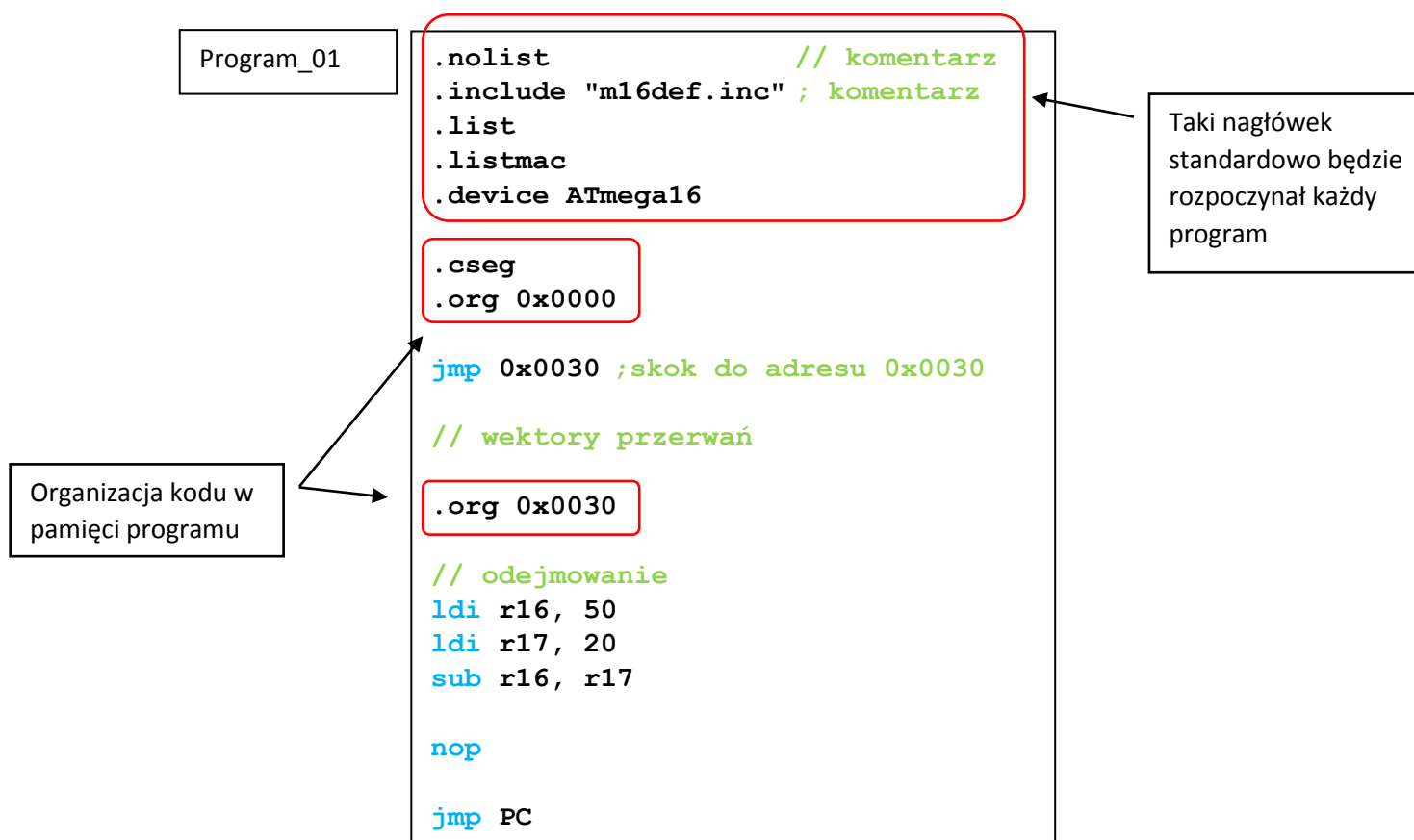


## 1. Zagadnienia

- Organizacja kodu programu w pamięci programu
- Analiza kodu instrukcji (na przykładzie instrukcji JMP)
- Przekroczenie zakresu liczb 8-bitowych
- Etykiety, inkrementacja zawartości rejestru, proste instrukcje arytmetyczne i logiczne

## 2. Organizacja kodu programu w pamięci programu

- Utwórz nowy projekt (**Project > Project Wizard**) według wskazówek podanych w instrukcji do Ćwiczenia 1.
- Wpisz kod programu: Odejmowanie liczb:  $50 - 20 =$



- Sprawdź działanie programu w pracy krokowej, poprawność skoku i poprawność otrzymanego wyniku odejmowania. Dobierz liczby tak, aby uzyskać wynik ujemny (np.  $100 - 250 =$ ). Obserwuj zachowanie znacznika C w rejestrze SREG.
- Dobierz liczby tak, aby wynik odejmowania był równy zero (np.  $10 - 10 =$ ). Obserwuj zachowanie znacznika Z.

W każdym programie na początku umieszczamy nagłówek złożony z kilku dyrektyw, które przygotowują środowisko programistyczne do pracy z wybranym typem procesora (ATmega16). Następnie określamy, że nasz kod docelowo powinien znaleźć się w pamięci programu (.cseg).

Organizujemy rozmieszczenie kodu korzystając z dyrektyw **.org**. Pierwsza wykonywana instrukcja skoku **JMP 0x0030** umieszczona jest w komórce o adresie 0x0000, osiągamy to za pomocą dyrektywy **.org 0x0000**. Wykonując instrukcję skoku **JMP 0x0030** procesor "skacze" do adresu 0x0030, zatem tam powinny zostać umieszczone kolejne instrukcje naszego programu, uzyskujemy to stosując dyrektywę **.org 0x0030**.

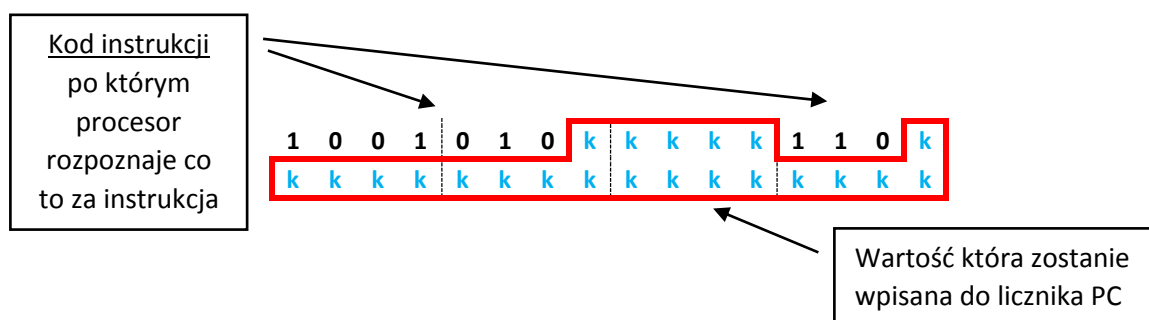
### 3. Analiza działania instrukcji JMP

Co to znaczy, że instrukcja **JMP** wykonuje skok (przekierowanie) do wskazanego adresu? W pomocy (**Help > Assembler Help**) możemy zapoznać się z opisem instrukcji i przeanalizować działanie skoku.

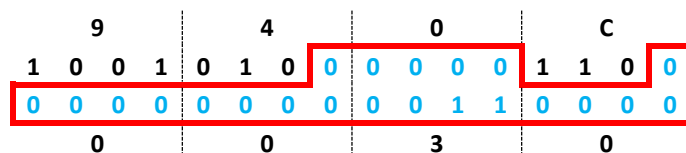
Działanie instrukcji JMP polega na bezpośrednim wpisie wartości (liczby k) do licznika (rejestru) Program Counter (PC). **Program Counter** jest najważniejszym rejestrem w każdym procesorze. Jego aktualna wartość wskazuje bezpośrednio adres w pamięci programu z którego zostanie pobrana kolejna instrukcja do wykonania.

#### Analiza struktury instrukcji JMP

Pamięć programu mikrokontrolera ATmega16 zorganizowana jest w 16-bitowe słowa (2 bajty). Jedna "komórka" pamięci to jedno słowo. Instrukcja JMP składa się z 32 bitów (4 bajty). W pamięci programu umieszczona zostanie w dwóch kolejnych komórkach. Instrukcję JMP można przedstawić w następujący sposób:



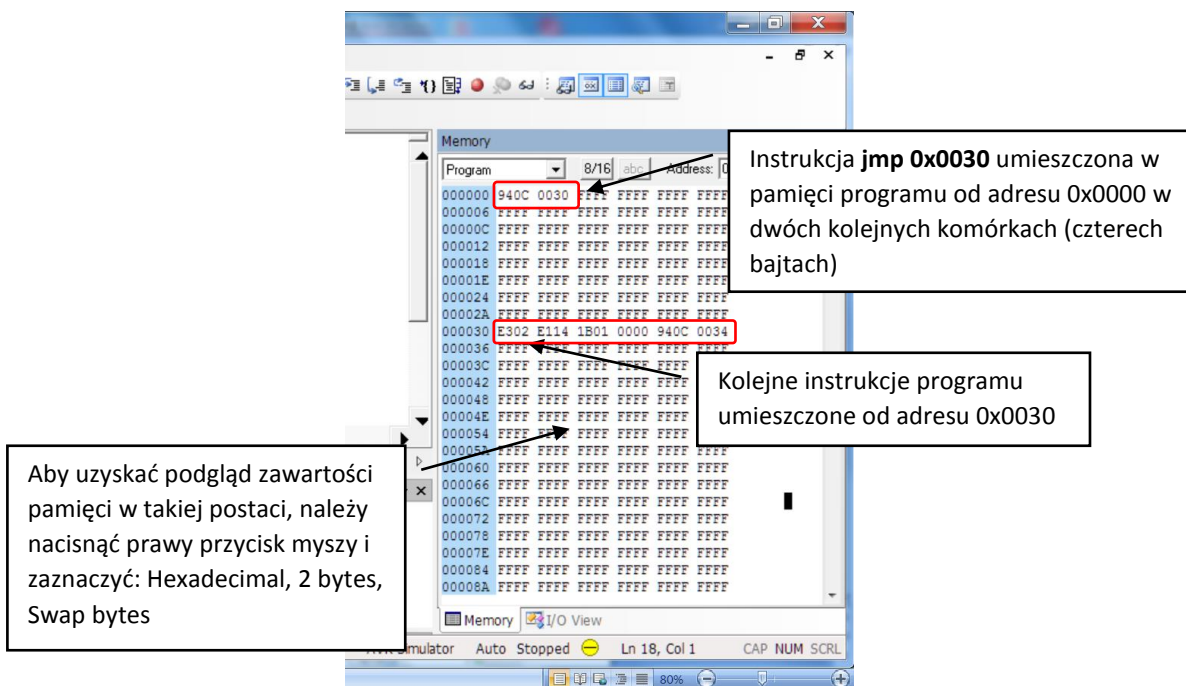
Cała instrukcja składa się z kodu instrukcji i argumentu (liczby k), który określa do jakiego adresu ma zostać wykonany skok. Przykładowo, jeżeli chcemy wykonać bezwarunkowy skok do adresu 0x0030 to musimy przyjąć wartość  $k = 30_{(HEX)}$ :



$$30_{(HEX)} = 00110000_{(2)}$$

Zatem instrukcja **JMP 0x0030** przedstawiona w kodzie maszynowym to **940C0030**. W instrukcji JMP do zapisu liczby k przeznaczono 22 bity - zatem przestrzeń adresowa do której możemy wykonać skok z wykorzystaniem instrukcji JMP jest całkiem spora, zawiera się między 0 a  $2^{22}$ , czyli 4MB.

- Po skompilowaniu (**F7**) programu i uruchomieniu trybu symulatora - otwórz podgląd pamięci (**View > Memory**), wybierz pamięć programu (Program). W pamięci programu możemy odnaleźć instrukcję **JMP 0x0030**.



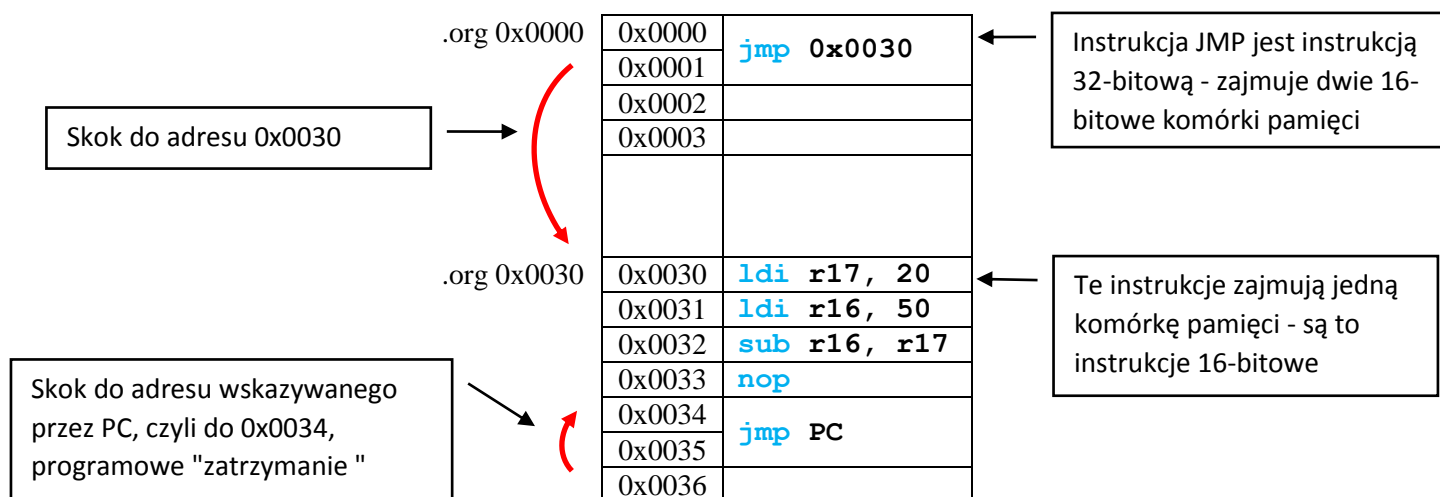
Aby uzyskać podgląd zawartości pamięci w takiej postaci, należy nacisnąć prawy przycisk myszy i zaznaczyć: Hexadecimal, 2 bytes, Swap bytes

Instrukcja **jmp 0x0030** umieszczona w pamięci programu od adresu 0x0000 w dwóch kolejnych komórkach (czterech bajtach)

Kolejne instrukcje programu umieszczone od adresu 0x0030

Oczywiście - programując w assemblerze - nie będziemy tak szczegółowo analizowali każdej instrukcji. Warto jednak mieć pewną wiedzę o tym, jak nasz program wygląda w kodzie maszynowym i gdzie jest lokowany w pamięci programu.

Rozmieszczenie instrukcji naszego programu można przedstawić obrazując mapę pamięci programu:



Oprócz instrukcji JMP, lista rozkazów procesorów AVR zawiera też inne instrukcje skoku: RJMP (*Relative Jump*) i IJMP (*Indirect Jump*). Ich działanie nieznacznie różni się od działania instrukcji JMP, ale wszystkie one służą do wykonania skoku bezwarunkowego.

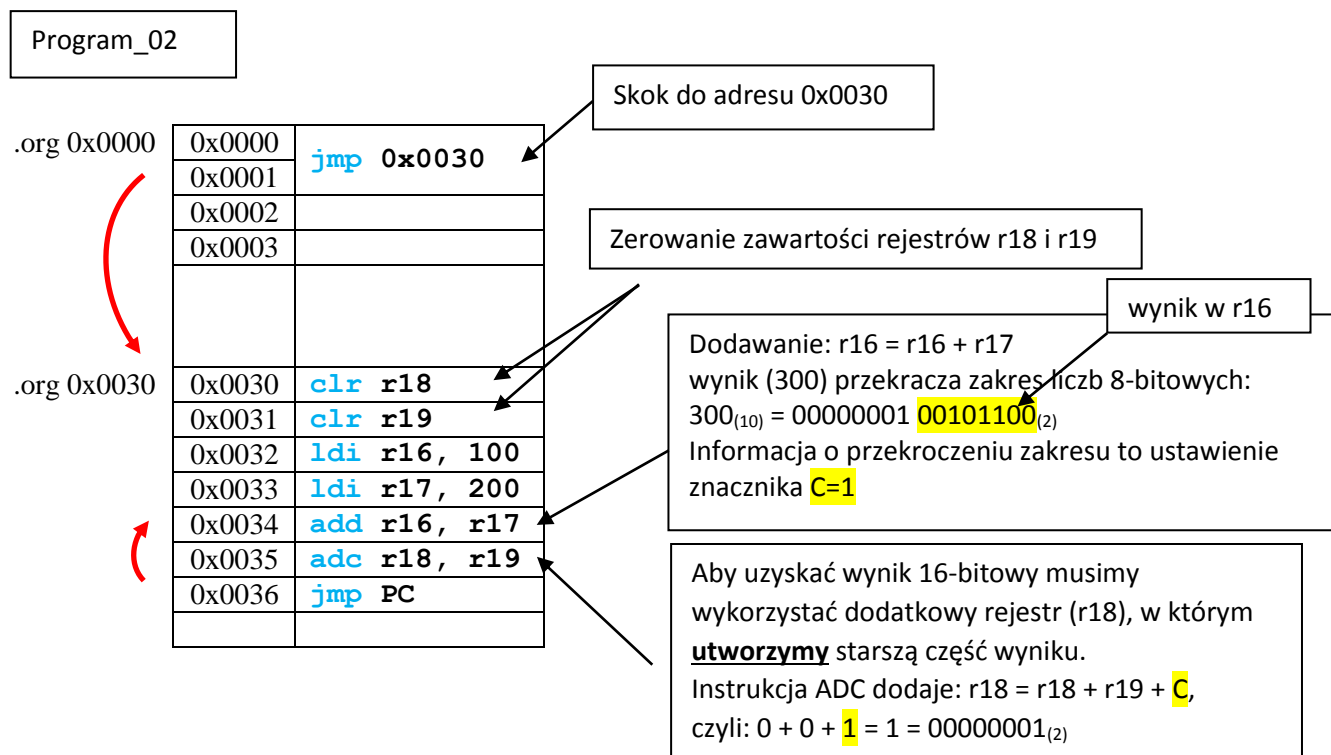
## 4. Dodawanie liczb 8-bitowych z przekroczeniem zakresu

Jeżeli chcemy wykonać dodawanie dwóch liczb 8-bitowych i spodziewamy się przekroczenia zakresu, to musimy wynik zapisać w postaci 16-bitowej. Przykładowo:

$$100_{(10)} + 200_{(10)} = 300_{(10)} = 00000001\ 00101100_{(2)}.$$

Do zapisania wyniku potrzebne są dwa bajty. Po wykonaniu instrukcji ADD r16, r17 młodszy bajt [00101100] jest już w rejestrze r16, starszy bajt [00000001] musimy samodzielnie utworzyć. Wykorzystamy do tego rejestr r18. Zatem wynik dodawania będzie zapisany w dwóch rejestrach: r18 - starszy bajt, r16 - młodszy bajt.

- Utwórz nowy program zawierający kod podany w poniższym przykładzie:



Instrukcja ADC działa podobnie jak instrukcja ADD, z tą różnicą, że do wyniku dodawana jest jeszcze zawartość znacznika C. Jeżeli po wykonaniu poprzedniej instrukcji (w tym przypadku jest to instrukcja ADD) zakres liczb 8-bitowych zostanie przekroczony - znacznik C zostanie ustawiony (C=1) i w kolejnej instrukcji (ADC) wartość 1 zostanie dodana do rejestru wynikowego r18. W ten prosty sposób "tworzymy" starszy bajt 16-bitowego wyniku. Jeżeli wynik dodawania liczb nie przekroczy zakresu 8-bitowego, to znacznik C=0, i w rejestrze r18 będziemy mieli nadal 0. Zatem 16-bitowy wynik też będzie prawidłowy.

Rejestr r19 (o wartości 0) wykorzystany jest tylko pomocniczo w operacji ADC (instrukcje działają na rejestrach, a nie na stałych). Aby mieć pewność, że w wykorzystywanych rejestrach nie ma "przypadkowych" wartości - na początku "czyścimy" je (instrukcja CLR). Końcowy wynik działania naszego programu:

r18	r16
00000001	00101100

## 5. Zwiększanie i zmniejszanie zawartości rejestrów

Bardzo często układach procesorowych zachodzi potrzeba zwiększenia (lub zmniejszenia) zawartości rejestru o wartość 1 (np. gdybyśmy chcieli napisać program odmierzający czas (zegar), należało by, np. co 1 sekundę zwiększyć licznik sekund - aż do 60, potem zwiększyć licznik minut, godzin... i tak dalej). Żeby zwiększyć/zmniejszyć zawartość rejestru możemy użyć poznanych już instrukcji dodawania (ADD)/odejmowania (SUB) - lub użyć specjalnych instrukcji inkrementacji/dekrementacji zawartości rejestru.

- Napisz program:

Program\_03

```
.nolist // komentarz
#include "m16def.inc"
.list
.listmac
.device ATmega16

.cseg
.org 0x0000
jmp START ;skok do etykiety START

// wektory przerwań

.org 0x0030
START:

//inkrementacja
ldi r16, 0
INC_REG:
inc r16
jmp INC_REG

LOOP_END:
jmp LOOP_END
```

W programie zastosowano etykiety. Etykieta to nazwa (ciąg liter lub cyfr - nazwa etykiety nie może zaczynać się cyfrą!) zakończona znakiem dwukropka (:). Wielkość liter nie ma znaczenia - START, start, STart - to ta sama etykieta. W programie zadeklarowano etykiety START, INC\_REG, LOOP\_END.

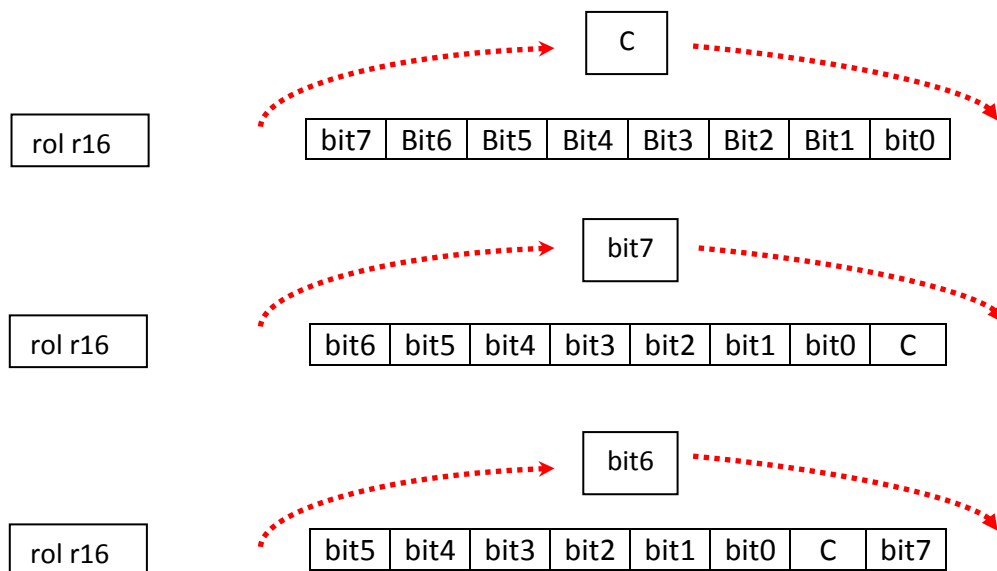
Dzięki zastosowaniu etykiet nie musimy podawać (i pamiętać) bezpośrednio wartości liczbowych adresów do których ma zostać wykonany skok (np. JMP 0x0030), środowisko AVR Studio automatycznie przypisze wartości liczbowe do odpowiednich etykiet.

- Uruchom program w pracy krokowej, obserwuj rejestr r16, znaczniki C i Z. Czy program osiągnie końcową pętlę LOOP\_END (jest to odpowiednik "zatrzymania" programu skokiem JMP PC).
- Zmień instrukcję INC na instrukcję DEC, wpisz dowolną wartość początkową do r16, obserwuj działanie programu.

## 6. Instrukcje przesuwania zawartości rejestru

Zawartość rejestru możemy przesuwać w lewo lub w prawo używając instrukcji ROL (*Rotate Left through Carry*) lub ROR (*Rotate Right through Carry*):

- Napisz program który w pętli będzie przesuwał zawartość rejestru r16 w lewo (ROL). Sprawdź działanie w pracy krokowej, obserwuj stan znacznika C.



itd ...

- Podobnie sprawdź działanie instrukcji ROR.

Oprócz instrukcji ROL i ROR zawartość rejestru możemy przesuwać za pomocą instrukcji LSR (*Logical Shift Right*) i LSL (*Logical Shift Left*):

- Napisz program który porówna działanie instrukcji ROL i LSL oraz ROR i LSR. Jaka jest różnica w działaniu tych instrukcji. Można zauważyć, że logiczne przesuwanie w lewo (LSL) o jedną pozycję - to mnożenie przez 2, a logiczne przesuwanie w prawo o jedną pozycję (LSR) - to dzielenie przez 2.

## 7. Instrukcja mnożenia MUL

W zestawie instrukcji mikrokontrolera AVR jest kilka instrukcji mnożących. W przypadku mnożenia liczb całkowitych stosowana jest instrukcja MUL.

- Napisz program mnożenia dwóch liczb 8-bitowych wykorzystując instrukcję MUL. Sprawdź działanie programu. Gdzie znajduje się wynik mnożenia? Czy wynik mnożenia jest liczbą 8- czy 16-bitową?

## 8. Instrukcje dzielenia

W zestawie instrukcji arytmetycznych mikrokontrolerów AVR nie ma instrukcji dzielenia, zatem gdy chcemy wykonać dzielenie musimy samodzielnie napisać odpowiedni fragment kodu programu.



## 9. Zadania do samodzielnej realizacji

### 1. Analiza instrukcji MOV

- W opisie działania instrukcji MOV (**Help > Assembler Help**) znajdź binarny kod instrukcji (16-bit Optcode).
- Spróbuj określić jaki kod będzie miała instrukcja MOV r16, r17 (zastanów się, jaką wartość przyjmują liczby r i d w kodzie instrukcji) ?
- Zastosuj instrukcję MOV r16 w swoim programie. W trybie symulatora otwórz podgląd pamięci programu (**View > Memory**), wybierz pamięć programu (**Program**). W pamięci programu znajdź (zapisaną szesnastkowo) instrukcję MOV r16, r17.
- Sprawdź, czy twoje przewidywania potwierdzą się dla innych rejestrów (np: MOV r1,r2, MOV r21,r31 itd...).
- Otwórz plik wynikowy \*.hex i również zlokalizuj instrukcje MOV.

### 2. Programy

- **Zadanie 1:** Napisz program, który wykona mnożenie zawartości rejestru r16 przez 10, wykorzystując instrukcje dodawania (proszę na razie nie stosować pętli i skoków warunkowych). Następnie napisz ten sam program mnożenia przez 10 wykorzystując instrukcje przesunięcia i dodawania. Aby nie przekroczyć zakresu liczb 8-bitowych, największa wartość którą można użyć podczas testów to 25. "Zmierz" w symulatorze liczbę taktów i czas działania obu programów- który program jest szybszy?
- **Zadanie 2:** Napisz program, który wykona dzielenie zawartości rejestru r16 przez 32, wykorzystując instrukcje przesunięcia. Zmierz w symulatorze liczbę taktów i czas działania programu?
- **Zadanie 3:** Napisz program odwracający kolejność bitów liczby zapisanej w rejestrze r16, wynik umieść w rejestrze r20 (wykorzystaj instrukcje przesunięcia).
- **Zadanie 4:** zadanie podane przez prowadzącego.

### 3. Zadania dodatkowe (dla chętnych)

- **Zadanie A:** Napisz program obliczający wartość funkcji:  $y = 32x + 20$ , wykorzystaj poznane sposoby mnożenia i dodawania. Zakładamy, że wartość x będzie liczbą 8-bitową a wynik będzie liczbą 16-bitową.
- **Zadanie B:** Napisz program dodawania dwóch liczb 16-bitowych (liczby zapisane w rejestrach r16, r17 oraz r18, r19). Zakładamy, że wynik będzie liczbą 32-bitową zapisaną w trzech kolejnych rejestrach.
- **Zadanie C:** zadanie podane przez prowadzącego.

## 10. Sprawozdanie

- W sprawozdaniu należy umieścić kody wykonanych programów z odpowiednim wyjaśnieniem działania zastosowanych dyrektyw i instrukcji.
- Na podstawie noty katalogowej ATmega16 wyjaśnić, dlaczego w prezentowanych programach wykonano skok do adresu 0x0030, omijając pewien obszar pamięci programu (*strona 45, Interrupts*).