Asembler A51

1. Symbole

Nazwy symboliczne

Symbol jest nazwą, która może być użyta do reprezentowania wartości stałej numerycznej, wyrażenia, ciągu znaków (tekstu), adresu lub nazwy rejestru. Nazwy symboliczne mogą składać się z liter (a - z, A - Z), cyfr (0 - 9), znaku podkreślenia (_) i znaku zapytania (?). Cyfra nie może wystąpić na początku nazwy.

Etykiety

Etykieta określa adres w pamięci programu lub danych, jest ona nazwą zakończoną dwukropkiem. Etykieta powinna wystąpić jako pierwsze pole tekstowe w linii (może być jednak poprzedzona spacjami lub znakami tabulacji).

Przykłady:

```
petla:  ...             ; etykieta określa adres w pamięci kodu (docelowe miejsce skoku)
        DJNZ    R7, petla

tekst:  DB      'nazwa' ; etykieta określa początkowy adres ciągu znaków w pamięci kodu
zmienna: DS     1       ; etykieta określa adres zmiennej
```

Wartości liczbowe

Wartości liczbowe mogą być podawane w formacie dziesiętnym, ósemkowym, szesnastkowym lub dwójkowym. Format liczby sygnalizuje odpowiedni przyrostek (brak przyrostka oznacza domyślny format dziesiętny).

Format	**Cyfry**	**Przyrostek**	**Przykłady**
dziesiętny	0 - 9	d, D	1234, 1234d, 1234D
ósemkowy	0 - 7	o, O, q, Q	123o, 123O, 123q, 123Q
szesnastkowy	0 - 9, a - f, A - F	h, H	123h, 123H, 0F123h
dwójkowy	0, 1	b, B	0111b, 11100B

Jeśli zapis liczby w formacie szesnastkowym zaczyna się od litery, to musi być ona poprzedzona cyfrą 0. Na przykład zapis AFh jest niepoprawny, prawidłowa postać tej liczby to 0AFh

Znaki

W sytuacji gdy potrzebna jest wartość liczbowa reprezentująca kod ASCII znaku, można użyć znaku objętego pojedynczym apostrofem ('). W kontekście wartości 16-bitowej można użyć dwóch kolejnych znaków objętych apostrofami.

Przykłady:

'A' - kod ASCII znaku A: 41h
'AB' - kody ASCII znaków A i B: 4142h

Ciągi znaków

W kontekście dyrektywy DB można użyć ciągu znaków objętych pojedynczymi apostrofami ('), co odpowiada ciągowi kodów ASCII kolejnych znaków. Aby w ciągu znaków umieścić pojedynczy apostrof, należy podać go dwukrotnie.

Przykłady:

```
text_1:   DB    'Podaj dane', 0     ; można łączyć w jednej linii ciąg znaków i wartości liczbowe
text_2:   DB    'It"s end'          ; podwójne użycie apostrofu pozwala wstawić go w środku tekstu
```

2. Definicje symboli

Aby przypisać symbolowi określoną wartość można użyć jednej z poniższych dyrektyw:

```
symbol      EQU     wyrażenie
symbol      SET     wyrażenie
```

Wartość przypisana symbolowi dyrektywą EQU nie może być później zmieniona, przy dyrektywie SET jest to możliwe. Wyrażenie może być prostą stałą ale może też zawierać inne symbole i operatory arytmetyczne. Wartość wyrażenia powinna być możliwa do określenia na etapie tłumaczenia programu.

Przykłady:

```
wartosc_1   EQU     10
wartosc_2   EQU     wartosc_1 + 3
wartosc_3   SET     5 * wartosc_2
```

Do przypisania symbolowi adresu z wybranej przestrzeni kodu lub danych służą podane niżej dyrektywy. Symbol raz zdefiniowany przy pomocy jednej z tych dyrektyw nie może być zdefiniowany powtórnie.

```
symbol      BIT     adres bitu
symbol      CODE    adres kodu
symbol      DATA    adres danej z wewnętrznej pamięci danych adresowanej bezpośrednio
symbol      IDATA   adres danej z wewnętrznej pamięci danych adresowanej pośrednio
symbol      XDATA   adres danej z zewnętrznej pamięci danych
```

Przykłady:

```
out_0       BIT     0           ; out_0 oznacza bit 0 w komórce pamięci 20h
out_1       BIT     out_0 + 1   ; out_1 jest bitem umieszczonym po out_0 czyli
                                  bitem 1 w komórce 20h
out_2       BIT     P1.5        ; out_2 oznacza bit 5 w porcie P1
start       CODE    0000h       ; start oznacza adres 0 w pamięci kodu
timer_0     CODE    start + 0Bh ; timer_0 oznacza adres 0Bh w pamięci kodu
wynik       DATA    30h         ; wynik jest zmienną w wewnętrznej pamięci danych
                                  pod adresem 30h
temp        IDATA   97h         ; temp oznacza zmienną w wewnętrznej pamięci
                                  danych pod adresem 97h
bufor       XDATA   5000h       ; bufor oznacza zmienną w pamięci zewnętrznej pod
                                  adresem 5000h
```

3. Używanie segmentów

Segment to blok danych w pamięci kodu lub danych. Kod lub dane umieszczone w segmencie o tej samej nazwie w różnych miejscach programu (lub nawet w różnych modułach) łączone są w jeden wspólny segment. Segmenty mogą być absolutne (bezwzględne) lub relokowalne. Segment absolutny jest umieszczany w pamięci pod zadanym adresem natomiast o umieszczeniu segmentu relokowalnego decyduje program łączący (linker).

Deklaracja segmentu

Aby możliwe było użycie segmentu powinien być on wcześniej zadeklarowany przy użyciu dyrektywy SEGMENT w następujący sposób:

```
nazwa segmentu    SEGMENT    klasa pamięci [relokacja]
```

nazwa segmentu - nazwa, która będzie później używana przy wyborze segmentu

klasa pamięci - określa w którym obszarze pamięci segment ma być umieszczony:

BIT	obszar adresowany bitowo:	20h - 2Fh
CODE	obszar kodu:	0000h - 0FFFFh
DATA	obszar danych IRAM adresowany bezpośrednio: 00h - 7Fh i SFR	
IDATA	obszar danych IRAM adresowany pośrednio: 00h - 0FFh	
XDATA	obszar danych XRAM	0000h - 0FFFFh

relokacja

BITADDRESSABLE	segment w obszarze adresowanym bitowo (tylko dla klasy DATA)
INBLOCK	segment w jednym bloku 2048 bajtów (tylko dla klasy CODE)
INPAGE	segment umieszczony w jednej stronie 256 bajtowej

Wybór segmentu

Wybór segmentu relokowalnego (zadeklarowanego wcześniej dyrektywą SEGMENT) wykonywany jest następująco:

```
RSEG    nazwa segmentu
```

Wybrany segment jest aktywny dopóki nie zostanie wybrany inny segment (dyrektywą CSEG lub RSEG).

Wybór segmentów absolutnych pod zadanym adresem:

```
BSEG    AT adres        segment typu BIT
CSEG    AT adres        segment typu CODE
DSEG    AT adres        segment typu DATA
ISEG    AT adres        segment typu IDATA
XSEG    AT adres        segment typu XDATA
```

Rezerwacja pamięci

Aby zarezerwować pewną liczbę bajtów pamięci można użyć następujących dyrektyw:

```
etykieta:  DBIT   wyrażenie      ; w obszarze bitowym
etykieta:  DS     wyrażenie      ; w obszarze danych (pamięć wewnętrzna lub zewnętrzna)
```

etykieta - określa adres zarezerwowanej pamięci (numer bitu lub adres bajtu)

wyrażenie - liczba rezerwowanych bitów lub bajtów (w najprostszym przypadku jest to stała)

Przykłady:

```
          RSEG   I_DATA      ; wybór segmentu danych w pamięci wewnętrznej
data_0:   DS     1           ; zarezerwowanie miejsca na dwie zmienne
data_1:   DS     1

          RSEG   BIT_DATA    ; wybór segmentu danych bitowych
bit_0:    DBIT   1           ; zarezerwowanie miejsca na dwie zmienne bitowe
bit_1:    DBIT   1
```

Inicjalizacja pamięci

Możliwe jest zainicjowanie pamięci (tylko w obszarze kodu) określonymi wartościami. Służą do tego następujące dyrektywy:

```
etykieta:  DB     wyrażenie, wyrażenie,...    ; inicjalizacja wartościami bajtowymi
etykieta:  DW     wyrażenie, wyrażenie,...    ; inicjalizacja słowami 16-bitowymi
```

Przykłady:

```
dana_8:   DB     5             ; pod adresem dana_1 umieszczona będzie wartość 5
tablica:  DB     'A', 'B', 'C' ; pod adresem tablica umieszczone zostaną kolejno kody ASCII
                                 liter A,B i C
tekst:    DB     'Wynik = ', 0 ; pod adresem tekst umieszczone zostaną kolejne kody ASCII
                                 tekstu, na końcu 0
dana_16:  DW     1234h         ; pod adresem dana_16 umieszczony zostanie bajt 12h, pod
                                 kolejnym 34h
adresy:   DW     adr_0, adr_1  ; w pamięci umieszczone zostaną dwa adresy adr_0 i adr_1
```

Dyrektywa ORG

W każdym segmencie jest tak zwany licznik położenia, który wskazuje bieżący adres (względny dla segmentu relokowalnego, bezwzględny dla segmentu absolutnego). Na początku (przy pierwszym użyciu segmentu) wartość licznika wynosi 0, jest ona później zmieniana po każdej instrukcji, zależnie od jej długości. Dyrektywa ORG ustala nową wartość licznika położenia. Typowym przykładem jej użycia jest wymuszenie określonego adresu, pod którym powinna być umieszczona sekwencja instrukcji, na przykład obsługa przerwania.

Przykład:

W segmencie absolutnym:

```
          ORG    0           ; następny rozkaz zostanie umieszczony pod adresem 0000h
          ...
          ORG    0Bh         ; następny rozkaz zostanie umieszczony pod adresem 000Bh
          ...
```

Jeśli sekwencja rozkazów po ORG 0 w podanym wyżej przykładzie zajmuje więcej niż 11 bajtów (adresy 0 .. 0Ah), to wystąpi zjawisko nakładania kodu (*code overlap*) i pojawi się błąd linkera. Poza bardzo specyficznymi sytuacjami jest to niedopuszczalne.

4. Wybór banku rejestrów

Do określenia który bank rejestrów jest bankiem bieżącym w danym fragmencie kodu służy dyrektywa USING (domyślnym bankiem rejestrów jest bank 0).

USING numer banku (0..3)

Informacja zawarta w dyrektywie USING jest wykorzystywana na dwa sposoby:

1. Dzięki niej linker posiada informację jakie banki rejestrów są używane i odpowiednio rezerwuje pamięć. Na przykład jeśli używany jest tylko bank 0, linker może umieścić segment danych z pamięci wewnętrznej rozpoczynając od adresu 08h. Jeśli używany będzie również bank 1, to pierwszym wolnym adresem będzie dopiero 10h.
2. Argumentem niektórych instrukcji (np. PUSH i POP) mogą być tylko adresy absolutne a nie symbole rejestrów. Nie jest dopuszczalna instrukcja PUSH R0, można zapisać natomiast PUSH AR0, gdzie AR0 będzie oznaczać adres rejestru R0 (0 dla banku 0, 8 dla banku 1 itd.). Aby możliwe było określenie tego adresu, konieczna jest informacja jaki bank rejestrów jest aktualnie używany.

Należy podkreślić, że dyrektywa USING nie generuje żadnego kodu i nie zwalnia programisty od przełączenia banku (przez odpowiednie ustawienie bitów w rejestrze PSW).

5. Praca z wieloma plikami

Jeśli program ma postać kilku modułów zawierających różne fragmenty programu, umieszczone w różnych plikach, to może zachodzić potrzeba udostępniania na zewnątrz symboli zdefiniowanych w danym module lub korzystania z symboli zdefiniowanych w innym module. Do tego celu służą dyrektywy PUBLIC i EXTRN.

PUBLIC symbol_1, symbol_2, ... udostępnia symbole zdefiniowane w bieżącym pliku źródłowym
EXTRN klasa (symbol_1, symbol_2,...) wylicza symbole zdefiniowane w innym pliku źródłowym

klasa - klasa pamięci (CODE, BIT, DATA, IDATA, XDATA) w której zostały zdefiniowane symbole podane w nawiasie. Może być również użyta klasa NUMBER dla symboli nie związanych z żadnym typem pamięci.

Przykład:

W pierwszym pliku określamy udostępniane na zewnątrz symbole:

```
PUBLIC  symbol_1, symbol_2, adres_1, adres_2, data_1, data_2

symbol_1 EQU    10
symbol_2 SET    symbol_1 + 20

         RSEG   PROG
adres_1: ...
adres_2: ...
         RSEG   D_DATA          ; wybór segmentu danych w pamięci wewnętrznej typu DATA
data_1:  DS     1               ; zarezerwowanie miejsca na dwie zmienne
data_2:  DS     1
```

W drugim pliku podajemy z jakich symboli zewnętrznych będziemy korzystać:

```
EXTRN   CODE            (adres_1, adres_2)
EXTRN   DATA            (data_1, data_2)
EXTRN   NUMBER          (symbol_1, symbol_2)
```

Warto zwrócić uwagę na sens podawania klasy pamięci. Bez tej informacji przy przetwarzaniu pliku zawierającego symbole zewnętrzne nie byłoby możliwe sprawdzenie, czy dany symbol jest używany we właściwym kontekście. Na przykład adres procedury lub skoku musi być wartością 16-bitową w pamięci kodu. Nie powinien on być użyty na przykład jako adres zmiennej w pamięci wewnętrznej.

6. Inne dyrektywy

Dyrektywa END powinna pojawić się na końcu programu w asemblerze. Ewentualny dodatkowy tekst występujący po tej dyrektywie jest ignorowany.

```
END
```

Do określenia nazwy modułu obiektowego (powstającego po przetłumaczeniu programu) służy dyrektywa NAME. Nazwa modułu jest czymś innym niż nazwa pliku.

```
NAME   nazwa modułu
```

Dyrektywa NAME nie jest obowiązkowa, może ona wystąpić w pliku tylko raz. Jeśli w pliku brak dyrektywy NAME, to jako nazwa modułu obiektowego przyjmowana jest nazwa pliku źródłowego (bez rozszerzenia).

7. Przykład programu

Poniżej podany jest przykład programu, w którym wykorzystano wiele różnych dyrektyw asemblera i pokazano jak można korzystać z segmentów. Program główny nie realizuje żadnego sensownego algorytmu, pokazuje tylko różne sposoby dostępu do zmiennych.

Przykład:

```
PROG        SEGMENT     CODE            ; deklaracja segmentu programu głównego
INT         SEGMENT     CODE            ; deklaracja segmentu obsługi przerwań
STACK       SEGMENT     IDATA           ; deklaracja segmentu stosu
BIT_DATA    SEGMENT     BIT             ; deklaracja segmentu danych bitowych
D_DATA      SEGMENT     DATA            ; deklaracja segmentu danych adresowanych
                                          bezpośrednio
I_DATA      SEGMENT     IDATA           ; deklaracja segmentu danych adresowanych tylko
                                          pośrednio
X_DATA      SEGMENT     XDATA           ; deklaracja segmentu danych w pamięci
                                          zewnętrznej

CONST_0     EQU         10              ; stała CONST_0 równa 10
CONST_1     SET         (CONST_0 + 5)   ; stała CONST_1 równa stałej CONST_0 + 5

            CSEG    AT  0              ; początek programu głównego pod adresem 0000h
            LJMP    start              ; skok do kodu programu (ominięcie obsługi
                                          przerwań)

            CSEG    AT  0Bh            ; pod adresem 0Bh początek obsługi przerwania
                                          Timera 0
            LJMP    int_timer_0        ; skok do właściwej obsługi przerwania
```

```
                CSEG    AT  1Bh          ; pod adresem 1Bh początek obsługi przerwania
                                           Timera 1
                LJMP    int_timer_1     ; skok do właściwej obsługi przerwania

                RSEG    INT             ; wybór segmentu obsługi przerwań
int_timer_0:    ...                     ; kod obsługi przerwania Timera 0
                RETI                    ; powrót z obsługi przerwania
int_timer_1:    ...                     ; kod obsługi przerwania Timera 1
                RETI                    ; powrót z obsługi przerwania

                RSEG    PROG            ; wybór segmentu programu głównego
start:          MOV     SP, #STACK-1    ; zainicjowanie wskaźnika stosu (uwaga: początek
                                           stosu - 1)
SETB            bit_0                   ; ustawienie bitu bit_0
                CLR     bit_1           ; skasowanie bitu bit_1
                MOV     d_data_0, #CONST_0 ; wpis wartości CONST_0 do zmiennej d_data_0
                MOV     A, d_data_1     ; wpis wartości zmiennej d_data_1 do akumulatora
                MOV     R0, #i_data_0   ; wpis adresu zmiennej i_data_0 do R0
                MOV     @R0, #CONST_1   ; wpis wartości CONST_1 do zmiennej i_data_0
                MOV     R1, #i_data_1   ; wpis adresu zmiennej i_data_1 do R1
                MOV     @R1, A          ; wpis akumulatora do zmiennej i_data_1
                MOV     DPTR, #x_data_0 ; załadowanie DPTR adresem zmiennej x_data_0
                MOVX    A, @DPTR        ; odczyt zmiennej x_data_0 do akumulatora
                MOV     DPTR, #x_data_1 ; załadowanie DPTR adresem zmiennej x_data_1
                MOVX    @DPTR, A        ; wpis akumulatora do zmiennej x_data_1

                SJMP    $               ; pętla

                RSEG    STACK           ; wybór segmentu stosu
                DS      30              ; zarezerwowanie 30 bajtów na stos

                RSEG    BIT_DATA        ; wybór segmentu danych bitowych
bit_0:          DBIT    1               ; zarezerwowanie miejsca na dwie zmienne bitowe
bit_1:          DBIT    1

                RSEG    D_DATA          ; wybór segmentu danych adresowanych
                                           bezpośrednio
d_data_0:       DS      1               ; zarezerwowanie miejsca na dwie zmienne
d_data_1:       DS      1

                RSEG    I_DATA          ; wybór segmentu danych adresowanych tylko
                                           pośrednio
i_data_0:       DS      1               ; zarezerwowanie miejsca na dwie zmienne
i_data_1:       DS      1

                RSEG    X_DATA          ; wybór segmentu danych w pamięci zewnętrznej
x_data_0:       DS      1               ; zarezerwowanie miejsca na dwie zmienne
x_data_1:       DS      1

                END                     ; koniec programu
```