrukcji wykonującej operację `R0=R0+R1` pokazano poniżej.

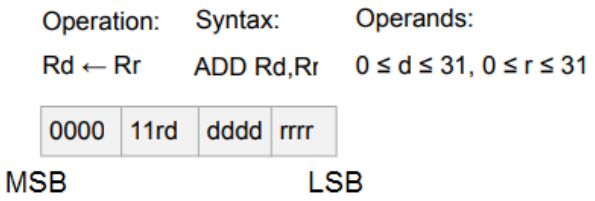
```
add R0,R1
```

3. Kodowanie instrukcji

Instrukcje programu przechowywane są w pamięci programu mikrokontrolera. Pojedyncza komórka pamięci programu ma rozmiar 16 bitów. Większość instrukcji mieści się w pojedynczej komórce pamięci.

	Pamięć programu	
Instrukcje assemblera	**Adres**	**Dana**
`ldi R18, 0xFF`	`0x0000`	`0xef2f`
`ldi R19, 0x01`	`0x0001`	`0xe031`
`eor R18, R19`	`0x0002`	`0x2723`
`out PORTD, R18`	`0x0003`	`0xbb22`

Poniżej pokazano sposób zakodowania instrukcji `add` (tzw. opkod). Sześć najstarszych bitów zawiera kod instrukcji (`000011`), czyli decyduje o tym jaka instrukcja ma być wykonana. Pozostałe bity (`r` i `d`) decydują o tym, na jakich rejestrach instrukcja ma wykonać operację (numery rejestrów). Należy zwrócić uwagę, że w przypadku przykładowej instrukcji, na oba argumenty instrukcji (`Rd` i `Rr`) przeznaczono po pięć bitów co pozwala zaadresować wszystkie 31 rejestrów roboczych.



Przykładowo, słowo odpowiadające instrukcji `add R0,R31` wygląda następująco :

`0000 1110 0000 1111` binarnie, `0x0e0F` heksadecymalnie.

Listę instrukcji mikrokontrolerów AVR można znaleźć w *Atmel-0856-AVR-Instruction-Set-Manual.pdf*

4. Instrukcje arytmetyczne i transferu danych

Instrukcje mikrokontrolerów AVR można podzielić na kilka grup. Są to między innymi instrukcje arytmetyczne (przykład powyżej) oraz transferu danych.

Transfer danych może zachodzić pomiędzy rejestrami roboczymi (np. `mov R0,R1`; *R0←R1*).

Może to być również wpisanie stałej do rejestru. Poniżej pokazano sposób zakodowania instrukcji `ldi` (*Load Immediate*), która zapisuje ośmiobitową stałą do określonego rejestru roboczego. Należy zwrócić uwagę, że numer rejestru roboczego zapisany jest na czterech, a nie na pięciu bitach. Pozwala to zaadresować tylko 16 rejestrów. W przypadku instrukcji `ldi` są to rejestry `R16-R31`.

Przykład instrukcji `ldi`, która zapisuje do wybranego rejestru liczbę 17: `ldi R30,17`.

Operation:	Syntax:	Operands:
Rd ← K	LDI Rd,K	16 ≤ d ≤ 31, 0 ≤ K ≤ 255

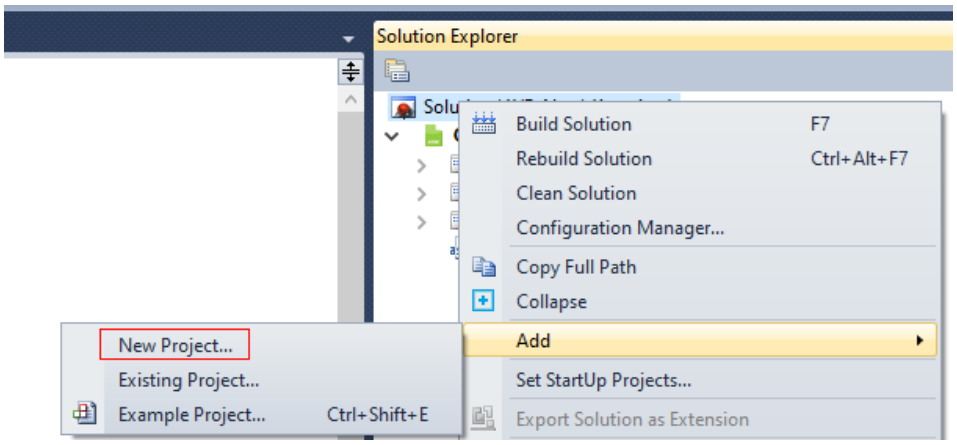
1110	KKKK	dddd	KKKK

Ćwiczenie 1. Poniżej pokazano przykład programu, który dodaje do siebie dwie liczby, przy czym wynik dodawania zapisywany jest w rejestrze `R20`.

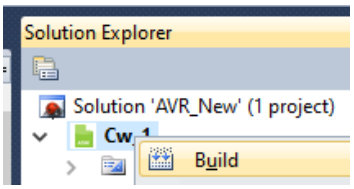
```
ldi R20, 14
ldi R21, 17
add R20, R21
```

a) Stworzyć nowy projekt w ramach bieżącej Solucji.

PRYPOMNIENIE: nazwa projektu powinna być zgodna z numerem ćwiczenia, czyli „cw_1”



b) Wstawić powyższy kod programu do pliku *.asm w projekcie, a następnie skompilować Projekt (nie cała Solucję).



c) Podczas krokowania obserwować zawartość rejestrów użytych w programie

d) Zmodyfikować powyższy program tak, aby wynik operacji znalazł się w rejestrze `R0`. Program powinien zawierać jedną i tylko jedną operację dodawania wyglądającą w następujący sposób: `add R0,R1` (uwaga na ograniczenie instrukcji `ldi`).

Kopiowanie projektu

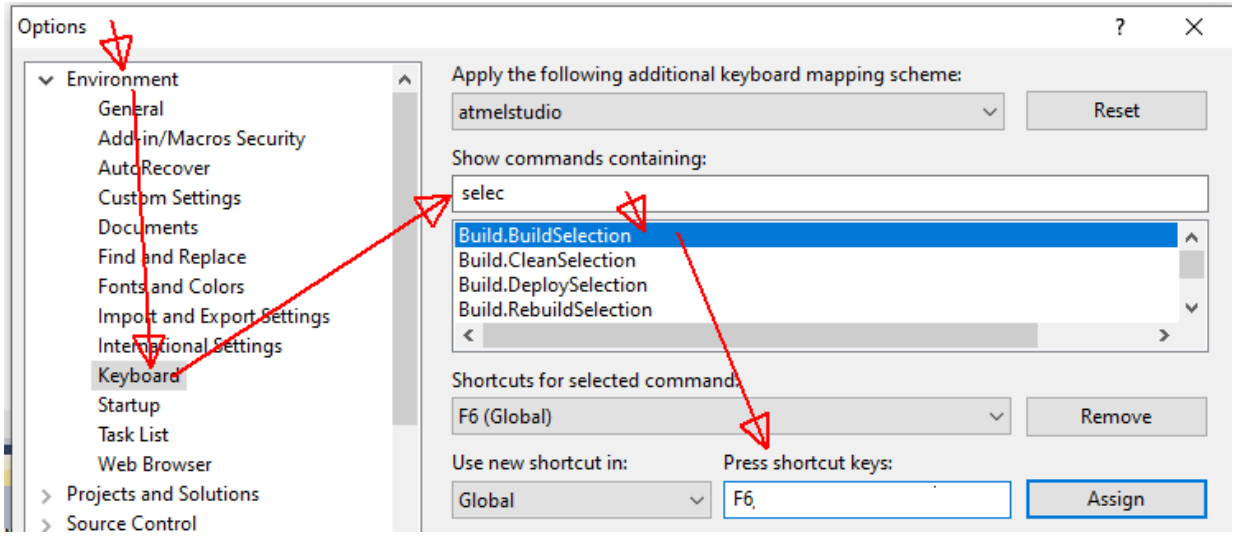
Tworzenie projektu może być uciążliwe, dlatego istnieje możliwość skopiowania projektu:

1. Poza *Atmel Studio* otworzyć katalog z Solucją (powinny być widoczne katalogi projektów)
2. Skopiować katalog projektu pod nazwą nowego projektu (np. „cw_2”)
3. Wejść do skopiowanego katalogu i ustawić nazwę pliku projektu na taką samą jak nazwa katalogu
4. W *Atmel Studio* dodać do Solucji utworzony projekt:
   a. *ikona Solucji\Menu kontekstowe\Add\Existing Project*
   b. wejść do katalogu projektu i dodać plik typu *assembler project* (powinien być jeden taki plik).

Kompilacja aktywnego projektu za pomocą skrótu klawiaturowego

Kompilacja wybranego projektu za pomocą myszki i menu jest w praktyce uciążliwa.
Istnieje możliwość ustawienia kompilacji aktywnego projektu za pomocą skrótu klawiaturowego:

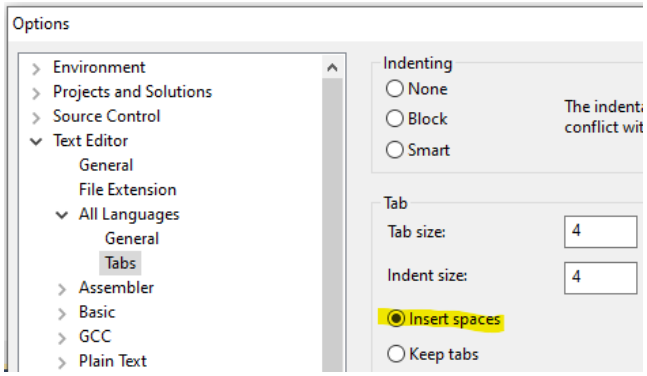
Menu\Tools\Options



Automatyczna zamiana tabulatorów na spacje

Sugeruje się włączyć automatyczną zamianę tabulatorów na spacje

Menu\Tools\Options



Formatowanie kodu

Sugeruje się uwypuklanie fragmentów bloków kodu przy pomocy wcięć. Dodawanie/usuwanie wcięć zaznaczonych bloków kodu uzyskuje się za pomocą przycisków *Tab/Shift+Tab*.

Należy konsekwentnie używać dużych lub małych liter do etykiet, instrukcji i nazw rejestrów.

5. Instrukcje arytmetyczne a rejestr statusu

Oprócz rejestrów roboczych, mikrokontrolery AVR posiadają także rejestr statusu (`SREG`).

Zawiera on osiem flag. Jedna z nich służy do globalnego blokowania/odblokowywania przerwań. Pozostałe siedem z nich odzwierciedla pewne cechy wyniku ostatnio wykonanej operacji arytmetycznej lub logicznej.

Flaga Z

Jedną z flag w `SREG` jest flaga `Z`. Jest ona ustawiana/kasowana, gdy wynikiem operacji arytmetycznej lub logicznej było zero/nie zero.

Ćwiczenie 2. Napisać i przekrokować program, który zapisze do rejestru `R0` liczbę 3, a następnie trzykrotnie odejmie od niej liczbę 1. Zaobserwować stan flagi `Z` po każdej instrukcji odejmowania (5 instrukcji do wyboru).

Flaga C

Wynik dodawania dwóch liczb ośmiobitowych może przekroczyć pojemność rejestru roboczego (czyli pojemność zmiennej 8-bitowej, tzw. przepełnienie). W efekcie, w rejestrze przeznaczenia pozostaje wartość, która jest resztą z dzielenia przez 256 (modulo 2^8).

Ćwiczenie 3. Zmodyfikować jeden z poprzednich programów tak, aby dodał do siebie liczby `100` i `200`. Sprawdzić czy wynik operacji jest resztą z dzielenia przez 256.

Istnieje możliwość zarejestrowania faktu przepełnienia. Służy do tego flaga `C` z rejestru statusu. Można powiedzieć, że flaga `C` wraz z rejestrem przeznaczenia ostatnio wykonanej instrukcji arytmetycznej/logicznej tworzą jeden rejestr 9-bitowy. 9 bitów wystarczy do zapisania wyniku z dodawania dowolnych dwóch liczb 8-bitowych. Innymi słowy, flaga `C` w przypadku dodawania sygnalizuje fakt przeniesienia.

Ćwiczenie 4. Sprawdzić czy program z poprzedniego ćwiczenia powoduje zapalenie flagi. Dodać na koniec programu sekwencję instrukcji, która spowoduje zgaszenie flagi `C`.

Flaga `C` może być modyfikowana przez instrukcję, ale może też stanowić jej argument.

Ćwiczenie 5. Napisać program, który doda do siebie liczby `100` i `200` znajdujące się w rejestrach odpowiednio `R20` i `R21`, w taki sposób, że osiem młodszych bitów wyniku znajdzie się w rejestrze `R20`, a najstarszy dziewiąty bit wyniku znajdzie się w rejestrze `R21` (jako *LSB*). (Uwaga: Sprawdzić, w której instrukcji dodawania flaga `C` jest argumentem operacji. Ograniczyć się do instrukcji przesyłania danych i dodawania.)

```
ldi R20,100
ldi R21,200
add R20,R21
ldi R21,0
adc R21,R21
```

Ćwiczenie 6. Wykorzystując instrukcje z poprzednich programów napisać program, który poprawnie doda dwie liczby 16-bitowe, `300` i `400`. Pierwsza z liczb powinna być zapisana w rejestrach `R20` (młodsza część) i `R21` (starsza część). Druga w rejestrach `R22` i `R23`. Wynik powinien znaleźć się w rejestrach `R20` (młodsza część) i `R21` (starsza część). (Podpowiedź: przypomnieć sobie dodawanie w słupkach)

Flaga `C` wykorzystywana jest również w operacji odejmowania. Jest ona ustawiana w przypadku, gdy od mniejszej liczby odejmowana jest większa, albo kasowana przeciwnym razie. Innymi słowy, flaga `C` w przypadku odejmowania sygnalizuje fakt pożyczki.

Ćwiczenie 7. Znaleźć a następnie sprawdzić działanie instrukcji odejmowania w wersji modyfikującej flagę `C`.

6. Skoki bezwarunkowe

Wykonanie instrukcji w dotychczasowych programach następowało zawsze w kolejności, w jakiej występowały one w programie. Istnieje grupa instrukcji, które pozwalają na zmianę wspomnianej kolejności. Są to instrukcję skoku. Instrukcje te operują na rejestrze zwanym `PC` (ang. *Program Counter*). Zadaniem rejestru jest wskazywać na komórkę pamięci programu, z której ma być pobrana następna instrukcja do wykonania. W przypadku większości instrukcji rejestr `PC` jest po prostu inkrementowany (lub zwiększany o 2 lub 3, w zależności od rozmiaru instrukcji). W przypadku instrukcji skoku rejestr `PC` jest modyfikowany w sposób, który pozwala na „skok" programu do określonego miejsca w programie.

Poniżej pokazano fragment specyfikacji przykładowej instrukcji skoku (`rjmp`). Jest to instrukcja skoku względnego. Jak widać, do licznika programu jest dodawana 12-bitowa liczba określona w argumencie instrukcji (`k`). Liczba ta jest interpretowana jako liczba ze znakiem (U2), czyli może ona przyjmować wartości ujemne. Pozwala to wykonywać skoki nie tylko do instrukcji znajdujących się za instrukcją skoku, ale również przed nią. To z kolei pozwala na wykonywanie pętli znanych z języków wysokiego poziomu.

Operation:	Syntax:	Operands:
PC ← PC + k + 1	RJMP	k -2K ≤ k < 2K

1100	kkkk	kkkk	kkkk

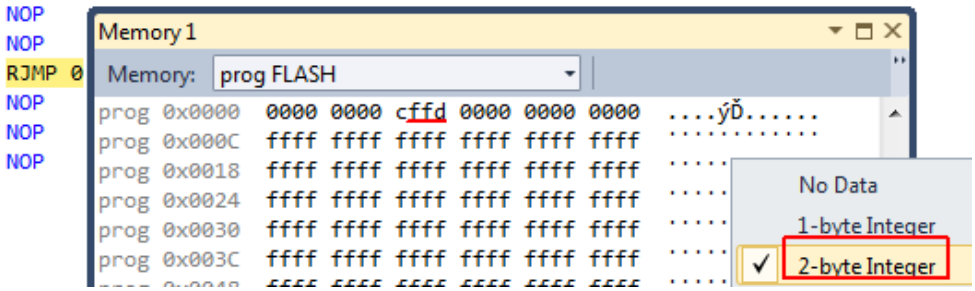
Poniżej pokazano przykład programu zawierającego instrukcję `rjmp`. W celu zwiększenia przejrzystości wykorzystano w niej instrukcję `nop` (nic nie rób). (Należy zapamiętać, że opkod tej instrukcji jest równy `0x0000`.) Można również założyć, że instrukcje znajdują się w pamięci kodu począwszy od adresu zerowego.

```
nop
nop
rjmp 0
nop
nop
nop
```

Ćwiczenie 8. Przekrokować powyższy program – sprawdzić do którego adresu skacze instrukcja `rjmp`. Następnie zwiększać argument instrukcji `rjmp` o 1, obserwując jak wpływa to na działanie programu.

Jak widać, liczba wpisywana jako argument instrukcji `rjmp` określa adres w pamięci kodu, do którego nastąpi skok.

Ćwiczenie 9. Powtórzyć czynności z programu, jednocześnie notując dla każdej wartości argumentu od 0 do 4, jaki był opkod instrukcji `rjmp` wygenerowany przez kompilator. W tym celu, po wejściu w tryb debugowania, należy w oknie kodu ustawić wyświetlanie w trybie 2-bajtowym.



Jak można zauważyć, część opkodu reprezentująca adres skoku (`ffd`, 3 najmłodsze nibble) różni się od argumentu instrukcji skoku (`0`). Argument instrukcji skoku oznacza bezwzględny adres, do którego ma nastąpić skok (liczby większe od zera) natomiast wspomniana część opkodu określa przesuniecie względem aktualnego stanu rejestru *Program Counter*. Przesunięcie jest zapisane w kodzie U2 i w przypadku instrukcji `rjmp` może przyjmować wartości od -2k do 2k-1 (k=1024). Konwersji między dwoma notacjami dokonuje kompilator.

Ćwiczenie 10. Napisać program, który zapisze w rejestrze `R20` liczbę `5`, a następnie będzie go dekrementował w nieskończoność.

Podawanie adresu skoku w przypadku bardziej złożonych programów jest niepraktyczne. Zamiast tego można wykorzystać tzw. etykiety. Pozwalają one na nadanie nazwy określonemu miejscu w programie w celu odwołania się do niej w instrukcji. Przykład wykorzystania etykiety pokazano poniżej. (Uwaga: Etykieta jest instrukcją kompilatora a nie mikroprocesora. Kompilator zamienia etykiety na odpowiednie liczby-adresy.)

```
      nop
Loop: nop
      rjmp Loop
      nop
      nop
```

Ćwiczenie 11. W kodzie z poprzedniego ćwiczenia (Ćw. 10) użyć etykiety zamiast adresu skoku.

7. Skoki warunkowe

Instrukcja `rjmp` jest skokiem bezwarunkowy tzn., że skok jest wykonywany zawsze, gdy wykonywana jest instrukcja. Nie pozwala to na tworzenie np. pętli o skończonej liczbie iteracji.

Mikrokontrolery AVR posiadają dwa zestawy instrukcji skoków warunkowych. Jedna z nich składa się z instrukcji, w których wykonanie skoku zależy od stanu rejestru statusu (`SREG`). Są to instrukcje zależne od jednego określonego bitu (np. `brcs`, *Branch if Carry Set*) lub od więcej niż jednego bitu (np. `brbs`, *Branch if Status Flag Set*, czyli skocz, jeżeli jakikolwiek bit z `SREG` jest ustawiony). Podobnie jak w przypadku instrukcji `rjmp`, skoki `brxx` są skokami, w których argumentem jest wartość przesunięcia względem aktualnego stanu licznika programu (`PC`). Należy zwrócić uwagę, że w przypadku skoków warunkowych `brxx` zakres tego przesunięcia jest znacznie mniejszy i wynosi -64..+63.

Ćwiczenie 12. Zmodyfikować program z poprzedniego ćwiczenia tak, aby wykonywał pętlę do momentu osiągnięcia przez rejestr `R20` wartości 0.
Uwaga: należy skorzystać z faktu, że instrukcja `dec` modyfikuje `SREG`.

Ćwiczenie 13.

a) W celu usprawnienia debugowania wstawić program z poprzedniego ćwiczenia do pętli nieskończonej.
b) Zmierzyć liczbę cykli potrzebną na wykonanie kodu zawartego w pętli nieskończonej, tj.:
    - przekrokować program do instrukcji inicjalizacji licznika pętli
    - skasować *cycle counter*
    - przekrokować program do pierwszej instrukcji po pętli wewnętrznej
    - odczytać licznik cykli
c) Na podstawie kodu programu oraz specyfikacji instrukcji podać wzór na liczbę cykli potrzebną na wykonanie kodu zawartego w pętli nieskończonej, np. *Cycles=(R20*12)+1*. Wzór zapisać w kodzie programu po komentarzu. Zwrócić uwagę na to ile cykli jest potrzebnych na wykonanie instrukcji `brxx` ?
d) Zmodyfikować program tak, aby wzór wyglądał następująco: *Cycles=(R20*5)*. Użyć instrukcji `nop`. Sprawdzić dla licznika równego 10. Zamiast krokowania przez 10 pętli użyć *breakpointa*.
e) Zmodyfikować poprzedni program tak, aby wzór wyglądał następująco: *Cycles=(R20*5)+5*

Ćwiczenie 14. Napisać program, w którym będzie jedna i tylko jedna instrukcja `nop` i który wykona tę instrukcję 500 razy. Użyć zagnieżdżonych pętli warunkowych.

Ćwiczenie 15. Napisać program, którego pojedyncze wykonanie będzie trwać 10 000 cykli. Użyć zagnieżdżania pętli. Ograniczyć liczbę `nop`-ów do minimum.

Ćwiczenie 16. Ile milisekund zajmie wykonanie powyższego programu? Zmodyfikować częstotliwość symulatora procesora na 8 MHz (W okienku z rejestrami procesora wpisać odpowiednią wartość w pole *Frequency*). Zmodyfikować powyższy program tak, aby jego wykonanie zajęło jedną milisekundę.

Ćwiczenie 17. Rozbudować poprzedni program tak, aby generował opóźnienie dane wzorem: *R22*1ms*, czyli określone przez zawartość rejestru `R22`. Użyć dodatkowego zagnieżdżenia. Dopuszcza się błąd opóźnienia nie większy niż 0,1%.

Ćwiczenie 18. Zrealizować funkcjonalność poprzedniego programu, ale z podwójnym zagnieżdżeniem.

- Zmniejszyć zagnieżdżenie używając licznika 16-bitowego zrealizowanego z dwóch rejestrów
- Licznik powinien służyć do realizacji opóźnienia 1 ms (dopuszcza się 8001 taktów z pierwszym `ldi`)
- Licznik powinien być inkrementowany, czyli warunkiem wyjścia z pętli powinno być przepełnienie starszego bajtu licznika
- Do inkrementacji licznika należy skorzystać z odpowiednich instrukcji dodawania oraz bitu przeniesienia (`C`) (patrz ćwiczenia 6 i 7)
- Do przeliczeń użyć kalkulatora z widokiem programisty
- Liczby w formacie heksadecymalnym zapisuje się z prefiksem `$` (np. `$F4`).

Ćwiczenie 19. Zmodyfikować poprzedni program tak, aby wykorzystywał instrukcję odejmowania pracującą na liczbach 16-bitowych (parach rejestrów, *Subtract Immediate from Word*).

8. Podprogramy

Podprogramy w języku Assembler funkcjonalnością odpowiadają funkcjom w językach wyższych poziomów. Poniższy przykład ilustruje użycie podprogramów.

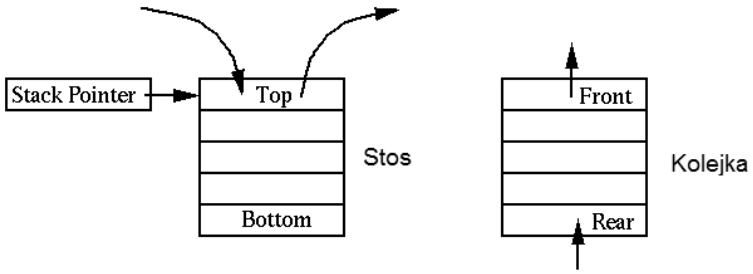
Podstawową wersją instrukcji wywołania jest `call`. Opkod takiej instrukcji zawiera bezwzględny adres procedury obsługi przerwania. Niektóre wersje mikrokontrolerów nie posiadają tej instrukcji, a zamiast niej mają instrukcję `rcall`. Tak jest w przypadku mikrokontrolera używanego na zajęciach. W odróżnieniu od `call`, opkod `rcall` zawiera, zamiast adresu skoku, wartość przesunięcia względem obecnego stanu licznika programu (PC). Analogicznie do instrukcji skoku bezwarunkowego (`rjmp`).

Różnica między zwykłą instrukcją skoku (`rjmp`) a instrukcją wywołania podprogramu (`rcall`) polega na tym, że ta ostatnia, oprócz modyfikacji licznika programu, zapamiętuje również adres powrotu, czyli adres następnej instrukcji (w poniższym przykładzie jest to `rjmp`). Adres ten jest używany przez instrukcję powrotu z podprogramu (`ret`). Instrukcja `ret` powoduje powrót z podprogramu, wpisując zapamiętany przez `rcall` adres powrotu do licznika programu (`PC`).

```
MainLoop:
rcall DelayNCycles ;
rjmp  MainLoop

DelayNCycles: ;zwykła etykieta
nop
nop
nop
ret           ;powrót do miejsca wywołania
```

Jak już wspomniano, wywołanie oraz powrót z podprogramów wiążą się odpowiednim z zapamiętaniem oraz odczytem adresu. Operacje te wykonywane są z użyciem struktury danych nazywanej stosem. W odróżnieniu od kolejki, w której pierwszy element, który wchodzi jest pierwszym elementem który wychodzi (*first in first out, FIFO*), w stosie jako pierwszy wychodzi element, który był odłożony jako ostatni (*last in first out, LIFO*). Zaleta stosowania stosu do zapamiętywania adresów powrotu podprogramów jest widoczna w momencie, kiedy istnieje potrzeba użycia wywołań zagnieżdżonych, czyli np. jeżeli w podprogramie znajduje się wywołanie innego podprogramu. Użycie stosu zapewnia stosunkowo prosty powrót do miejsca ostatniego wywołania podprogramu.



W mikrokontrolerach AVR, podobnie jak w większości mikrokontrolerów, stos znajduje się w pamięci danych oraz „rośnie w górę" od adresów starszych do młodszych. Aktualny adres wierzchołka stosu (*top*) przechowywany jest w rejestrze nazywanym wskaźnikiem stosu (`SP`, ang. *Stack Pointer*). Ponieważ szerokość słowa pamięci danych wynosi jeden bajt, a do zapamiętania adresu potrzebne są dwa bajty, dlatego `rcall` zmniejsza wskaźnik stosu o dwa, a `ret` zwiększa wskaźnik stosu o dwa. Stos umieszczany jest zwykle na końcu pamięci (najstarsze adresy) danych. Po resecie mikrokontrolera AVR, wskaźnik stosu ustawiony jest na ostatni bajt pamięci danych.

Ćwiczenie 20. Skompilować i uruchomić program z przykładu.

a) Przekrokować program bez wchodzenia do podprogramu (*F10*) oraz z wejściem do podprogramu (*F11*). Zwrócić uwagę na działanie wskaźnika stosu (`SP`). Najlepiej ustawić format wyświetlania na dziesiętny.
b) Obliczyć czas potrzebny na wywołanie i wykonanie podprogramu, a następnie zweryfikować wartość przy pomocy symulatora.

Ćwiczenie 21. Dodać pojedyncze zagnieżdżenie (dowolny podprogram). Zaobserwować jak pracuje wskaźnik stosu. Zaobserwować zawartość stosu (Okienko *Memory*, pamięć *data IRAM*, koniec pamięci). Sprawdzić czy adresy odkładane na stos zgadzają się z adresami powrotów.

Ćwiczenie 22. Wstawić pętlę opóźniająca z programu z ćwiczenia 19 do podprogramu `DelayInMs`, a następnie doprowadzić program do poprawnego działania. Uwaga: Inicjalizacja licznika pętli opóźniającej powinna znaleźć się tuż przed wywołaniem podprogramu.

Ćwiczenie 23. Z podprogramu `DelayInMs` wydzielić podprogram `DelayOneMs`, a następnie doprowadzić program do poprawnego działania.

W nietrywialnych programach istnieje od kilku do nawet kilkudziesięciu poziomów zagnieżdżeń, a poszczególne podprogramy korzystają z więcej niż dwóch rejestrów. W takich programach nie ma możliwości, aby każdy podprogram mógł posiadać na wyłączność jakąś grupę rejestrów. Z drugiej strony podprogramy nie mogą pracować na jednej wspólnej grupie rejestrów (patrz ćwiczenie poniżej).

Ćwiczenie 24. W programie z ćwiczenia 23 ograniczyć do dwóch ilość rejestrów używanych przez podprogramy `DelayInMs` oraz `DelayOneMs`. Sprawdzić działanie programu.

Jak widać podprogram wywoływany (`DelayOneMs`) modyfikuje zawartość zmiennej podprogramu wywołującego (`DelayInMs`, licznik pętli) czyli innymi słowy gubi jego stan. Ochronę stanu podprogramu wywołującego można zapewnić używając pamięci danych mikrokontrolera.

Ćwiczenie 25. Doprowadzić program z ćwiczenia 24 do poprawnego działania bez zwiększania liczby używanych rejestrów. W tym celu, w podprogramie wywoływanym (`DelayOneMs`) zapewnić ochronę odpowiedniego rejestru, używając instrukcji `lds` i `sts` oraz komórki pamięci danych o adresie `0x60`. Obserwować zawartość komórki `0x60` podczas krokowania podprogramu `DelayInMs`.

Rozwiązanie z ćwiczenia nie jest do końca bezpieczne, ponieważ zapewnia ochronę tylko jednego konkretnego rejestru. W tym przypadku możemy sobie na to pozwolić, ponieważ wiemy, jak działa program wywołujący (wiadomo jakiego rejestru używa `DelayInMs`). Zwykle nie jest z góry wiadomo jakich rejestrów będzie używać program wywołujący (lub programy wywołujące w przypadku wielokrotnego zagnieżdżenia). Z tego powodu każdy podprogram powinien zapewnić ochronę każdego rejestru, który modyfikuje.

Ćwiczenie 26. Zapewnić pełną ochronę rejestrów w podprogramie `DelayOneMs` używając komórek pamięci danych o adresach `0x60` i `0x61`.

Implementacja ochrony rejestrów z użyciem określonych na stałe komórek pamięci wymaga uzgodnienia wspomnianych adresów między wszystkimi podprogramami, które mogą być potencjalnie użyte. Należy zauważyć, że pomijając trywialne przypadki, nie można z góry określić jakie podprogramy oraz w jakiej kolejności zostaną użyte w danym programie. Najprostszym rozwiązaniem (w sensie idei) byłoby przydzielenie każdemu podprogramowi określonych komórek pamięci danych. Byłoby to rozwiązanie skomplikowane w implementacji oraz mało efektywne. Załóżmy, że mamy do dyspozycji 24 podprogramy od `A` do `Z`. Jest mało prawdopodobne, że wszystkie będą aktywne jednocześnie, tj., że `A` wywoła `B`, `B` wywoła `C`, … `Y` wywoła `Z`, a co za tym idzie, że będziemy korzystać z całej pamięci zarezerwowanej dla ochrony rejestrów.

Rozwiązaniem powyższej kwestii, które zapewnia jednocześnie uproszczenie implementacji jak i zwiększenie efektywności (wykorzystania pamięci) jest wykorzystanie stosu (zwykle tego samego, którego używają instrukcje do wywołania i powrotu z podprogramów).

Ćwiczenie 27a. Zapewnić pełną ochronę rejestrów w podprogramie `DelayOneMs` używając stosu. W tym celu znaleźć i zastosować instrukcje do wkładania i zdejmowania ze stosu rejestrów. (Uwaga na kolejność użycia instrukcji.)

Podobnie jak przedstawione wcześniej rozwiązanie, ochrona rejestrów z użyciem stosu wymaga użycia pamięci danych. Zasadnicza różnica polega na tym, że ilość wykorzystanej pamięci zależy głównie od maksymalnej liczby jednocześnie aktywnych podprogramów (maksymalna liczna zagnieżdżeń). Jest ona zwykle znacznie mniejsza niż liczba wszystkich podprogramów użytych w programie. Ponadto użycie stosu zwalnia nas z konieczności alokacji określonych komórek pamięci dla określonego podprogramu. W przypadku użycia stosu alokacja ta przebiega de facto automatycznie.

Ćwiczenie 27b.

W podprogramie `DelayInMs`:

- zapewnić pełną ochronę rejestrów używając stosu.

- zmodyfikować kod tak, aby przekazywanie liczby milisekund do podprogramu odbywało się przez rejestry `R16` (część młodsza), `R17` (część starsza).

9. Makra

Makro jest listą instrukcji, które są wstawiane przez kompilator w miejscu wywołania. Początek i koniec makra definiuje się dyrektywami `.macro` oraz `.endmacro`. Makro posiada nazwę oraz może przyjmować do 10 argumentów. Kolejne argumenty, oddzielone przecinkami, podaje się przy wywołaniu makra, natomiast w deklaracji makra odnosi się do nich kolejno `@0` – `@9`. Przykład makra, które zamienia wartości między rejestrami pokazano poniżej.

Definicja:

```
.macro XCHANGE
push @0
push @1
pop  @0
pop  @1
.endmacro
```

Użycie:

```
XCHANGE R0,R1
```

Ćwiczenie cw_m1. Napisać makro `LOAD_CONST`, które będzie zapisywać do podanych rejestrów stałą liczbową. Np. `LOAD_CONST R17,R16,1234`. Wystarczy jeśli makro będzie działać na rejestrach `R16` i wyższych. Do wyodrębniania starszego i młodszego bajtu z liczby użyć funkcji `low` i `high` assemblera (np. `low(1234)`).

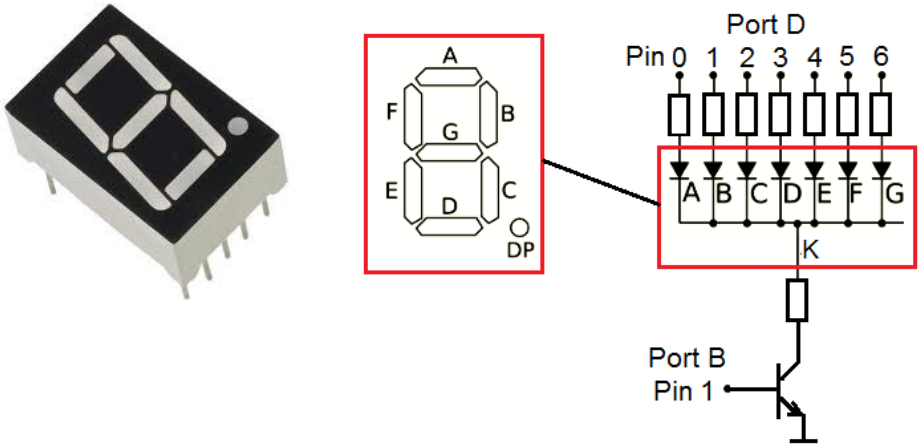
Ćwiczenie cw_m2. Użyć stworzonego makra w programie z ćwiczenia 27b (ładowanie wartości początkowych liczników pętli).

10. GPIO

Podobnie jak inne mikrokontrolery, mikrokontrolery AVR posiadają porty. W przypadku rodziny AVR są to porty 8-bitowe. Praca z danym portem odbywa się za pośrednictwem grupy rejestrów. Podstawowe z nich to rejestr do ustawiania kierunku pinów (`0`-wejście), rejestr służący do ustawiania stanu pinów oraz rejestr służący do odczytu stanu pinów. Do ustawiania stanu całych rejestrów służą instrukcje `OUT` i `IN`. Jako jeden z argumentów wspomnianych instrukcji należy podać adres rejestru. Jest to dosyć niewygodne, dlatego zamiast adresów można użyć etykiet: `DDRx` – rejestr kierunku, `PORTx` – rejestr ustawiania stanu pinu, `PINx` – rejestr odczytu stanu pinu, `x` nazwa portu (`A`, `B`, itd).

Ćwiczenie 28. Napisać program, który będzie na przemian cyklicznie ustawiał `0` i `1` na pinach 1-4 portu `B`. Nie zapomnieć o ustawieniu kierunku odpowiednich pinów. Wynik działania programu sprawdzić na symulatorze.

Ćwiczenie 29. Na rysunku pokazano sposób podłączenia jednej z cyfr wyświetlacza podłączonego do zestawu uruchomieniowego. Napisać program, który będzie na przemian co 250 ms wyświetlał „0” i „1”. Sposób wgrywania programu do mikrokontrolera podano w Appendixie B.



Ćwiczenie 30a. Zmodyfikować program z poprzedniego ćwiczenia tak, żeby cyklicznie wyświetlał „0” po kolei na wszystkich cyfrach zaczynając od najmłodszej (w każdym momencie powinna świecić się tylko jedna cyfra). Sposób podłączenia poszczególnych cyfr wyświetlacza do pinów portu `B` podano poniżej. Poszczególne segmenty wszystkich cyfr wyświetlacza są podłączone do portu `D`, tak jak na powyższym rysunku.

Cyfra-Pin
0 – 1
1 – 2
2 – 3
3 – 4

Ćwiczenie 30b. Używając dyrektywy `.equ` nadać etykiety portom `B` i `D` (`PORTB – Digits_P`, `PORTD – Segments_P`).

Ćwiczenie 31. W programie z poprzedniego ćwiczenia zmodyfikować podprogram `DelayInMs` tak, aby można było generować opóźnienie większe niż 255 ms. W tym celu zaimplementować licznik pętli podobnie jak w podprogramie `DelayOneMs`. Ponadto dodać ochronę rejestrów. Sprawdzić działanie dla przełączania cyfr co sekundę.

Ćwiczenie 32. Dobrać opóźnienie tak, aby częstotliwość odświeżania CAŁEGO wyświetlacza wynosiła 50 Hz.

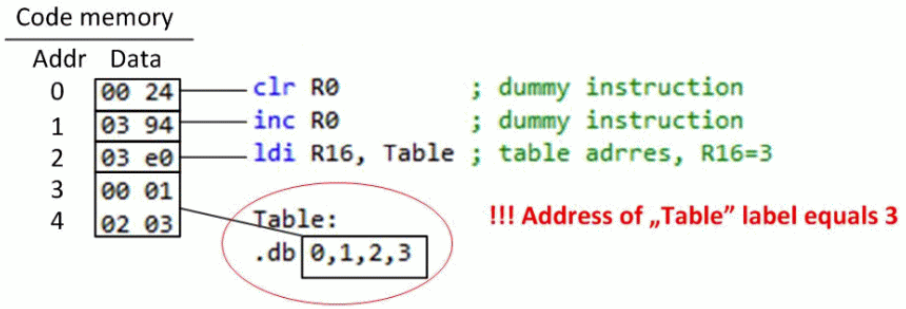
Ćwiczenie 33. Zmodyfikować program z poprzedniego ćwiczenia tak, aby na wyświetlaczu wyświetlało się „0123”.

Ćwiczenie 34. Zmodyfikować program z poprzedniego ćwiczenia tak, aby stan cyfr (kod siedmiosegmentowy) był przechowywany w rejestrach `R2` do `R5` (`R2` – najmłodsza cyfra). Rejestry należy zainicjalizować przed pętlą główną.

Ćwiczenie 35. Używając dyrektywy `.def` nadać rejestrom z poprzedniego ćwiczenia nazwy `Digit_0`, … `Digit_3`, a następnie posłużyć się nimi w kodzie. Informacje na temat dyrektyw assemblera można znaleźć w dokumentacji assemblera AVR (link na stronie przedmiotu).

11. Tablice stałych

Pamięć kodu może przechowywać oprócz instrukcji programu dowolne liczby, w tym min. tablice liczb (stałych). Definiuje się je przy użyciu dyrektyw `.db` (deklaracja bajtu) lub `.dw` (deklaracja słowa) . Poniżej pokazano przykład deklaracji tablicy.



Do odczytu danych z pamięci kodu służy instrukcja `LPM Rd,Z`, która ładuje do określonego rejestru bajt o adresie znajdującym się w rejestrze `Z`. Rejestr `Z` jest 16-bitowym rejestrem składającym się z rejestrów `R31` i `R30` (`R31:R30` - *MSB*:*LSB*). Rejestr `Z` adresuje pojedyncze bajty a nie słowa. Czyli, np. w celu odczytania starszego bajtów instrukcji `inc` powyższego programu, do rejestru `Z` należało by wpisać wartość `2` zamiast `1`. Etykiety (np. `Table` w przykładowym programie) są adresowane tak jak instrukcje, czyli słowami 16-bitowymi. Z tego powodu, po lub w trakcie inicjalizacji rejestru `Z` adresem etykiety początku tablicy stałych (`Table`), należy wartość adresu pomnożyć przez dwa. Można do tego użyć dyrektywy preprocesora (<<). Ilustruje to poniższy przykład. Użyto w nim również funkcji assemblera `low` i `high` służących do wyodrębniania odpowiednio młodszej i starszej części słowa 16-bitowego.

```
// Program odczytuje 4 bajty z tablicy stałych zdefiniowanej w pamięci kodu do rejestrów
R20..R23

ldi R30, low(Table<<1)  // inicjalizacja rejestru Z
ldi R31, high(Table<<1)

lpm R20, Z      // odczyt pierwszej stałej z tablicy Table

adiw R30:R31,1 // inkrementacja Z
lpm R21, Z      // odczyt drugiej stałej

adiw R30:R31,1 // inkrementacja Z
lpm R22, Z      // odczyt trzeciej stałej

adiw R30:R31,1 // inkrementacja Z
lpm R23, Z      // odczyt czwartej stałej

nop

Table: .db 0x57, 0x58, 0x59, 0x5A // UWAGA: liczba bajtów zdeklarowanych
                                  // w pamięci kodu musi być parzysta
```

Ćwiczenie 36. Przekrokować powyższy kod. Obserwować zawartość odpowiednich rejestrów. Znaleźć w pamięci kodu stałe zdefiniowane w tablicy.

Ćwiczenie 37. Napisać i sprawdzić podprogram, który będzie konwertował przy pomocy tablicy stałych liczby od 0-9 na ich wartość podniesioną do kwadratu. Wartość powinna być przekazywana do i z podprogramu za pomocą rejestru `R16`.

Ćwiczenie 38. Zmodyfikować podprogram z poprzedniego ćwiczenia tak, aby zamieniał liczby 0-9 na ich kod siedmiosegmentowy. Program powinien nazywać się `DigitTo7segCode`. Program powinien zapewniać ochronę rejestrów.

Ćwiczenie 39a. Zmodyfikować program z ćwiczenia 35 tak, aby w zmiennych `Digit_X` zamiast kodu siedmiosegmentowego znajdowały się liczby (cyfry), które mają być wyświetlane (czyli np. jeżeli na wyświetlaczu ma się wyświetlać „9753”, to w zmiennych powinny być liczby 3, 5, 7, 9). Do przekodowywania liczb na kod siedmiosegmentowy użyć podprogramu `DigitTo7segCode`. Przeprowadzić 3 testy: „3210”, „7654”, „9876”.

Ćwiczenie 39b. Pętla główna składa się z czterech grup instrukcji odpowiadających odświeżeniu poszczególnych cyfr wyświetlacza. Zastąpić każdą grupę wywołaniem makra, którego argument stanowi numer wyświetlacza.

```
MainLoop:
       SET_DIGIT 0
       SET_DIGIT 1
       SET_DIGIT 2
       SET_DIGIT 3
      rjmp MainLoop
```

12. Liczniki

Ćwiczenie 40. Dodać do pętli głównej fragment kodu, który w każdej iteracji będzie zwiększał o jeden zawartość rejestru odpowiadającego najmłodszej cyfrze wyświetlacza. W momencie osiągnięcia wartości dziesięć, rejestr powinien być resetowany (0,1,…,9,0,1,..9). Należy użyć następujących instrukcji: `inc`, `ldi`, `cp`, `brne`, `clr` oraz pamiętać, że skoki mogą być wykonywane także do przodu. Testy przeprowadzić dla częstotliwości odświeżania wyświetlacza (nie cyfry) równej 1 Hz.

Ćwiczenie 41. Zmodyfikować program z poprzedniego ćwiczenia tak, aby realizował licznik dekadowy, gdzie poszczególne dekady to rejestry odpowiadające cyfrom wyświetlacza. Innymi słowy cyfra 0 powinna się inkrementować (modulo 10) w każdej iteracji pętli głównej. Cyfra 1 powinna się inkrementować w momencie przejścia cyfry 0 z 9 na 0 itd. Do implementacji opisanej funkcjonalności powinien wystarczyć zbiór instrukcji podany w poprzednim ćwiczeniu. Nie ma potrzeby stosować pętli zagnieżdżonych. W celu przetestowania wszystkich cyfr należy użyć różnych częstotliwości odświeżania wyświetlacza (max. 100 Hz).

Licznik dekadowy niezbyt efektywnie wykorzystuje pamięć. Do przechowywania liczby 9999 (maksymalna liczba możliwa do wyświetlenia na wyświetlaczu) wystarczy 14 bitów, czyli w praktyce 2 rejestry ośmiobitowe.

Ćwiczenie 42. Napisać podprogram `Divide`, który będzie dzielił przez siebie dwie liczby 16-bitowe i zwracał wartość całkowitą (`quotient`) oraz resztę z dzielenia (`remainder`). Czyli 1200/500 powinno zwrócić `quotient`=2 i `remainder`=200. Zapewnić ochronę rejestrów. Użyć poniższego algorytmu i definicji.

```
Quotient=0
while (Divident>=Divisor) { // for “>” use cp and cpc instructions
Divident = Divident - Divisor;
Quotient++; // use adiw instruction
}
Remainder = Divident
```

```
;*** Divide ***
; X/Y -> Quotient,Remainder
; Input/Output: R16-19, Internal R24-25

; inputs
.def XL=R16 ; divident
.def XH=R17

.def YL=R18 ; divisor
.def YH=R19

; outputs

.def RL=R16 ; remainder
.def RH=R17

.def QL=R18 ; quotient
.def QH=R19

; internal
.def QCtrL=R24
.def QCtrH=R25
```

Ćwiczenie 43. Napisać podprogram `NumberToDigits`, który zamieni jedną liczbę 0-9999 na cztery liczby odpowiadające jej cyfrom (np. 1357 -> 1,3,5,7). Należy użyć podprogramu `Divide`. Użyć poniższego algorytmu i definicji.

Algorytm:

1. Podziel liczbę wejściową przez 1000 w celu wyliczenia najstarszej cyfry (tysiące)
2. Podziel resztę z poprzedniego dzielenia przez 100 (setki)
3. Podziel resztę z poprzedniego dzielenia przez 10 (dziesiątki)
4. Reszta z poprzedniego dzielenia to jedności

```
;*** NumberToDigits ***
;input : Number: R16-17
;output: Digits: R16-19
;internals: X_R,Y_R,Q_R,R_R - see _Divide

; internals

.def Dig0=R22 ; Digits temps
.def Dig1=R23 ;
.def Dig2=R24 ;
.def Dig3=R25 ;
```

Ćwiczenie 44.

1. Zamienić licznik dekadowy na licznik binarny 16-bitowym (dwa rejestry).

```
.def PulseEdgeCtrL=R0
.def PulseEdgeCtrH=R1
```

2. W każdej iteracji pętli głównej licznik powinien być inkrementowany modulo 1000 (*((Ctr++)%1000*).
3. Po każdej inkrementacji licznika aktualizować wartość wyświetlacza (`Digit_0..3`) zawartością licznika przy użyciu podprogramu `NumberToDigits`.

W celu przetestowania wszystkich cyfr należy użyć różnych częstotliwości odświeżania wyświetlacza (max. 100 Hz).

13. Przerwania

Podobnie jak większość mikrokontrolerów, mikrokontrolery AVR wyposażone są w mechanizm przerwań. W momencie wystąpienia sygnału przerwania (reset mikrokontrolera, zmiana stanu pinu, przepełnienie licznika, itd.) procesor wykonuje skok do określonego miejsca w kodzie o określonym adresie, nazywanego też wektorem przerwań. W mikrokontrolerach AVR wektory przerwań przyporządkowane są na stałe do źródeł przerwań. Poniżej pokazano fragment tablicy z wektorami przerwań mikrokontrolera ATtiny2313A. Jak widać, wektory umieszczone są jeden za drugim, a co za tym idzie, wektor zajmuje jedno słowo pamięci kodu, czyli miejsce na jedną instrukcję. Z tego powodu, w praktyce, pod adresami przerwań nie umieszcza się procedur obsługi przerwań (*Interrupt Rservice Routines, ISR*), ale skoki do wspomnianych procedur (listing poniżej).

Vector No.	Program Address	Label	Interrupt Source
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset, and Watchdog Reset
2	0x0001	INT0	External Interrupt Request 0
3	0x0002	INT1	External Interrupt Request 1
4	0x0003	TIMER1 CAPT	Timer/Counter1 Capture Event
5	0x0004	TIMER1 COMPA	Timer/Counter1 Compare Match A
6	0x0005	TIMER1 OVF	Timer/Counter1 Overflow
7	0x0006	TIMER0 OVF	Timer/Counter0 Overflow

```
0x0000      rjmp RESET           ; Skok Reset
            . . .
0x0005      rjmp TIM1_OVF_ISR    ; Timer1 Overflow
0x0006      rjmp TIM0_OVF_ISR    ; Timer0 Overflow
0x0007      rjmp USART0_RXC_ISR ; USART0 RX Complete
            . . .
0x0014      rjmp PCINT2_ISR      ; PCINT2
```

W dotychczasowych programach nie używano przerwań. Kompilator umieszczał kod programu (pętli głównej i podprogramów) począwszy od zerowego adresu pamięci. Aby kompilator umieścił kod programu (pętli głównej i podprogramów) za wektorami przerwań, należy użyć dyrektyw `.cseg` i `.org`. Dyrektywa `.cseg` informuje program, że znajdujący się za nią kod powinien zostać umieszczony w pamięci kodu (w odróżnieniu od dyrektywy `.dseg` która nakazuje kompilatorowi umieszczenie kodu w pamięci danych). Dyrektywa `.org` informuje kompilator od jakiego adresu powinien umieszczać kod znajdujący się po dyrektywie.

Poniżej pokazano przykład programu z dwoma przerwaniami od resetu i od timera. Ponieważ używane są tylko dwa przerwania (od resetu i od timera), nie ma konieczności rezerwowania miejsca dla wszystkich wektorów przerwań. Co za tym idzie, kod programu (procedura obsługi przerwania oraz pętla główna) może znaleźć się zaraz za wektorem przerwania od timera. Definicje adresów wektorów przerwań (np. `OVF0addr`) zostały zdefiniowane w pliku *tn2313Adef.inc* (patrz *project\dependencies*).

```
.cseg                ; segment pamięci kodu programu

.org 0        rjmp _main        ; skok do programu głównego
.org OC1Aaddr rjmp _timer_isr   ; skok do obsługi przerwania timera

_timer_isr:          ; procedura obsługi przerwania timera
            inc R0 ; jakiś kod
            reti   ; powrót z procedury obsługi przerwania (reti zamiast ret)

_main:
main_loop:           ; pętla główna
            inc R16
            rjmp  main
```

Ćwiczenie 45. Wstawić na początek programu z poprzedniego ćwiczenia poniższy kod. Upewnić się, że po resecie nastąpi skok do programu głównego (w razie konieczności zmodyfikować etykiety i uruchomić program na sprzęcie).

```
.cseg                    ; segment pamięci kodu programu

.org 0         rjmp _main       ; skok po resecie (do programu głównego)
.org OC1Aaddr  rjmp _timer_isr ; skok do obsługi przerwania timera

_timer_isr:              ; procedura obsługi przerwania timera
            inc R0       ; jakiś kod
            reti         ; powrót z procedury obsługi przerwania (reti zamiast ret)
```

Ćwiczenie 46. Wstawić przed pętlą główną inicjalizację *Timera 1* oraz globalne odblokowanie przerwań (jedna instrukcja). Timer należy ustawić w tryb `CTC` z preskalerem 256. Następnie należy ustawić stałą przeładowania Timera tak, aby zgłaszał przerwanie co ok. 25600 cykli zegara. Następnie należy odblokować odpowiednie przerwanie Timera. Test przeprowadzić na symulatorze. Sprawdzić czy mikrokontroler wchodzi do przerwania co ok. 25600 cykli zegara.

Ćwiczenie 47. Przenieść inkrementację licznika `PulseEdgeCtr` oraz konwersję licznika na cyfry wyświetlacza z pętli głównej do procedury obsługi przerwania Timera (pamiętać o ochronie rejestrów). Wydłużyć okres Timera do 1 s. Sprawdzić działanie programu na zestawie uruchomieniowym.

Ćwiczenie 48. Zmodyfikować program tak, aby inkrementacja licznika `PulseEdgeCtr` dokonywała się w odpowiedzi na zmianę stanu linii `PB0` mikrokontrolera (na linię tą podawany jest sygnał z generatora częstotliwości). Dodać wektor przerwania, procedurę obsługi przerwania oraz inicjalizację przerwań zewnętrznych. Sprawdzić działanie programu na zestawie uruchomieniowym dla wszystkich częstotliwości generatora, zaczynając od 1 Hz. Zastanowić się, dlaczego program wiesza się dla wysokich częstotliwości. Co jeszcze, oprócz rejestrów `Rx`, powinno być chronione w procedurze obsługi przerwania? Poprawić i sprawdzić program (pamiętać o obu procedurach obsługi przerwania).

Ćwiczenie 49. Zmodyfikować program tak, aby realizował funkcjonalność miernika częstotliwość. (Należy m.in. dodać zerowanie licznika w odpowiednie miejsce.)