Akademia Górniczo-
Hutmicza w Krakowie
Katedra Elektroniki
WIET

Technika mikroprocesorowa

Instrukcja 3

Stos i podprogramy

Autor: Paweł Russek
Tłumaczenie: Marcin Pietroń
http://www.fpga.agh.edu.pl/tm
ver. 04/04/15

1. Cel laboratorium

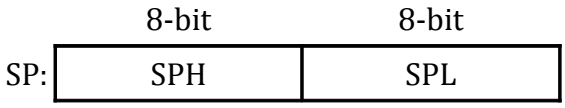
Głównym celem laboratorium jest zapoznanie studentów z instrukcjami procesora AVR, które umożliwiają kontrolowanie przebiegu programów oraz pozwalają na podział programu na moduły w postaci osobnych podprogramów. Praca podprogramów możliwa jest dzięki strukturze zwanej stosem.

2. Wiadomości wstępne

- Znajomość zagadnień laboratoriów z techniki mikroprocesorowej nr. 1 i 2 .

3. Stos, wskaźnik stosu oraz instrukcje stosowe

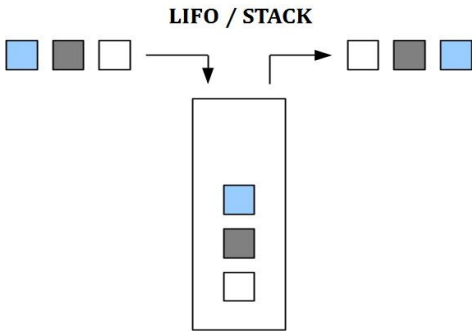
Stos to część pamięci RAM procesora służąca przechowywania danych tymczasowych. Procesor wykorzystuje przestrzeń danych w postaci stosu ponieważ dysponuje ograniczoną liczbą rejestrów wewnętrznych. Procesor AVR posiada specjalny rejestr, który służy do przechowywania adresu stosu (*SP – stack pointer register*). Szerokość rejestru wskaźnika stosu powinna być odpowiednio duża, aby możliwe było zaadresowanie całej przestrzeni pamięci RAM. Procesor AVR dysponuje dwoma rejestrami 8-bitowymi *SPL* i *SPH*. Rejestr *SP* jest realizowany jako połączenie dwóch wyżej wymienionych rejestrów 8-bitowych.



Wskaźnik stosu (zawartość rejestru *SP*) wskazuje na szczyt stosu. Program procesora może zapisać/wstawić dane na stos (poprzez instrukcję *push*) oraz odczytać/ściągnąć dane ze stosu (poprzez instrukcję *pop*). Dane mogą być wstawiane i ściągane jedynie ze szczytu stosu.

Stos to struktura typu Last-In-First_Out (LIFO). Według definicji struktura LIFO to lista, do której dane mogą dodawane lub wyciągane jedynie z jej jednego końca (rysunek poniżej).

Biały element został zapisany na stosie jako pierwszy, więc będzie odczytywany/ściągany ze stosu jako ostatni. Porządek elementów na wyjściu (po odczytaniu elementów ze stosu) jest odwróceniem wejścia (kolejności zapisu elementów na stos). W większości procesorów (np. rodzina procesorów x86) a także w procesorze AVR zawrtość rejestru wskaźnika stosu jest zmniejszana podczas zapisywania danych na stos.



3.1. Instrukcja zapisywania/wkładania danych na stos (push)

Instrukcja zapisuje zawartość rejestru *Rn* na stosie (pod adresem wskazywanym przez rejestr *SP*): (SP)←*Rn*. Wartość w rejestrze jest zmniejszana o wartość jeden po wykonaniu instrukcji.

push Rn ;0 ≤ n ≤ 31 (*SP*)←*Rn*, *SP*←*SP - 1*

3.2. Instrukcja ściągania/czytania danych ze stosu (pop)

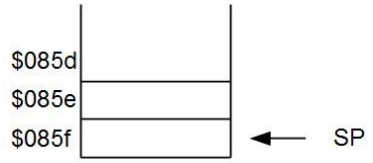
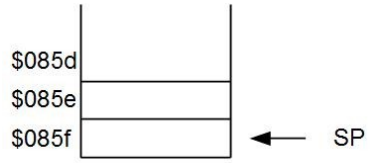
Instrukcja zapisuje rejestr *Rn* wartością ze stosu: *Rn*←(*SP*). Wartość w rejestrze jest zwiększana o wartość jeden przed wykonaniem instrukcji.

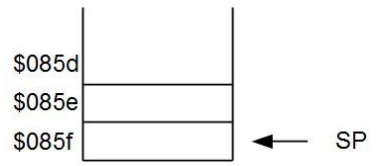
pop Rn ;0 ≤ n ≤ 31 *SP*←*SP + 1*, *Rn*←(*SP*),

Przykład.

; Step by step analysis of stack operations

```
ldi r16, $55
ldi r17, $aa
; Save r16 on the Stack
push r16
; Save r17 on the Stack
push r17
ldi r16,$00
ldi r17,$00
; Restore r17
pop r17
; Restore r16
pop r16
```

	r16	**r17**	**sp**	
ldi r16, $55	$55	$00	$085f	
ldi r17, $aa	$55	$aa	$085f	
push r16	$55	$aa	$085e	
push r17	$55	$aa	$085d	
ldi r16,$00	$00	$aa	$085d	
ldi r17,$00	$00	$00	$085d	
pop r17	$00	$aa	$085e	

pop r16	$55	$aa	$085f	

Uwaga

W większości procesorów (np. rodzina procesorów x86), a także w procesorze AVR zawartość rejestru wskaźnika stosu jest zmniejszana podczas zapisywania danych na stos

3.3. Inicjalizacja wskaźnika stosu

Po uruchomieniu procesora i po sygnale reset, rejestr *SP* zawiera wartość 0, która stanowi wskaźnik do adresu 0x00. Dlatego aby rozpocząć korzystanie ze stosu należy najpierw zainicjalizować rejestr *SP* odpowiednią wartością. Częstym rozwiązaniem jest inicjalizacja stosu adresem najwyższej możliwej lokacji w pamięci RAM. Ponieważ różne procesory rodziny AVR posiadają pamięci o różnej wielkości asembler oferuje zdefiniowany symbol *ramend*, reprezentującą adres ostatniej lokacji RAM.

Przykład

```
ldi r16, high(ramend)     ;load sph
out sph, r16
ldi r16, low(ramend)      ;load spl
out spl, r16
```

3.4. Funkcje high() oraz low()

Funkcje high() oraz low() zwracają odpowiednio starszy i młodszy bajt 16-bitowej wartości, na przykład: high(0x4455)=0x44 oraz low(0x4455)=0x55.

4. Podprogramy (instrukcja *call*)

Instrukcja *call* wykorzystywana jest do wywoływania podprogramów. Podprogramy (odpowiednik funkcji w językach wysokiego poziomu) używane są do realizacji często wykonywanych zadań w programie. Pozwala to uzyskać lepszą strukturę programu oraz oszczędza ilość wykorzystywanej pamięci. Aby możliwy był powrót z podprogramu do właściwej instrukcji programu, procesor AVR w trakcie wywołania instrukcji *call* zapisuje właściwy adres powrotu na stosie (zapisany jest rejestr *PC – program counter,* który wskazujące na kolejną instrukcję do wykonanie po powrocie z podprogramu). Po przejściu do podprogramu instrukcje pobierane są z innych lokacji pamięci. Każdy podprogram musi być zakończony instrukcją powrotu *ret* w celu ściągnięcia adresu powrotu ze stosu i wykonania skoku do kolejnej instrukcji programu.

4.1. Instrukcja wywoływania podprogramów (call)

Instrukcja wywołania podprogramu. Adres powrotu (adres instrukcji po instrukcji *call*) jest zapisywany na stos. Wskaźnik stosu jest zmniejszany o wartość dwa po wywołaniu *call*.

```
call k      ;0 ≤ k < 64K   PC ← k      STACK ← PC+2
            ;                          SP ← SP-2, (2 bytes, 16 bits)
```

4.2. Instrukcja powrotu z podprogramu (ret)

Powrót z podprogramu. Adres powrotu jest pobierany ze stosu. Wskaźnik stosu jest zwiększany o dwa przed odczytem adresu powrotu ze stosu:

```
ret         ; SP←SP + 2,  PC←  STACK
```

4.3. Stan rejestrów w trakcie wykonania podprogramów

Dobrą praktyką programowania w asemblerze jest zapisywanie wartości rejestrów, które będą wykorzystywane w podprogramie na stosie. Jednym ze sposobów jest użycie instrukcji *push* oraz *pop*. Zawartość rejestru wkładamy na stos poprzez instrukcję *push* na początku podprogramu.W celu odtworzenia stanu programu sprzed wywołania podprogramu, instrukcją *pop* ściągamy/odczytujemy dane z powrotem na końcu wykonywania podprogramu .

Przykład:

```
repeat:             ldi r16, $00
                    ldi r17, $ff
                    out portb, r16
                    call wait_subroutine
                    out portb, r17
                    call wait_subroutine
                    rjmp repeat
                    ...
wait_subroutine:    push r16            ; store r16
                    push r17            ; store r17
                    ldi r16,100         ; load new value onto r16
outter_loop:        ldi r17,10          ; load new value onto r16
inner_loop:         nop                 ; do something in the loop
                    dec r17
                    brne inner_loop     ; repeat it 10 times
                    dec r16
                    brne outter_loop    ; repeat it 100 times
                    pop r17             ;restore original r17 value
                    pop r16             ;restore original r16 value
                    ret
```

Ćwiczenie 3.1

Napisz program zliczający na diodach LED z opóźnieniem jednosekundowym. Opóźnienie zrealizuj za pomocą podprogramu. Po zakończeniu ćwiczenia zachowaj program jako "one_sec_subroutine.asm"

5. Dyrektywy asemblera

Program asemblera AVR zaczyna się podczas procesu jego startowania albo resetu od lokacji $0000 w przestrzeni pamięci programowej (PC=0x0000). Aby zadeklarować lokację pamięci, gdzie asembler ma umieścić program można użyć do tego przeznaczonej dyrektywy *ORG*.

Do pamięci programu można załadować nie tylko instrukcje procesora, ale także wartości stałe, które będą dostępne w trakcie wykonania programu. Używając tej możliwości można tworzyć tablice stałych w nieulotnej pamięci programu. Do definiowani zawartości pamięci programu, które nie są instrukcjami procesora służą dyrektywy .DB oraz .DW.

5.1. Dyrektywa *.ORG*

Dyrektywa *ORG* pozwala na ustawienie licznika lokacji w pamięci na wartość podaną jako parametr dyrektywy.

Składnia:

.ORG expression

Example:

```
.org 0
jmp start ;This instruction will be located at address 0x00 (16-bit words !!!)
.org 0x32
start:
nop ;This instruction will be located at address 0x32 (16-bit words !!!)
nop ;This instruction will be located at address 0x33 (16-bit words !!!)
```

Uwaga

Pamięć programu w procesorze AVR jest pamięcią w któej szerokość pojedynczej komórki pamięci jest 16-bitowa. W przypadku pamięci programu *org* wskazuje na adres w pamięci 16-bitowej.

6. Rezerwacja pamięci danych

Program asemblera ładowany jest do pamięci programu. Istnieje również możliwość zadeklarowania w assemblerze obszaru w pamięci danych. Aby rozdzielić w programie pamięć programową i pamięć danych używa się dyrektyw .CSEG oraz .DSEG. Dyrektywy .CSEG i .DSEG umieszcza się przed następującymi po nich segmentami programu. W praktyce w segmencie .DSEG używa się dyrektywy .BYTE.

6.1. Dyrektywa *BYTE*

Dyrektywa BYTE rezerwuje zadany obszar pamięć SRAM. Dyrektywa powinna być poprzedzona etykietą w celu możliwości odwoływania się do zarezerwowanej lokacji w pamięci. Dyrektywa wymaga jednego argumentu, który określa liczbę rezerwowanych bajtów. Dyrektywa może być używana jedynie w segmencie danych .DSEG. Zarezerwowana w ten sposób lokacja pamięci danych jest nie zainicjalizowana.

Składnia:

LABEL: .BYTE expression

Example:

```
.EQU    tab_size=16
.CSEG                           ;start program memory
.ORG 0
                rjmp prg        ;jump to the program
.DSEG                           ;start data memory
.ORG    0x60                    ;beginning of the SRAM memory in data space
        var1:   .BYTE 2         ; reserve 2 bytes to var1
table:  .BYTE tab_size          ; reserve 16 bytes
.CSEG                           ;program memory again
.ORG    0x32
prg:            nop             ; beginning of the program code
                ldi r16,0xaa
                sts table, r16  ; Store 0xaa at location 0x60 in SRAM
```

Uwaga

Pamięć danych w procesorze AVR jest pamięcią w któej szerokość pojedynczej komórki pamięci jest 8-bitowa. W przypadku pamięci programu *org* wskazuje na adres w pamięci 8-bitowej.

7. Instrukcje LDS i STS a pamięć danych

Instrukcje *lds* i *sts* służą do zapisu i odczytu danych z pamięci w rejestrach ogólnego przeznaczenia (instrukcja 2). Można wykorzystać te instrukcje do odczytu/zapisu lokacji pamięci zarezerwowanych poprzez dyrektywę *.BYTE*.

Przykład:

```
.CSEG                   ;start program memory
.ORG 0
        rjmp prg
.DSEG                   ;start data memory
```

```
.ORG 0x60                ;beginning of the SRAM memory in data space
var1: .BYTE 1            ; reserve 1 byte to var1
.CSEG                    ;program memory again
prg:  lds r16, var1      ; example of data memory direct addressing
      ;...
      sts var1, r18;     ;pass argument to subroutine using var1
      call subr
      lds r19, var1      ;receive results form subroutine in var1
      ;...
subr: lds r20, var1      ;read the subroutine parameter from var1
      ;...
      sts var1, r21      ; store the result of subroutine execution to var1
      ret
```

Przykład 3.2

Zmień program z ćwiczenia 3.1. Używając .DSEG, zarezerwuj pamięć SRAM. Użyj zarezerwowanej lokacji do przekazania parametru do podprogramu. Użyj tego parametru do zdefiniowania wartości opóźnienia. Zachowaj program jako ' one_sec_subroutine_param.asm'