Programowanie mikrokontrolera 8051

Podane poniżej informacje mogą pomóc w nauce programowania mikrokontrolerów z rodziny 8051. Opisane są tu pewne specyficzne cechy tych procesorów a także podane przykłady rozwiązywania niektórych elementarnych zadań programistycznych. Opis uwzględnia tylko cechy podstawowe procesora, bez dodatkowych możliwości oferowanych w niektórych rozszerzeniach.

1. Specyfika rejestrów

Akumulator

Akumulator jest rejestrem roboczym, najbardziej uniwersalnym ponieważ może być argumentem wielu rozkazów, w których użycie innego rejestru nie jest możliwe. Nie powinien być on używany do przechowywania danych w dłuższych sekwencjach programu, gdyż prawdopodobnie będzie potrzebny do bieżących operacji w kolejnych rozkazach.

Mikrokontroler 8051, w przeciwieństwie do wielu innych procesorów, nie posiada flagi zera. W związku z tym przy wykonywaniu rozkazów JZ i JNZ istotna jest zawsze bieżąca wartość akumulatora.

W rozkazach, w których akumulator jest specjalnym argumentem zapisywany jest on jako A. Można też traktować akumulator jako rejestr z obszaru SFR, i wówczas zapisywany jest on jako ACC (asembler tłumaczy taki zapis na adres akumulatora). Na przykład w poniższych rozkazach informacja, że jednym z argumentów jest akumulator zawarta jest już w kodzie rozkazu i nie jest możliwe użycie symbolu ACC:

```
CLR   A          ; zerowanie akumulatora
MOVX  A, @DPTR   ; wpis zawartości komórki pamięci zewnętrznej, adresowanej
                   przez DPTR, do akumulatora
ADD   A, @R0     ; dodanie do akumulatora zawartości komórki pamięci
                   wewnętrznej adresowanej przez R0
```

W niektórych rozkazach użycie symbolu ACC jest konieczne. Chodzi o rozkazy, w których argumentem może być tylko adres komórki pamięci wewnętrznej, na przykład:

```
PUSH  ACC        ; przesłanie akumulator na stos
DJNZ  ACC, skok  ; dekrementacja akumulatora i skok, jeśli po dekrementacji
                   wartość niezerowa
```

W pewnej grupie rozkazów argumentem może być zarówno A jak i ACC. Należy wówczas wybierać wersję z użyciem symbolu A, ponieważ daje ona rozkaz o kodzie krótszym o jeden bajt, na przykład:

Prawidłowo		Nieoptymalne		
INC	A	INC	ACC	; inkrementacja akumulatora
MOV	A, R7	MOV	ACC, R7	; wpis zawartości rejestru R7 do akumulatora

Możliwa jest nawet sytuacja, w której argumentami jednego rozkazu są symbole A oraz ACC. Niekiedy może to być sensowne, jak w przykładzie poniżej (chociaż można to zrobić w prostszy sposób):

```
ADD   A, ACC     ; dodanie akumulatora do akumulatora (pomnożenie przez 2 lepiej
                   wykonać przez RL A)
```

```
XRL     A, ACC          ; operacja Exclusive-OR akumulatora z nim samym (prostszy
                          wariant to CLR A)
```

Akumulator jest jednym z rejestrów z obszaru rejestrów specjalnych (SFR) które są adresowane bitowo. Przy operacjach bitowych bit akumulatora jest identyfikowany przez użycie symbolu ACC, po kropce podawany jest numer bitu:

```
SETB    ACC.0           ; ustawienie bitu 0 w akumulatorze
MOV     C, ACC.5        ; przesłanie bitu 5 akumulatora do CY
```

Rejestr B

Rejestr B jest rejestrem pomocniczym, który ma specjalne zastosowanie w operacjach mnożenia i dzielenia i nie może być w nich zastąpiony innym rejestrem. W pozostałych przypadkach funkcjonuje na prawach komórki pamięci i dlatego w większości przypadków nie jest opłacalne używanie go zamiast rejestrów R0 - R7. Rejestr B jest jednak adresowany bitowo (nie jest to możliwe dla rejestrów R0 - R7) co ilustrują poniższe przykłady:

```
SETB    B.3             ; ustawienie bitu 3 w rejestrze B
MOV     B.5, C          ; przesłanie CY do bitu 5 rejestru B
JB      B.0, skok       ; skok jeśli bit 0 rejestru B jest ustawiony
```

Rejestry R0 - R7

Rejestry R0 - R7 są uniwersalnymi rejestrami umieszczonymi w początkowym obszarze wewnętrznej pamięci RAM. Przyporządkowanie rejestrów do komórek pamięci nie jest stałe lecz zależy od aktualnie wybranego banku pamięci (bitami RS0, RS1 w rejestrze PSW). Przełączanie banków wykorzystuje się najczęściej w obsłudze przerwania.

Rejestry R0 i R1 są wyróżnione w ten sposób, że tylko one mogą być stosowane do pośredniego adresowania pamięci wewnętrznej.

2. Rozkazy arytmetyczne

Wykrywanie przepełnienia

Należy pamiętać, że rozkazy inkrementacji i dekrementacji (INC, DEC) nie ustawiają żadnych flag, nawet jeśli w wyniku operacji nastąpi przejście 0FFh → 0 lub 0 → 0FFh. Aby wykryć taką zmianę trzeba przeprowadzić dodatkowe sprawdzenie:

```
        INC     R7              ; inkrementacja R7
        CJNE    R7, #0, dalej
        ...                     ; akcja wykonywana jeśli nastąpiła zmiana 0FFh → 0
dalej:

        DEC     R7              ; dekrementacja R7
        CJNE    R7, #0FFH, dalej
        ...                     ; akcja wykonywana jeśli nastąpiła zmiana 0 → 0FFh
dalej:
```

Jeśli argumentem operacji jest komórka pamięci, to sprawdzenie może wyglądać następująco:

```
INC     direct          ; inkrementacja komórki pamięci
MOV     A, direct       ; kopiujemy komórkę pamięci do A
JNZ     dalej
...                     ; akcja wykonywana jeśli nastąpiła zmiana 0 → 0FFh
```

```
dalej:
        DEC     direct          ; dekrementacja komórki pamięci
        MOV     A, direct       ; kopiujemy komórkę pamięci do A
        INC     A               ; jeśli nastąpiło przejście 0 → 0FFh, to po inkrementacji A = 0
        JNZ     dalej
        ...                     ; akcja wykonywana jeśli nastąpiła zmiana 0 → 0FFh
dalej:
```

Ostatni przykład z dekrementacją można również zrealizować następująco:

```
        DEC     direct          ; dekrementacja komórki pamięci
        MOV     A, direct       ; kopiujemy R0 do A
        CJNE    A, #0FFH, dalej
        ...                     ; akcja wykonywana jeśli nastąpiło przepełnienie 0 → 0FFh
dalej:
```

Dodawanie i odejmowanie

Rozkaz dodawania, którego pierwszym argumentem jest zawsze akumulator (w nim pozostaje również wynik dodawania) jest dostępny w dwóch wariantach:

```
ADD     - dodawanie bez przeniesienia
ADDC    - dodawanie z przeniesieniem (flaga CY)
```

Przy dodawaniu wartości jednobajtowych wykorzystywany jest rozkaz ADD:

```
ADD     A, R7           ; A ← A + R7
```

Przy dodawaniu wartości wielobajtowych, do operacji na najmłodszych bajtach należy użyć rozkazu ADD, przy kolejnych zaś rozkazu ADDC. Poniżej pokazane jest to na przykładzie dodawania dwóch wartości 16 bitowych, podanych w parach rejestrów R4 | R5 oraz R6 | R7, wynik umieszczony będzie w parze rejestrów R6 | R7:

```
MOV     A, R5           ; młodszy bajt pierwszego składnika
ADD     A, R7           ; sumowanie młodszych bajtów
MOV     R7, A           ; młodszy bajt wyniku
MOV     A, R4           ; starszy bajt pierwszego składnika
ADDC    A, R6           ; sumowanie starszych bajtów z uwzględnieniem przeniesienia
MOV     R6, A           ; starszy bajt wyniku
```

Rozkaz odejmowania SUBB (odjemna zawsze w akumulatorze, tam również wynik) jest wykonywany zawsze z pożyczką której funkcję pełni flaga CY. Przy prostym odejmowaniu wartości jednobajtowych należy pamiętać o wyzerowaniu CY;

```
CLR     C               ; wyzerowanie pożyczki
SUBB    A, R7           ; A ← A - R7
```

Mnożenie i dzielenie

Do mnożenia i dzielenia służą rozkazy MUL AB oraz DIV AB, mają one ustalone argumenty, którymi są zawsze akumulator i rejestr B:

```
MUL     AB              ; wynik mnożenia A* B jest wartością 16-bitową B | A (młodszy
                          bajt w A)
DIV     AB              ; wynik dzielenia A / B w A, reszta z dzielenia w B
```

Wadą tych rozkazów jest stosunkowo długi czas wykonywania (4 cykle maszynowe). Przy mnożeniu lub dzieleniu przez 2 można wykorzystać zamiast nich rozkazy przesunięcia (rotacji) akumulatora. Wykorzystywany jest przy tym fakt, że mnożenie przez 2 odpowiada przesunięciu w lewo o jeden bit, zaś dzielenie przesunięciu o jeden bit w prawo:

```
        CLR     C               ; wyzerowanie przeniesienia (będzie to najmłodszy bit wyniku)
        RLC     A               ; A ← A ∗ 2,  w CY 9 bit wyniku

        CLR     C               ; wyzerowanie przeniesienia (będzie to najstarszy bit wyniku)
        RRC     A               ; A ← A / 2, w CY reszta z dzielenia
```

Jeśli mamy pewność, że wartość w akumulatorze jest mniejsza niż 128 (a więc ACC.7 jest wyzerowany), to mnożenie przez 2 można wykonać stosując tylko jeden rozkaz:

```
        RL      A               ; A ← A ∗ 2 (ACC. 0 ← ACC.7)
```

3. Porównywanie wartości

Porównanie z wykorzystaniem rozkazu CJNE

Lista rozkazów mikrokontrolera 8051 nie zawiera rozkazu porównania, który ustawiałby tylko flagę przeniesienia CY. Jest natomiast rozkaz CJNE (**C**ompare and **J**ump if **N**ot **E**qual) w którym ustawiana jest flaga CY oraz wykonywany skok jeśli porównywane argumenty nie są równe.

```
        CJNE    arg_1, arg_2, skok

        Jeśli arg_1 < arg_2         to CY = 1,    skok wykonywany jest zawsze
        Jeśli arg_1 >= arg_2        to CY = 0,    skok wykonywany jest tylko jeśli arg_1 > arg_2
```

Wykorzystanie rozkazu CJNE do sprawdzenia czy arg_1 < arg_2 polega na ustawieniu adresu skoku na następny rozkaz, tak więc niezależnie od rezultatu porównania skok jest wykonywany zawsze w to samo miejsce. Cały sens rozkazu CJNE sprowadza się wówczas do ustawienia flagi CY.

Sprawdzenie, czy akumulator jest mniejszy od stałej:

```
        CJNE    A, #data, skok  ; jeśli A < data to CY będzie ustawione
skok:   JC      mniejszy
        ...                     ; akcja wykonywana jeśli A >= data
        SJMP    dalej
mniejszy: ...                   ; akcja wykonywana jeśli A < data
dalej:
```

Jeśli konieczne jest wykonanie akcji tylko w jednym przypadku, należy odpowiednio dobrać warunek skoku (JC lub JNC) tak, aby uzyskać najprostszą strukturę programu:

```
        CJNE    A, #data, skok  ; jeśli A < data to CY będzie ustawione
skok:   JC      dalej
        ...                     ; akcja wykonywana jeśli A >= data
dalej:
        CJNE    A, #data, skok  ; jeśli A < data to CY będzie ustawione
skok:   JNC     dalej
        ...                     ; akcja wykonywana jeśli A < data
dalej:
```

Stosunkowo rzadko występuje sytuacja, w której musimy wykonać trzy różne akcje dla trzech możliwych wyników porównania. Program wówczas wyglądałby następująco:

```
        CJNE    A, #data, skok  ; jeśli A < data, CY będzie ustawione
        ...                     ; akcja wykonywana jeśli A = data
        SJMP    dalej
skok:   JC      mniejszy
        ...                     ; akcja wykonywana jeśli A > data
```

```
        SJMP    dalej
mniejszy: ...                   ; akcja wykonywana jeśli A < data
dalej:
```

Oczywiście, jeśli zależy nam tylko na wyróżnieniu przypadku równości, to mamy następujące trzy przypadki:

1. Określona akcja ma być wykonana tylko jeśli argumenty są równe:

```
        CJNE    A, #data, dalej
        ...                     ; akcja wykonywana jeśli A = data
dalej:
```

2. Określona akcja ma być wykonana tylko jeśli argumenty są różne:

```
        CJNE    A, #data, rozne
        SJMP    dalej
rozne:  ...                     ; akcja wykonywana jeśli A <> data
dalej:
```

3. Jeśli argumenty są równe, to wykonywana jest jedna akcja, w przeciwnym wypadku inna:

```
        CJNE    A, #data, rozne
        ...                     ; akcja wykonywana jeśli A = data
        SJMP    dalej
rozne:  ...                     ; akcja wykonywana jeśli A <> data
dalej:
```

Oprócz pokazanej możliwości porównania akumulatora ze stałą, rozkaz CJNE umożliwia jeszcze:

```
CJNE    A, direct, skok     ; porównanie akumulatora z komórką pamięci
CJNE    Rn, #data, skok     ; porównanie rejestru R0..R7 ze stałą
CJNE    @Ri, #data, skok    ; porównanie komórki pamięci adresowanej przez R0 lub R1 ze
                              stałą
```

Porównanie z wykorzystaniem odejmowania

Do porównania wartości można zastosować rozkaz odejmowania, wykorzystując fakt że operacja ta ustawia przeniesienie. Ilustruje to przykład porównania rejestru ze stałą:

```
CLR     C               ; wyzerowanie pożyczki
MOV     A, R7           ; wpisanie wartości porównywanej do akumulatora
SUBB    A, #data        ; jeśli A < data to CY = 1, w przeciwnym wypadku CY = 0
JC      dalej           ; skok jeśli A < data (R7 < data)
...                     ; akcja wykonywana jeśli R7 >= data
```

Podany powyżej przykład pozwala rozróżnić dwa przypadki: R7 < data oraz R7 >= data. Jeśli zachodzi potrzeba aby przypadek równości był połączony z relacją mniejszości (rozróżnienie przypadków R7 <= data oraz R7 > data), to można to zrobić na dwa pokazane niżej sposoby:

```
SETB    C               ; ustawienie pożyczki
MOV     A, R7           ; wpisanie wartości porównywanej do akumulatora
SUBB    A, #data        ; jeśli A < data + 1 to CY = 1, w przeciwnym wypadku CY = 0
JC      dalej           ; skok jeśli A <= data (R7 < =data)
...                     ; akcja wykonywana jeśli R7 > data

CLR     C               ; wyzerowanie pożyczki
MOV     A, #data        ; wpisanie wartości stałej do akumulatora
SUBB    A, R7           ; jeśli A < R7 to CY = 1, w przeciwnym wypadku CY = 0
JNC     dalej           ; skok jeśli A >= R7 (R7 < =data)
...                     ; akcja wykonywana jeśli R7 > data
```

Porównanie z wykorzystaniem operacji logicznych

Do sprawdzanie, czy dwie wartości są równe można wykorzystać operację logiczną XOR realizowaną przez rozkaz XRL.

Sprawdzenie, czy rejestr R7 jest równy stałej wartości:

```
MOV   A, R7       ; załadowanie rejestru R7 do akumulatora
XRL   A, #data    ; wynik operacji XOR jest równy 0 jeśli wartości są równe
JNZ   dalej       ; skok jeśli wartości są różne
```

Sprawdzenie, czy rejestry R6 i R7 są równe:

```
MOV   A, R6       ; załadowanie rejestru R6 do akumulatora
XRL   A, R7       ; wynik operacji XOR jest równy 0 jeśli wartości są równe
JNZ   dalej       ; skok jeśli wartości są różne
```

4. Realizacja licznika dwubajtowego

W wielu przypadkach zachodzi potrzeba skorzystania z licznika dwubajtowego. Jedynym rejestrem na którym można wykonywać operacje 16-bitowej inkrementacji jest DPTR. Ponieważ jednak rejestr ten zwykle jest potrzebny przy dostępie do pamięci zewnętrznej (poza tym nie może być on dekrementowany) pokazane zostanie jak zrealizować 16 bitowy licznik używając pojedynczych rejestrów 8-bitowych, w tym przykładzie R6 i R7.

```
LICZNIK EQU   1000                    ; liczba cykli licznika
```

Wersja z dekrementacją licznika

```
        MOV   R6, #HIGH(LICZNIK)      ; załadowanie starszej części licznika
        MOV   R7, #LOW(LICZNIK)       ; załadowanie młodszej części licznika
petla:  ...                           ; akcja wykonywana w pętli

        MOV   A, R7                   ; młodsza część licznika (przed dekrementacją)
        DEC   R7                      ; dekrementacja młodszej części licznika
        JNZ   skok                    ; jeśli przed dekrementacją niezerowa, to nic nie
                                        robimy
        DEC   R6                      ; dekrementacja starszej części licznika
skok:   MOV   A, R7                   ; młodsza część licznika (po dekrementacji)
        ORL   A, R6                   ; operacja OR na starszej i młodszej części
                                        licznika
        JNZ   petla                   ; licznik niezerowy, kontynuacja pętli
```

Wersja z inkrementacją licznika

```
        MOV   R6, #0                  ; wyzerowanie starszej części licznika
        MOV   R7, #0                  ; wyzerowanie młodszej części licznika
petla:  ...                           ; akcja wykonywana w pętli

        INC   R7                      ; inkrementacja młodszej części licznika
        CJNE  R7, #0, skok            ; skok jeśli brak przepełnienia w młodszej części
        INC   R6                      ; przepełnienie młodszej części, inkrementacja
                                        starszej
skok:   CJNE  R7, #LOW(LICZNIK), petla ; młodsza część nie osiągnęła wartości granicznej
        CJNE  R6,#(HIGH(LICZNIK), petla ; starsza część nie osiągnęła wartości granicznej
```

5. Problem zasięgu krótkich skoków

Rozkaz SJMP jak też wszystkie skoki warunkowe (JZ, JNZ, JC, JNC, JB, JNB, JBC) są rozkazami skoków krótkich. Adres skoku (zakodowany w jednym bajcie) jest względnym przesunięciem w stosunku do pierwszego bajtu kolejnej instrukcji, w zakresie -128 do + 127. Przy dłuższych programach może wystąpić sytuacja, w której zakres skoku jest niewystarczający (wystąpi odpowiedni komunikat o błędzie):

```
        JZ      dalej
        ...                     ; długi fragment programu
dalej:
```

Rozwiązaniem tego problemu jest zmiana warunku skoku i użycie rozkazu długiego skoku:

```
        JNZ     tutaj
        LJMP    dalej
tutaj:  ...                     ; długi fragment programu
dalej:
```

6. Operacje bitowe

Porównywanie bitów

Operacje bitowe nie obejmują rozkazu XRL, tak więc nie ma rozkazu, który umożliwiałby wprost porównanie bitów. Porównanie bitów bit_0 i bit_1 można zrealizować następująco:

```
        MOV     C, bit_0        ; przesłanie bit_0 do CY
        JNB     bit_1, bit_1_0  ; skok jeśli bit_1 = 0
        CPL     C               ; tutaj  bit_1 = 1, w CY zanegowany bit_0
bit_1_0: JC     rozne           ; jeśli CY = 1 to: (bit_0 = 1 i bit_1 = 0) lub (bit_0 = 0 i bit_1 = 1)
równe:          ...             ; akcja wykonywana, jeśli bity równe
        SJMP    dalej           ; jeśli dla bitów różnych brak akcji to skok jest zbędny
rozne:          ...             ; akcja wykonywana, jeśli bity różne
dalej:
```

W powyższym przykładzie można zmienić typ skoku z JC na JNC aby był on wykonywany w przypadku, gdy oba bity są równe.

Modyfikacja zadanej grupy bitów

Poniższe przykłady pokazują jak można wyzerować, ustawić lub zanegować zadaną grupę bitów w bajcie (w szczególnym przypadku może to być tylko jeden bit). Maska w podanych przykładach jest wartością stałą ale drugim argumentem rozkazów ANL, ORL i XRL może być również rejestr R0 - R7 lub komórka pamięci (adresowana bezpośrednio lub pośrednio).

```
maska   EQU     00111100B       ; maska dla grupy bitów b5 - b2
        ANL     A, #NOT(maska)  ; wyzerowanie zadanych bitów (bit AND 0 = 0), operator NOT
                                  neguje bity maski na 11000011B
        ORL     A, #maska       ; ustawienie zadanych bitów (bit OR 1 = 1)
        XRL     A, #maska       ; zanegowanie zadanych bitów (bit XOR 1 = not bit)
```

Testowanie zadanej grupy bitów

Sprawdzenie, czy przynajmniej jeden z zadanych bitów jest ustawiony :

```
MOV   A, R7          ; testujemy bity w R7
ANL   A, #maska      ; bity odpowiadające 0 w masce są zerowane, pozostałe bez zmian
JZ    dalej          ; skok jeśli wszystkie zadane bity są zerowe
      ...            ; akcja jeśli przynajmniej jeden z zadanych bitów jest ustawiony
```

Sprawdzenie, czy wszystkie zadane bity są ustawione:

```
MOV   A, R7          ; testujemy bity w R7
ANL   A, #maska      ; bity odpowiadające 0 w masce są zerowane, pozostałe bez zmian
XRL   A, #maska      ; jeśli bit w grupie jest ustawiony, to daje wynik 0 (1 XOR 1 = 0)
JNZ   dalej          ; skok jeśli nie wszystkie zadane bity są ustawione
...                  ; akcja, jeśli wszystkie zadane bity są ustawione
```

Sprawdzenie, czy przynajmniej jeden z zadanych bitów jest wyzerowany:

```
MOV   A, direct      ; testujemy bity w komórce pamięci wewnętrznej
ANL   A, #maska      ; bity odpowiadające 0 w masce są zerowane, pozostałe bez zmian
XRL   A, #maska      ; jeśli bit w grupie jest ustawiony, to daje wynik 0 (1 XOR 1 = 0)
JZ    dalej          ; skok jeśli wszystkie zadane bity są ustawione
...                  ; akcja, jeśli przynajmniej jeden z zadanych bitów jest
                       wyzerowany
```

Sprawdzenie, czy wszystkie zadane bity są wyzerowane:

```
MOV   A, @R0         ; testujemy bity w komórce pamięci wewnętrznej adresowanej
                       przez R0
ANL   A, #maska      ; bity odpowiadające 0 w masce są zerowane, pozostałe bez zmian
JNZ   dalej          ; skok jeśli przynajmniej jeden z zadanych bitów jest ustawiony
...                  ; akcja jeśli wszystkie zadane bity są wyzerowane
```

7. Obsługa stosu

Stos wykorzystywany jest przede wszystkim do zapamiętywania adresu powrotu przy wywołaniu procedur oraz tymczasowego przechowywania danych (rozkazy PUSH i POP). Przy obsłudze stosu wykorzystywany jest rejestr SP (**S**tack **P**ointer) czyli wskaźnik stosu, który zawiera adres ostatniej zajętej komórki pamięci stosu. Wynika stąd, że przed odłożeniem danej na stos najpierw zwiększany jest wskaźnik stosu a następnie dana wpisywana do komórki adresowanej przez SP. Przy pobieraniu danej najpierw odczytywana jest wartość komórki adresowanej przez SP a następnie wartość wskaźnika stosu jest zmniejszana.

W rozkazach LCALL i ACALL (jak również przy obsłudze przerwania) na stosie zapamiętywany jest 16-bitowy licznik rozkazów PC (**P**rogram **C**ounter) w ten sposób, że najpierw na stos odkładany jest młodszy bajt a później starszy (będzie on umieszczony na szczycie stosu).

Po restarcie procesora wskaźnik stosu ma wartość 7, czyli rezerwowany jest obszar banku 0 rejestrów R0 - R7 (trudno wyobrazić sobie sensowny program nie używający tych rejestrów). W każdym przypadku obowiązkiem programisty jest umieszczenie stosu w bezpiecznym obszarze pamięci, tak aby nie zostały zniszczone używane komórki pamięci.

Poniżej pokazany zostanie typowy sposób zainicjowania stosu:

```
rozmiar  EQU      30      ; rozmiar stosu (przykładowo 30 bajtów)
PROG     SEGMENT  CODE    ; deklaracja segmentu programu
```

```
STOS    SEGMENT   IDATA      ; deklaracja segmentu stosu
RSEG    PROG                 ; wybór segmentu programu
        MOV       SP, #STOS-1; na początku programu zainicjowanie wskaźnika stosu
        ...                  ; wyjaśnienie dlaczego STOS-1 poniżej
RSEG    STOS                 ; wybór segmentu stosu
        DS        rozmiar    ; zarezerwowanie obszaru na stos
```

Warto zauważyć, że nazwa segmentu STOS oznacza adres początkowy segmentu w pamięci. Ponieważ wskaźnik stosu pokazuje zawsze ostatni zajęty bajt stosu, dlatego inicjowany jest on adresem mniejszym o 1 niż adres początkowy stosu.

Podstawową zasadą pracy ze stosem jest dbanie o to, aby rozkazy PUSH i POP były używane parami, to znaczy każdy rozkaz PUSH powinien mieć swój odpowiednik w postaci POP (z wyjątkiem zaawansowanych operacji na stosie). Niezamierzone pozostawienie danej na stosie w programie głównym oznacza wyłączenie obszaru stosu z dalszego używania. Wykonanie takiej operacji w procedurze spowoduje błędny powrót przy rozkazie RET.

Najbardziej typowym błędem jest również przejście do procedury (czyli fragmentu programu zakończonego rozkazem RET) rozkazem skoku LJMP zamiast rozkazem LCALL. Rozkaz RET na końcu procedury wykona błędne pobranie ze stosu (i załadowanie do licznika rozkazów PC) danych, które nie są adresem powrotu (nie zostały tam wcześniej odłożone).

8. Powrót z przerwania

Należy pamiętać, aby obsługę przerwania kończyć zawsze rozkazem RETI a nie RET. W obu tych rozkazach operacja powrotu do programu głównego wykonana zostanie tak samo. Jednak rozkaz RETI informuje dodatkowo system przerwań procesora o zakończeniu obsługi przerwania. Jeśli zostanie użyty rozkaz RET, to kolejne przerwanie (o tym samym lub niższym priorytecie) nie będzie przyjęte gdyż system przerwań uważa, że procesor jest w trakcie obsługi przerwania.

9. Nietypowe rozkazy

Choć może się to wydać dziwne, możliwa jest sytuacja, że rozkaz formalnie poprawny, przetłumaczony przez asembler, nie zostanie jednak wykonany. Przykładem może być tu rozkaz zerowania lub ustawiania bitu parzystości.

Bit parzystości jest najmłodszym bitem w rejestrze PSW i wiąże się z zawartością akumulatora (jego wartość jest taka, aby liczba bitów ustawionych łącznie w akumulatorze i bicie parzystości była zawsze parzysta). Przyjrzyjmy się następującemu przykładowi:

```
MOV   A, #1     ; wartość 1 w akumulatorze, P = 1
CLR   P         ; próba wyzerowania bitu parzystości - niepowodzenie
MOV   A,#3      ; wartość 3 w akumulatorze, P = 0
SETB  P         ; próba ustawienia bitu parzystości - niepowodzenie
```

Oba rozkazy w powyższym przykładzie nie mogły się wykonać, gdyż doprowadziłoby to do sytuacji, w której bit parzystości nie byłby zgodny ze stanem akumulatora, co przeczyłoby jego definicji.