Programowanie niskopoziomowe

Wojciech Kordecki
Zakład Pomiarowej i Medycznej Aparatury Elektronicznej
Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

Wrocław 2002

1 Wstęp

Czym będziemy się zajmować?

1. Procesor Intel 8051,
    - architektura procesora,
    - rozkazy asemblera,
    - pamięć i jej adresowanie,
    - operacje arytmetyczne i logiczne,
    - zegary i liczniki,
    - operacje wejścia/wyjścia czyli o portach,
    - podprogramy i programy.

2. Rodzina 80x86.

3. Koprocesory.

Oprócz tego:

- Jak wyobrazić sobie procesor pisząc program w C++?
- Wstawki w asemblerze w programach pisanych w Pascalu i C++.
- Przykłady programów pisanych na PC-ta w asemblerze
- Trochę o procesorach innych niż Intel.

Literatura i programy dla procesora 8051:

[1] A. Rydzewski, *Mikrokomputery jednoukładowe rodziny MCS-51* Wydawnictwa Naukowo-Techniczne, Warszawa 1992.

[2] *Podstawy programowania mikroprocesora 8051*, Mikom, Warszawa 1995

[3] K. P. Dyrcz, C. T. Kowalski, Z. Żarczyński, *Podstawy techniki mikroprocesorowej*, Oficyna Wydawnicza P. Wr., Wrocław 1999.

[4] T. Starecki, *Mikrokontrolery jednoukładowe rodziny 51*, „NOZOMI", Warszawa 1996.

Używany na wykładzie asembler i disasembler procesora 8051

- asembler Dsm51ass [2],
- symulator – debugger SD51

Literatura dla procesorów 80x86:

[1] J. Duntemann, *Zrozumieć asembler*, Wiley – Translator, Warszawa ????

[2] A. Dudek, *Jak pisać wirusy*, Read Me, Warszawa 1994.

[3] http://www.binboy.org/

2 Procesor Intel 8051

2.1 Ogólne informacje o procesorach rodziny 8051

Procesory rodziny 8051 są ośmiobitowe, tzn. operują wyłącznie na danych ośmiobitowych. Produkowane są przez firmę Intel. Mają liczne wersje rozszerzone i zubożone.
Ich odpowiedniki są produkowane przez liczne firmy, m.in. Siemens, Philips, Atmel, AMD i in.

Procesory Intel 8051/52 jednoukładowe, tzn. w jednym układzie są:

- wewnętrzna pamięć danych
- wewnętrzna pamięć programu
- rejestry,
- układ czasowo-licznikowy,
- linie wejścia-wyjścia,
- port szeregowy,
- układ przerwań.

Pamięć wewnętrzna.

- Można adresować nie tylko bajty, ale i bity.
- Można adresować do 256 bajtów (od 00h do FFh) i tyle samo bitów.
- Pierwsze 128 bajtów 128 bitów znajduje się w pamięci danych, a pozostałe w w rejestrach specjalnych SFR (special function registers).

2.2 Pamięć wewnętrzna i SFR

Pamięć wewnętrzna o adresach od 00h do 7Fh ma postać

Zakres	Nazwa	przeznaczenie
00h – 07h	R0 – R7	Rejestry. użytkowe 1
08h – 0Fh	R0 – R7	Rejestry. użytkowe 2
10h – 17h	R0 – R7	Rejestry. użytkowe 3
18h – 1Fh	R0 – R7	Rejestry. użytkowe 4
20h – 2Fh		Obszar adresowany bitowo
30h – 7Fh		Obszar danych

W obszarze danych umieszcza się stos.

Obszar 00h – 1Fh może być również traktowany tak, jak obszar 30h – 7Fh.

Niektóre rejestry z obszaru SFR.

- Akumulator A lub Acc – najbardziej uniwersalny rejestr.
- B – rejestr pomocniczy, używany m.in. przy dzieleniu i mnożeniu.
- DPH i DPL, razem dają DPTR – wskaźnik danych.
- SP – wskaźnik stosu.
- PSW – słowo stanu programu.

2.3 Pierwsze kroki w asemblerze

Rozkazy nie mogą zaczynać się od początku wiersza.
Od początku wiersza zaczyna się etykieta, po której następuje znak ':'
Tekst po znaku ';' nie jest przetwarzany.
Najczęściej spotykanymi rozkazami są MOV, SJMP, LJMP.
Rozkaz MOV ma dwa argumenty: pierwszy dokąd ładować (adres komórki pamięci), drugi skąd lub co ładować.
Rozkaz SJMP ma jeden argument, którym może być etykieta, powoduje skok do niej, ale nie dalej niż o 128 bajtów kodu.
Rozkaz LJMP działa podobnie, ale nie ma ograniczenia długości skoku.
Liczba n w rozkazie oznacza adres rejestru, a $\#n$ oznacza liczbę (wartość).

Dyrektywa ORG, której argumentem jest adres powoduje, że następujący po nim program jest asemblowany od tego adresu.

```
          LJMP start
          ORG 03h
          SJMP alarm    ;nie jest wykonywany
          ORG 20H
start:    MOV 10h,#10   ;liczba 10 pod adres 16
          MOV A,10h     ;teraz do akumulatora
          MOV 11h,A     ;akumulator pod adres 17
          SJMP start    ;i od początku
alarm:    SJMP $
```

Inne często stosowane rozkazy:
INC – zwiększa o 1 zawartość komórki pamięci o adresie będącym argumetem rozkazu.
DEC – zmniejsza j.w.
JZ – skok jak w SJMP, ale tylko wtedy, gdy zawartość Acc jest równa zeru,
JNZ – j.w., ale gdy zawartość Acc jest różna od zera.

```
          LJMP start
          ORG 03h
          SJMP alarm
          ORG 20H
start:    MOV A,#250    ;liczba 250 do akumulatora
loop:     INC A
          JNZ loop      ;aż się licznik przekręci
alarm:    SJMP $        ;do nieskończoności
```

3 Asembler

3.1 Uwagi ogólne

Typowa linia programu (na przykładzie asemblera Dsm51ass):
`[<etykieta>] [<rozkaz>] [<operandy>] [;<komentarz>]`

`<etykieta>` – zaczyna się od litery lub znaku podkreślenia `_`, i może zawierać dowolną kombinację liter, cyfr i podkreśleń. Pierwszy znak etykiety jest pierwszym znakiem w linii. Jeśli etykieta jest zakończona dwukropkiem to jej wartością jest pozycja w kodzie źródłowym, czyli adres rozkazu z tej linii programu. Etykiety (symbole) stosowane z dyrektywami nadającymi im wartość nie są zakończone dwukropkiem.

`<rozkaz>` – mnemonik kodu maszynowego procesora, dyrektywa asemblera lub makro.

`<operandy>` – informacje wymagane przez mnemonik, dyrektywę asemblera lub makro. Poszczególne operandy są oddzielane przecinkami.

`<komentarz>` - wszystkie znaki występujące po średniku są ignorowane przez asembler.

Poszczególne pola linii programu są oddzielane co najmniej jednym znakiem spacji lub tabulacji. W programie mogą występować puste linie lub linie zawierające wyłącznie komentarz.

Stała liczbowa musi zaczynać się od cyfry.

Typ	Składnia	Przykład
Dziesiętny	`<cyfry>`	`125`
Szesnastkowy	`<cyfra><cyfry szesnastkowe>H`	`0FFFFH`
Ósemkowy	`<cyfry ósemkowe>O`	`77770`
Binarny	`<cyfry binarne>B`	`10101B`
Znakowy	`'<znak>'`	`'A'`

Stałą jest rozkaz NOP – tam gdzie został umieszczony, wstawia zero w kodzie programu w pamięci programu. Jego wykonanie trwa jeden cykl.

Dyrektywy danych:

`DB` – wstawienie w kod wartości numerycznych i tekstowych,

`DW` – wstawienie w kod dwubajtowych wartości numerycznych,

`EQU` – definiowanie stałej,

`BIT` – definiowanie stałej typu bit,

`REG` – definiowanie stałej typu rejestr,

`SET` – definiowanie zmiennej.

Przykład.

```
ok_message DB 'OK'
two_int DW 0403h,0201h
const_zero EQU 0
bit_1 BIT 1
```

Predefiniowanym symbolem jest ACC, który jest równy E0h, jest to adres akumulatora w SFR.

Dyrektywy sterujące:

`IF` – początek bloku warunkowej asemblacji,

`ELSE` – początek alternatywnego bloku warunkowej asemblacji,

`ENDIF` – koniec bloku warunkowej asemblacji,

`ORG` – ustawienie adresu dla następnego bloku kodu,

`END` – koniec programu.

Przykład.

```
1         0000   const_next EQU 0
2         [01]         IF const_next
3                            SJMP next ;jezeli const_next<>0
4         [01]         ELSE
5  0000:  80 FE               SJMP $    ;jezeli const_next=0
6         [00]         ENDIF
7  0002:  00                  NOP
8  0003:  80 FE   next:       SJMP $
9                           END
```

```
1         0001   const_next EQU 1
2         [01]       IF const_next
3  0000: 80 01             SJMP next ;jezeli const_next<>0
4         [01]       ELSE
5                         SJMP $    ;jezeli const_next=0
6         [00]       ENDIF
7  0002: 00                NOP
8  0003: 80 FE  next:      SJMP $
9                        END
```

Dyrektywy makrodefinicji:

`MACRO` – początek definicji makra,

`ENDM` – koniec definicji makra.

W makrodefinicji można umieszczać parametry, nie można lokalnych etykiet (w tym asemblerze).

Przykład.

```
five  MACRO ptr ;umieszcza liczbę 5 pod adresem ptr
          MOV R0,ptr
          MOV @R0,#5

      ENDM
          five #5
          five #6
          five #7

          END
```

Po zasemblowaniu otrzymujemy

```
     1                    five  MACRO ptr ;umieszcza liczbe 5 pod
     2                              MOV R0,ptr
     3                              MOV @R0,#5
     4                          ENDM
     5
     6                              five #5
>    1  0000: 78 05                 MOV R0,#5
>    2  0002: 76 05                 MOV @R0,#5
>    3                          ENDM
     7                              five #6
>    1  0004: 78 06                 MOV R0,#6
>    2  0006: 76 05                 MOV @R0,#5
```

```
>     3                         ENDM
      8                             five #7
>     1  0008: 78 07                MOV R0,#7
>     2  000A: 76 05                MOV @R0,#5
>     3                         ENDM
      9
     10                             END
```

3.2 Przesłania dla pamięci wewnętrznej

Rozkaz przesłania danej jednobajtowej MOV.
Pierwszy argument wskazuje dokąd należy przesłać bajt. Może to być: akumulator A, adres bezpośredni, rejestr z aktywnego zbioru rejestrów (od R0 do R7), adres pośredni (zawarty w rejestrze R0 lub R1 z aktywnego zbioru rejestrów, czyli @R0 lub @R1).
Drugi argument wskazuje co należy przesłać. Może to być jak poprzednio: akumulator A, adres bezpośredni, rejestr, adres pośredni lub liczba (postać #n).
Nie można przesłać w jednym rozkazie umieścić dwukrotnie symbolu rejestru.

Nielegalne są też rozkazy

```
MOV A,ACC
MOV ACC,A
```

Rozkazy przesłań do i z akumulatora są wykonywane szybciej od wielu innych. W szczególności rozkaz

```
MOV ACC,R0
```

zajmuje 2 cykle, a rozkaz

```
MOV A,R0
```

tylko jeden cykl.

Zestaw aktywnych rejestrów Rr jest wybierany przez dwubitową liczbę RS1 RS0 w rejestrze PSW.

Rozkazy wymiany XCH: pierwszym argumentem jest zawsze A, drugim adres bezpośredni, rejestr lub adres pośredni.
Wszystkie przesłania i wymiany operują na danych ośmiobitowych, z wyjątkiem rozkazu

```
MOV DPTR,#nn
```

gdzie #nn jest liczbą szesnastobitową.

3.3 Skoki i rozkazy sterujące

Skok długi LJMP – jego argumentem jest liczba szesnatobitowa – adres w pamięci programu. Jest ona wpisywana do PC i program wykona rozkaz umieszczony pod tym adresem. Jako argument LJMP należy umieścić etykietę.
Rozkaz

```
JMP @A+DPTR
```

spowoduje skok pod adres umieszczony w DPTR i zwiększony o zawartość akumulatora.
Rozkaz AJMP ma znaczenie historyczne i nie należy go używać.

Skoki krótkie – tylko o wielkość w zakresie $-128, 127$. Jako argument należy podać etykietę. Asembler sam obliczy wielkość skoku lub da komunikat błędu o za długim skoku.
SJMP – skok bezwarunkowy,
JC – jak SJMP, ale zostanie wykonany, gdy bit CY jest ustawiony.
JNC – jak JC, ale gdy CY jest wyzerowany,
JB, JNB – jak JC i JNC, ale argumentem rozkazu jest adres dowolnego bitu,
JBC – jak JB, po wykonaniu skoku bit jest zerowany,
JZ i JNZ – jak poprzednio, ale gdy akumulator jest lub nie jest wyzerowany.

Rozkaz CJNE porównuje pierwszy argument z drugim i gdy nie są równe, to wykonuje skok (krótki) o wielkość określoną przez trzeci argument. Trzeci argument powinien być etykietą. Jeżeli pierwszym argumentem jest A, to drugi może być albo adres bezpośredni albo liczba (postaci #n). Jeżeli pierwszym argumentem jest rejestr Rr, to drugim musi być liczba. Innych Możliwości nie ma.
Rozkaz DJNZ zmniejsza pierwszy argument o 1 i jeżeli stanie się on zerem, to zostanie wykonany skok (krótki) o wielkość określoną przez drugi argument. Drugi argument powinien być etykietą. Pierwszy argument musi być albo rejestrem Rr albo adresem bezpośrednim.

3.4 Podprogramy

Stos jest umieszczony w pamięci wewnętrznej. Wierzchołek stosu jest wskazywany przez rejestr SP. Rozkaz PUSH powoduje położenie jego argumentu, który jest adresem bezpośrednim na stosie i zwiększenie SP o 1, a rozkaz POP powoduje zdjęcie ze stosu bajtu i zapisanie jego wartości pod adresem, który jest argumentem rozkazu, a także zmniejszenie SP o 1.
Na początku programu powinno się ustalić wartość SP, czyli określić dno stosu.
Po zresetowaniu procesora, do rejestru SP wpisywana jest wartość 7. Nie należy polegać na domyślnych wartościach.

Rozkaz LCALL ma podobnie jak LJMP jeden argument, ale dodatkowo odkłada na stos adres powrotu, najpierw młodszy, potem starszy bajt. Bezargumentowy rozkaz RET wykonuje skok pod położony na stosie adres i zdejmuje go ze stosu. Wykorzystując rozkazy PUSH i POP można zmienić adres powrotu.

4 Pamięć i urządzenia zewnętrzne

4.1 Pamięć danych i pamięć programu

Do procesora można przyłączyć zewnętrzną pamięć danych o rozmiarze do 64 kB oraz zewnętrzną pamięć programu o rozmiarze do 64 kB. Niektóre wersje procesorów rodziny Intel 51 nie mają wewnętrznej pamięci programu.
W czasie wykonywania programu adres rozkazu w pamięci programu jest umieszczany w rejestrze PC.
Dane mogą być umieszczone w kodzie programu, np. dyrektywami asemblera DB oraz DW. Odczytywanie tych danych jest możliwe rozkazami MOVC A,@A+DPTR oraz MOVC A,@A+PC.

Rozkaz MOVC A,@A+DPTR powoduje umieszczenie w akumulatorze zawartości bajtu zapisanego w pamięci programu pod adresem, który jest sumą zawartości akumulatora i rejestru DPTR. Program

```
        MOV DPTR,#data
        MOV A,#3
        MOVC A,@A+DPTR
        ORG 0020h
data:   DB 'Ala ma kota'
        END
```

umieszcza w akumulatorze 20h, czyli kod spacji.

Rozkaz MOVC A,@A+PC powoduje umieszczenie w akumulatorze zawartości bajtu zapisanego w pamięci programu pod adresem, który jest sumą zawartości akumulatora i rejestru PC. PC jest równe adresowi po pobraniu rozkazu. Program

```
MOV A,#3
MOVC A,@A+PC
DB 'Ala ma kota'
END
```

umieszcza w akumulatorze 20h, czyli kod spacji.

Pamięć danych jest pamięcią zewnętrzną. Odczyt i zapis danej jest dokonywany rozkazem MOVX za pośrednictwem akumulatora.
MOVX A,@DPTR – bajt do akumulatora z pamięci zewnętrznej, adres wskazywany umieszczony w DPTR,
MOVX @DPTR,A – bajt z akumulatora do pamięci zewnętrznej, adres wskazywany umieszczony w DPTR.
Jest też możliwe adresowanie przy pomocy rejestru R0 lub R1 (bajt młodszy adresu), gdy ustalony jest bajt starszy adresu.

Przykład. Przepisanie łańcucha znaków z pamięci programu do pamięci zewnętrznej.

```
tab_x     EQU 10h
          MOV R0,#12          ;rozmiar tablicy
          MOV DPTR,#tab_x     ;adres tablicy - dane
          CLR A               ;adres - kod
          MOV B,A             ;  i do B
loop:     PUSH DPH            ;zapamiętanie adresu
          PUSH DPL            ;  dla danych
          MOV DPTR,#tab_c     ;adres tablicy - kod
          MOVC A,@A+DPTR      ;bajt w ACC
          POP DPL             ;odtworzenie adresu
```

```
        POP DPH             ;  dla danych
        MOVX @DPTR,A        ;bajt do pam. zewn.
        INC B               ;nast. adres - kod
        MOV A,B             ;  i do ACC
        INC DPTR            ;nast. adres - pam. zewn.
        DJNZ R0,loop
        SJMP $
        ORG 30h
tab_c:  DB 'Ala ma kota'
        DB 0                ;koniec napisu
```

4.2 Porty

Procesor 8051/52 ma cztery porty P0, P1, P2, P3.
Na ogół P2 służy do adresowania starszego bajtu, a P0 młodszego przy korzystaniu z pamięci zewnętrznej.
Rozkaz MOVX A,@DPTR m.in. wpisuje do portu P2 zawartość DPH, a do P0 zawartość DPL - tylko w czasie operacji odczytu/zapisu.
Jeżeli do P2 wpisany jest starszy bajt, zamiast rozkazu
MOVX A,@DPTR można użyć MOVX A,@Ri
zamiast
MOVX @DPTR,A można użyć MOVX @Ri,A
gdzie Ri jest albo R0 albo R1.

Port P1 służy do bezpośredniej komunikacji ze światem zewnętrznym.
Port P3 pełni również inne funkcje.
Przed odczytem danej z portu lub jednej jego linii, należy przedtem wpisać tam jedynkę.

4.3 Urządzenia zewnętrzne

Urządzenia zewnętrzne mają tę samą przestrzeń adresowa, co i zewnętrzna pamięć danych.
Jeżeli np. wyświetlacz ma podłączone kolejne litery pod adresy począwszy od 10h, to program przepisywania tablicy umieści napis 'Ala ma kota' na tablicy świetlnej. Na końcu łańcucha jest zero (jak w C).
Inna wersja tego programu, wyświetla napis aż do końca, czyli do naptkania znaku o kodzie 0.

```
tab_x     EQU 10h
          MOV R0,#100          ;maks. rozmiar tablicy
          MOV DPTR,#tab_x      ;adres tablicy - dane
          CLR A                ;adres - kod
          MOV B,A              ;  i do B
loop:     PUSH DPH             ;zapamiętanie adresu
          PUSH DPL             ;  dla danych
          MOV DPTR,#tab_c      ;adres tablicy - kod
          MOVC A,@A+DPTR       ;bajt w ACC
          JZ stop              ;jesli koniec napisu
          POP DPL              ;odtworzenie adresu
          POP DPH              ;  dla danych
```

```
          MOVX @DPTR,A          ;bajt do pam. zewn.
          INC B                 ;nast. adres - kod
          MOV A,B               ;  i do ACC
          INC DPTR              ;nast. adres - pam. zewn.
          DJNZ R0,loop
stop:     SJMP $
          ORG 30h
tab_c:    DB 'Ala ma kota'
          DB 0                  ;koniec napisu
```

5 Operacje arytmetyczne i logiczne

5.1 Arytmetyka

Dodawanie liczb x i y w systemie dwójkowym:
$x = x_7x_6x_5x_4x_3x_2x_1x_0$,
$y = y_7y_6y_5y_4y_3y_2y_1y_0$.

```
CY=0;
for i:=0 to 7 do
begin
     z[i]:=(x[i]+y[i]+CY) mod 2;
     CY:=(x[i]+y[i]+CY) div 2;
end;
```

Odejmowanie liczb x i y w systemie dwójkowym:

Niech $\bar{y} = \bar{y}_7\bar{y}_6\bar{y}_5\bar{y}_4\bar{y}_3\bar{y}_2\bar{y}_1\bar{y}_0$

Wtedy $x - y = x + (\bar{y} + 1)$

Przykład.

Wykonujemy $3 - 5 = -2$.

$y = +5$	00000101
$\bar{y}$	11111010
$+1$	1
$-y = -5$	11111011

Teraz dodajemy $3 + (-5)$

$+3$	00000011
-5	11111011
$=$	11111110

To ma być -2, czyli

$z = +2$	00000010
$\bar{z}$	11111101
$+1$	1
$-z = -2$	11111110

Jest to kod uzupełnień do dwóch.

Można w nim reprezentować liczby 0d -128 d0 127.

Uwaga. Taki właśnie jest zasięg skoków krótkich.

Kod BDC (binary decimal code): każdy bajt reprezentuje dwie cyfry dziesiętne, każdą przez połówkę bajtu.

Przykład. Liczba 29 w kodzie BDC jest równa $\underbrace{0010}_{2}\ \underbrace{1001}_{9}$, a liczba 78 jest równa $\underbrace{0111}_{7}\ \underbrace{1000}_{8}$

5.2 Rozkazy arytmetyczne

- Dodawanie: ADD dodaje do akumulatora drugi argument,
- Dodawanie z przeniesieniem ADDC dodaje do akumulatora drugi argument i dodaje bit CY,
- Odejmowanie (z przeniesieniem) SUBB odejmuje od akumulatora drugi argument i odejmuje bit CY, wynik w kodzie uzupełnień do dwóch.

Pierwszym argumentem jest zawsze A, drugim może być adres bezpośredni, rejestr, adres pośredni lub liczba.
Wynik jest zawsze umieszczony w akumulatorze. Jeśli wynik jest większy od FFh, to ustawiane są bity CY, AC i OV.

Rozkaz INC zwiększa o 1, a rozkaz DEC zmniejsza o 1 argument, którym może być akumulator A, adres bezpośredni, rejestr lub adres pośredni. Nie zmieniają się znaczniki.
Argumentem rozkazu INC może być też DPTR.

Mnożenie: MUL AB. Wynikiem jest liczba 16 bitowa, starszy bajt jest umieszczony w B, młodszy w A. Jeśli wynik jest większy od FFh, to bit OV jest ustawiany, a CY zerowany.
Dzielenie (całkowitoliczbowe) DIV AB. Liczbę zawartą w A dzielimy przez liczbę zawartą w B. Część całkowita wyniku jest wpisywana do A, reszta z dzielenia do B.

Korekcja dziesiętna DA A daje w połączenie z rozkazami ADD lub ADDC poprawny wynik , gdy dodajemy liczby w kodzie BDC. Ustawia bit CY, gdy wynik jest większy od 99.

Przykład. Dodając $29 + 78 = 107$ w kodzie BDC otrzymujemy :

$$\begin{array}{ll} 29 & 0010\,1001 \\ 78 & 0111\,1000 \\ \hline 107 & 1010\,0001 \end{array}$$

Powinno zaś być $\underbrace{0000}_{0}\ \underbrace{0111}_{7}$ i ustawiony znacznik przeniesienia, bit CY.

5.3 Rozkazy logiczne na bajtach

Rozkazy ANL, ORL, XRL są odpowiednio koniunkcją, alternatywą i alternatywą wykluczającą bitów argumentów. Pierwszym argumentem może być A lub adres bezpośredni.

- Jeżeli pierwszym argumentem jest A, to drugim może być adres bezpośredni, rejestr, adres pośredni lub liczba.
- Jeżeli pierwszym argumentem jest adres bezpośredni, to drugim jest albo A albo liczba.

Rozkaz CLR A powoduje wyzerowanie akumulatora. Daje ten sam efekt, co MOV A,#0 i wykonywany jest w tym samym czasie – jeden cykl rozkazowy.
Rozkaz CPL A wpisuje do A negację (bit po bicie) zawartości A.
Rozkaz SWAP A zamienia w akumulatorze starszą i młodszą połówkę bajtu.

Rozkazy przesunięć RL, RLC, RR, RRC jako argument zawsze mają A. Oznaczmy $A = a_7a_6a_5a_4a_3a_2a_1a_0$.

- RL A: $a_0 = a_7$, $a_{i+1} = a_i$ dla $i = 0, 1, \ldots, 6$,
- RLC A: $a_0 = CY$, $a_{i+1} = a_i$ dla $i = 0, 1, \ldots, 6$, $CY = a_7$,
- RR A: $a_7 = a_0$, $a_{i-1} = a_i$ dla $i = 1, 2, \ldots, 7$,
- RRC A: $a_7 = CY$, $a_{i-1} = a_i$ dla $i = 1, 2, \ldots, 7$, $CY = a_0$.

5.4 Rozkazy na bitach

Rozkaz CLR zeruje, a rozkaz SETB ustawia bit. Argumentem jest adres bezpośredni bitu lub bit C.
Rozkaz CPL daje negację bitu. Argumentem jest adres bezpośredni bitu lub bit C.
Rozkazy ANL i ORL dają koniunkcję lub alternatywę bitów. Pierwszym argumentem jest C, a drugim adres bezpośredni bitu.
Rozkaz MOV nadaje pierwszemu argumentowi wartość drugiego argumentu. Jednym argumentem jest C, drugim adres bezpośredni bitu.

Uwaga.
Bity można adresować przez podanie nazwy bajtu w SFR i numeru bitu w postaci
nazwa.numer
np. ACC.3 adresuje bit o numerze 4 w akumulatorze, P0.1 adresuje bit o numerze 1 portu P0.
Można też bit adresować przez nazwę bitu.
Przykład.
Słowo stanu programu PSW.

CY=PSW.7 ustawiany lub zerowany sprzętowo podczas wykonywania operacji arytmetycznych, sygnalizuje przeniesienie lub pożyczkę z bitu 7. Pełni rolę akumulatora boolowskiego. Dostępny też przez

nazwę C.

AC=PSW.6 znacznik przeniesienia pomocniczego przy używaniu zapisu BDC.

F0=PSW.5 bit do wszystkiego, zmieniany programowo.

RS1=PSW.4, RS0=PSW.3 liczba RS1 RS0 wskazuje na zestaw rejestrów.

OV=PSW.2 znacznik nadmiaru ustawiany lub zerowany sprzętowo, wskazuje na przekroczenie zakresu liczb w kodzie U2.

P=PSW.0 znacznik parzystości ustawiany lub zerowany sprzętowo, (P=1 – parzysta, P=0 – nieparzysta liczba jedynek w A.

6 Model procesora w C++

6.1 Kody rozkazów

Kod rozkazu zajmuje jeden bajt, argumenty jeśli występują, zajmują jeden lub dwa kolejne bajty. Czasem argument zajmuje część bajtu kodu rozkazu.

Przykłady kodów

MOV A,Rr – 1110 $1r_2r_1r_0$, gdzie $r_2r_1r_0$ jest numerem rejestru zbioru rejestrów określonym przez RS1 RS0.

Liczba cykli: 1

MOV Rr,A – $1111\,1r_2r_1r_0$, gdzie $r_2r_1r_0$ jest numerem rejestru zbioru rejestrów określonym przez RS1 RS0.
Liczba cykli: 1
MOV A,@Ri – $1110\,011i$, gdzie $i = 0$ lub $i = 1$ jest numerem rejestru zbioru rejestrów określonym przez RS1 RS0.
Liczba cykli: 1
MOV @Ri,A – $1111\,011i$, gdzie $i = 0$ lub $i = 1$ jest numerem rejestru zbioru rejestrów określonym przez RS1 RS0.
Liczba cykli: 1

CLR A – 1110 0100, zeruje akumulator.
Liczba cykli: 1
CLR C – 1100 0011, zeruje znacznik CY
Liczba cykli: 1
CLR bit – 1100 0010 $b_7 \ldots b_0$, zeruje bit o adresie $b_7 \ldots b_0$.
Liczba cykli: 1
MOV Rr,#n – 0111 $1r_2r_1r_0$ n, gdzie $r_2r_1r_0$ jest numerem rejestru zbioru rejestrów określonym przez RS1 RS0, a n jest liczbą (argumentem bezpośrednim).
Liczba cykli: 1

MOV ad,#n – 0111 0101 $a_7 \ldots a_0$ n, gdzie $a_7 \ldots a_0$ jest adresem, a n jest liczbą (argumentem bezpośrednim).
Liczba cykli: 2
RET – 0010 0010, adres powrotu jest wpisywany do PC, najpierw bardziej, potem mniej znaczący bajt. SP jest zmniejszany o 2.
Liczba cykli: 2

6.2 Procesor jako klasa w C++

Pamięć, to tablice. SFR jest przesunięta o 80h bajtów.
Realizacja rozkazów o tych samych nazwach, a odnosząca się do różnych danych jest zrealizowana przez przeciążanie funkcji.
Klasę p51 można dalej dziedziczyć, np. przez klasę p52 o rozszerzonej pamięci wewnętrznej lub inną klasę o np. większej liczbie portów.

```
#include <iostream.h>
#include <fstream.h>

enum Acc {A=0xE0,DPTR=0x82}; // DPL=82, DPH=83
enum REG {R0=0,R1=1,R2=2,R3=3,R4=4,R5=5,R6=6,R7=7};
enum AT_REG {atR0=0,atR1=1};

char LO(unsigned int nn);
char HI(unsigned int nn);

class p51;
```

```
char int_memo[0x7F]; // 128 B
char SFR[0x7F];      // 128 B
int tab_label[1000];


char LO(unsigned int nn)
{
     return nn%0x100;
};
```

```
char HI(unsigned int nn)
{
     return nn/0x100;
};

     const char ACC=0xE0;
     const char B=0xF0;
     const char SP=0x81;
     const char PSW=0xD0;
     const char DPL=0x82;
     const char DPH=0x83;
```

```
class p51
{
     public:
     char *code;         // kod
     char *ext;          // pamięc zewnętrzna
     int PC;

     // konstruktory i destruktor
     p51(unsigned,unsigned); // rozmiary pamięci kodu i zewn.
     p51(unsigned);          // rozmiary pamięci zewn., kod 4kB
     p51();                  // kod 4kB, bez pamięci zewn.
```

```
~p51();

void begin();
void end();
void nop();
void mov(Acc A, char n);        // MOV A,#n
void mov(Acc AD, int n);        // MOV A,n
void mov(Acc A, REG r);         // MOV A,Rr - r=0,1,...,7
void mov(Acc A, AT_REG i);      // MOV A,@Ri - i=0,1
void mov(REG r, Acc A);         // MOV Rr,A
void mov(REG r, char n);        // MOV Rr,#n
void mov(REG r, int n);         // MOV Rr,n
```

```
void mov(AT_REG i, char n);     // MOV @Ri,#n
void mov(AT_REG i, Acc A);      // MOV @Ri,A
void mov(AT_REG i, int n);      // MOV @Ri,n
void mov(int n, Acc A);         // MOV n,A
void mov(int n, REG r);         // MOV n,Rr
void mov(int n, int m);         // MOV n,m
void mov(int n, AT_REG i);      // MOV n,@Ri
void mov(int n, char m);        // MOV n,#m
void label(int n);              // etykieta n:
void ljmp(int n);               // LJMP n
void sjmp(int n);               // SJMP n
void c(unsigned i);
```

```
};

p51::p51(unsigned cd, unsigned mext)
{
     unsigned int i;
     code=new char[cd];
     ext=new char[mext];
     for(i=0;i<cd;i++){code[i]=char(0);};
     for(i=0;i<mext;i++){ext[i]=char(0);};
     for(i=0;i<0xEF;i++){SFR[i-0x80]=char(0);};
     PC=char(0);
     SFR[SP-0x80]=char(7);
```

```
     SFR[PSW-0x80]=char(0);
};

p51::p51(unsigned mext)
{
     unsigned int i;
     code=new char[0xFFF];
     ext=new char[mext];
     for(i=0;i<0xFFF;i++){code[i]=char(0);};
     for(i=0;i<mext;i++){ext[i]=char(0);};
     for(i=0;i<0xEF;i++){SFR[i-0x80]=char(0);};
     PC=char(0);
```

```
     SFR[SP-0x80]=char(7);
     SFR[PSW-0x80]=char(0);
};

p51::p51()
{
     unsigned int i;
     code=new char[0xFFF];
     for(i=0;i<0xFFF;i++){code[i]=char(0);};
     for(i=0;i<0xEF;i++){SFR[i-0x80]=char(0);};
     PC=char(0);
     SFR[SP-0x80]=char(7);
```

```
    SFR[PSW-0x80]=char(0);
};

p51::~p51()
{
        int i;
        ofstream cd;
        cd.open("code.bin");
        cd.setf(ios::hex);
        cd.unsetf(ios::dec);
        for(i=0;i<PC;i++)
        {cd<<(int)(code[i]&0x00FF)<<' ';};
```

```
        cd.close();
        delete[] code;
        delete[] ext;
}

void p51::begin()
{
        PC=0;
}

void p51::end(){};
```

```
void p51::nop()
{
        code[PC++]=(char)0;
}

void p51::mov(Acc A, char n)
{
     code[PC++]=(char)0x74;
     code[PC++]=n;
     SFR[A-0x80]=n;
};
```

```
void p51::mov(Acc AD, int n)
{
     if(AD==A){
     code[PC++]=(char)0xE5;
     code[PC++]=(char)n;
     SFR[A-0x80]=int_memo[n];
     }
     if(AD==DPTR)
     {
     code[PC++]=(char)0x90;
     code[PC++]=(SFR[DPTR-0x80]=LO(n));
```

```
     code[PC++]=(SFR[DPTR+1-0x80]=HI(n));
     }
};

void p51::mov(Acc A, REG r)
{
     code[PC++]=(char)(0xE8+r);
     SFR[A-0x80]=int_memo[((SFR[PSW-0x80]&0x18)>>3)*8+r];
}

void p51::mov(Acc A, AT_REG i)
{
```

```
     code[PC++]=(char)(0xE6+i);
     SFR[A-0x80]
        =int_memo[int_memo[((SFR[PSW-0x80]&0x18)>>3)*8+i]];
};

void p51::mov(REG r, Acc A)
{
     code[PC++]=(char)(0xF8+r);
     int_memo[((SFR[PSW-0x80]&0x18)>>3)*8+r]=SFR[A-0x80];
}

void p51::mov(REG r, char n)
```

```
{
     code[PC++]=(char)(0x78+r);
     code[PC++]=n;
     int_memo[((SFR[PSW-0x80]&0x18)>>3)*8+r]=n;
}

void p51::mov(REG r, int n)
{
     code[PC++]=(char)(0xA8+r);
     code[PC++]=(char)n;
     int_memo[((SFR[PSW-0x80]&0x18)>>3)*8+r]=(char)n;
}
```

```
void p51::mov(AT_REG i, char n)
{
     code[PC++]=(char)(0x76+i);
     code[PC++]=n;
     int_memo[int_memo[((SFR[PSW-0x80]&0x18)>>3)*8+i]]=n;
}

void p51::mov(AT_REG i, Acc A)
{
     code[PC++]=(char)(0xF6+i);
     int_memo[int_memo[((SFR[PSW-0x80]&0x18)>>3)*8+i]]
```

```
        =SFR[A-0x80];
}

void p51::mov(AT_REG i, int n)
{
     code[PC++]=(char)(0xA6+i);
     code[PC++]=(char)n;
     int_memo[int_memo[((SFR[PSW-0x80]&0x18)>>3)*8+i]]=n;
}

void p51::mov(int n, Acc A)
{
```

```
    code[PC++]=(char)0xF5;
    code[PC++]=(char)n;
    int_memo[n]=SFR[A-0x80];
};


void p51::mov(int n, REG r)
{
    code[PC++]=(char)(0x88+r);
    code[PC++]=(char)n;
    SFR[n-0x80]
       =int_memo[((SFR[PSW-0x80]&0x18)>>3)*8+r];
```

```
}

void p51::mov(int n, int m)
{
     code[PC++]=(char)0x85;
     code[PC++]=(char)n;
     code[PC++]=(char)m;
     int_memo[SFR[n-0x80]]=(char)m;
}

void p51::mov(int n, AT_REG i)
{
```

```
    code[PC++]=(char)(0x86+i);
    code[PC++]=(char)n;
    SFR[n-0x80]
       =int_memo[int_memo[((SFR[PSW-0x80]&0x18)>>3)*8+i]];
}

void p51::mov(int n, char m)
{
    code[PC++]=(char)0x75;
    code[PC++]=(char)n;
    code[PC++]=m;
    int_memo[SFR[n-0x80]]=m;
```

```
}

void p51::label(int n)
{
        tab_label[n]=PC;
}

void p51::ljmp(int n)
{
        code[PC++]=(char)0x02;
        code[PC++]=HI(tab_label[n]);
        code[PC++]=LO(tab_label[n]);
```

```
}

void p51::sjmp(int n)
{
        char d;
        int m;
        m=tab_label[n]-PC-2;
        d=(char)m;
        code[PC++]=(char)0x80;
        code[PC++]=d;
}
```

```
void p51::c(unsigned i)
{
     unsigned short int x;
     x=code[i]&0x00FF;
     cout<<(int)x<<' ';
};
```

```
main()
{
     cout.setf(ios::hex);
     cout.unsetf(ios::dec);

     int i;
//     p51 p(0xFFF,0x08FF);
     p51 p(0x08FF);
//     p51 p;
     for(i=0;i<2;i++)
     {
     p.begin();
```

```
p.label(1);
p.mov(A,R0);        // 0
p.mov(A,1);         // 1
p.mov(A,atR0);      // 3
p.mov(A,char(2));   // 4
p.label(2);         // 6
p.mov(R0,A);
p.mov(R0,3);
p.mov(R0,char(4));
p.mov(5,A);
p.mov(6,R0);
p.mov(7,8);
```

```
p.mov(9,atR0);
p.mov(10,char(11));
p.mov(atR0,A);
p.mov(atR0,12);
p.mov(atR0,char(13));
p.mov(DPTR,0x0F0E);
p.ljmp(2);
p.sjmp(4); // skok o 1 do przodu
p.label(3);
p.nop();
p.label(4);
p.sjmp(3); // skok o 3 do tyłu
```

```
    p.nop();
    p.end();
    }
    for(i=0;i<p.PC;i++){p.c(i);};
    cout<<endl;

};
```

Kod stworzony przez ten program:

```
e8
e5 01
e6
74 02
f8
a8 03
78 04
f5 05
88 06
85 07 08
86 09
```

75 0a 0b
f6
a6 0c
76 0d
90 0e 0f
02 00 06
80 01
00
80 fd
00

Skok do tyłu o 3:

$y = \text{FDh}$	11111101
$\bar{y}$	00000010
$+1$	1
$-y = -3$	-00000011

7 Przerwania, zegary i liczniki

7.1 Przerwania

Odebranie przez procesor zgłoszenia przerwania powoduje zawieszenie wykonywanego programu, zapisanie na stos adresu powrotu (aktualnego stanu PC) i wykonanie skoku do podprogramu. Podprogram musi kończyć się rozkazem RETI. Wykonanie takiego podprogramu nazywa się obsługą przerwania.
Przerwania mogą mieć różne priorytety. Jeżeli w czasie obsługi przerwania procesor odbierze zgłoszenie przerwania o wyższym priorytecie, to przerywa obsługę przerwania o niższym priorytecie i kończy ją po zakończeniu przerwania o wyższym priorytecie.

Zgłoszone przerwanie o niższym lub równym priorytecie jest obsługiwanie po zakończeniu bieżącej obsługi przerwania.
Procesor 8051 może przyjąć zgłoszenie z 5 źródeł:

- z wejścia INT0,
- z wejścia INT1,
- z przepełnienia licznika T0,
- z przepełnienia licznika T1,
- z portu szeregowego – koniec nadawania lub odbierania znaku (dwa osobne znaczniki).

Podprogram obsługi przerwania ma ustalony sprzętowo dla niego adres.

Przerwanie z danego źródła powoduje ustawienie na 1 odpowiedniego znacznika (bitu). Bity te można też ustawić programowo przez SETB. Przyjęcie zgłoszenia zeruje znaczniki, z wyjątkiem przychodzących z portu szeregowego. Wszystkie bity można zerować programowo przez CLR.
System przerwań można włączyć i wyłączyć programowo, a także każde zgłoszenie przerwania może być indywidualnie zamaskowane przez ustawienie odpowiedniego bitu w bajcie IE (interrupt enable).
Jeśli EA=1, to zgłoszenia przerwań są przyjmowane, a jeśli EA=0, to nie są.
Priorytety można zmienić bitami w bajcie IP (interrupt priority)

Znaczniki (bity) w bajtach EA i IP i znaczniki zgłoszenia przerwań (Zn) w bajtach TCON i SCON, adresy podprogramów obsługi przerwań.

IE	IP	Zn	Adres	Przerwanie	Priorytet
EX0	PX0	IT0	0003h	zewnętrzne INT0	najwyższy
ET0	PT0	TF0	000Bh	od licznika/ zegara T0	
EX1	PX1	IT0	0013h	zewnętrzne INT1	
ET1	PT1	TF1	001Bh	od licznika/ zegara T1	
ES	PS	TI/RI	0023h	od portu szeregowego	najniższy
EA				system przerwań	

Bity IE0 i IE1 sterują sposobem przyjęcia przerwania:

- IEi=0 – zgłoszenie niskim poziomem sygnału na wejściu
- IEi=1 – zgłoszenie opadającym zboczem sygnału.

Wejściem jest INT0=P3.2 lub INT1=P3.3

Uwaga. Różnica między rozkazami RET i RETI.

Rozkaz RETI kończy obsługę przerwania, a rozkaz RET nie kończy. Do momentu wystąpienia RETI nie będzie przyjęte żadne zgłoszenie przerwania o niższym lub równym priorytecie.

Ponieważ na podprogram obsługi przerwania procesor rezerwuje zaledwie 8 bajtów, a obszar pamięci programu zarezerwowany na te podprogramy nie powinien być wykorzystywany do innych celów, to typowy program ma strukturę:

```
LJMP start      ;skok do właściwego programu
ORG 0003h
LJMP int0proc   ;skok do obsługi INT0
ORG 000Bh
LJMP t0proc     ;skok do obsługi T0
ORG 0013h
LJMP int1proc   ;skok do obsługi INT1
ORG 001Bh
LJMP t1proc     ;skok do obsługi T1
ORG 0023h
LJMP rs_proc    ;skok do obsługi RS
```

```
        ORG 0030h
;procedury obsługi przerwań
int0proc:       ;obsługa przerwania od INT0
        RETI
t0proc:         ;obsługa przerwania od T0
        RETI
int1proc:       ;obsługa przerwania od INT1
        RETI
t1proc:         ;obsługa przerwania od T1
        RETI
rs_proc:        ;obsługa przerwania od RS
        RETI
```

```
start:
;tu zaczyna się właściwy program
        SETB EA  ;zezwolenie na przyjmowanie przerwań
        SETB EX0 ;odblokowanie przerwań z INT0
wait:   SJMP $
;czekamy na przerwanie z INT0, obsługujemy je
;i wracamy do linii z etykietą wait
```

7.2 Zegary i liczniki

Do mierzenia czasu lub zliczania impulsów wykorzystywane są dwa 16-bitowe liczniki T0 i T1. Każdy z nich składa się z dwóch bajtów THi i TLi, które mogą być wykorzystywane również niezależnie. Przepełnienie się licznika powoduje ustawienie znacznika TFi (o ile EA=1 oraz ETi=1). Sposób działania licznika jest określony przez ustawienie bitów w słowach TMOD i TCON.

TMOD:

$$\underbrace{\text{GATE C/T M1 M0}}_{\text{T1}}\ \underbrace{\text{GATE C/T M1 M0}}_{\text{T0}}$$

TCON:

TF1 TR1 TF0 TR0 IE1 IT1 IE0 IT0

- M1 M0 – tryb pracy od 0 do 3,
- C/T=0 – funkcja zegara, C/T=1 – funkcja licznika.
- Przyjmiemy, że GATE=0 (tak jest przy inicjacji systemu)
- TRi – uruchomienie licznika Ti

Omówione zostaną tylko tryby 1 i 2.

W trybie 1 odpowiedni licznik 16-bitowy Ti jest zwiększany o 1 przy wystąpieniu impulsu na wejściu T0=P3.4 lub T1=P3.5 (dla C/T=1) lub w każdym cyklu (dla C/T=0). Przy częstotliwości zegara 12 MHz, jest to co 1 μs. Przy przepełnieniu, tzn. przy przejściu z FFFFh na 0, generowane jest przerwanie. Impulsem jest zmiana poziomu sygnału na wejściu T0 lub T1 próbkowana w dwóch kolejnych cyklach.

Do właściwego odmierzania czasu trzeba ustawić odpowiednią wartość na początku, a potem w procedurze obsługi przerwania doładować licznik.
Przykład. Mamy odmierzać czas 40 ms w systemie z zegarem 12 MHz. Wartością początkową musi być 64C0h = FFFFh − 40000.
Ponieważ od chwili przepełnienia licznika do przyjęcia przerwania może upłynąć nieznany okres czasu, to licznik trzeba doładować w locie. Zakładamy, że ten okres jest mniejszy od 64. Wtedy doładowanie ma postać:

```
ORL TL0,#0C0h   ;dodanie do mniej znaczącego bajtu
MOV TH0,#64h    ;wpisanie bardziej znaczącego bajtu
```

W trybie 2 licznik Ti pracuje jako 8-bitowy (TLi) z automatycznym przepisaniem (reload) wartości początkowej z THi w chwili przepłnienia licznika. Maksymalny okres przerwań wynosi 256 cykli, czyli 256 μs przy zegarze 12 MHz.
Pozostałe reguły są takie jak w trybie 1.
Przykład. Zegar 11.0592 MHz. Jeden cykl trwa $t_c = 12/11.0592\,\mu$s. Jeśli wpiszemy do THi wartość 244, to przepełnienie nastąpi co

$$t_c \cdot 256 \cdot (256 - 244)\,\mu\text{s} = 10/3\,\text{ms}.$$

```
;Stałe przy kwarcu 11.0592MHz
;Stała do odmierzania 10/3 ms
set11     EQU 244   ;256-244=12
clk1      EQU 3     ;3 tyknięcia na 10ms
```

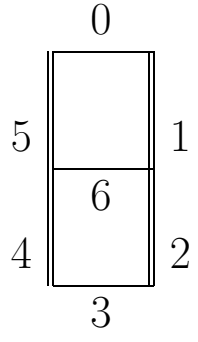
```
clk5      EQU 15    ;15 tyknięć na 50ms
fblink    EQU 20    ;co 0.2 sek.
```

Doładowanie ogranicza się do

```
        ;mniej znaczący bajt TL0 jest zerem
        ;w momencie przepełnienia
        MOV TH0,#set11    ;wpisanie bardziej znaczącego bajtu
```

8 Programowanie 8051

8.1 Tablice



Wyświetlacz siedmiosegmentowy.

Przykład. Cyfra 0, to 00111111 = 3Fh. W wyświetlaczu aktywny jest poziom niski, czyli na wyświetlacz wysłać należy $\overline{00111111} = 11000000$.

```
dot       EQU 080h   ;kropka dziesiętna
blank     EQU 000h   ;znak _ na wyświetlaczu 7-segmentowym
A_char    EQU 077h   ;litera A na wyświetlaczu 7-segmentowym
C_char    EQU 039h   ;litera C na wyświetlaczu 7-segmentowym
E_char    EQU 079h   ;litera E na wyświetlaczu 7-segmentowym
L_char    EQU 038h   ;litera L na wyświetlaczu 7-segmentowym
P_char    EQU 073h   ;litera P na wyświetlaczu 7-segmentowym
U_char    EQU 03Eh   ;litera U na wyświetlaczu 7-segmentowym
```

```
bl_cod    EQU 21          ;kod znaku pustego
A_cod     EQU 22          ;kod litery A
C_cod     EQU 23          ;kod litery C
E_cod     EQU 24          ;kod litery E
L_cod     EQU 25          ;kod litery L
P_cod     EQU 26          ;kod litery P
U_cod     EQU 27          ;kod litery U
;-------------------------------------------------------------

;Procedura zamiany cyfry na kod 7-segmentowy lub kropkę.
;Wejście: cyfra w ACC, (kropka=20),
;Wyjście: kod w ACC.
```

```
get_cod7: INC A           ;pomijamy RET
          INC A           ;pomijamy CPL A
          MOVC A,@A+PC    ;pobieramy kod
          CPL A           ;świecą się zera, a nie jedynki!
          RET
cod_7seg:
          DB 03Fh   ;0
          DB 006h   ;1
          DB 05Bh   ;2
          DB 04Fh   ;3
          DB 066h   ;4
```

```
DB 06Dh   ;5
DB 07Dh   ;6
DB 007h   ;7
DB 07Fh   ;8
DB 06Fh   ;9
DB 0BFh   ;.0
DB 086h   ;.1
DB 0DBh   ;.2
DB 0CFh   ;.3
DB 0E6h   ;.4
DB 0EDh   ;.5
DB 0FDh   ;.6
```

```
DB 087h   ;.7
DB 0FFh   ;.8
DB 0EFh   ;.9
DB dot    ;.
DB E_char ;litera E
DB blank  ;znak pusty
DB C_char ;litera C
DB P_char ;litera P
DB L_char ;litera L
DB U_char ;litera U
DB A_char ;litera A
```

Wysłanie znaku na wyświetlacz.

Wyświetlacz ma adres postaci 8000h + adres. Załóżmy, że adres wyświetlacza ma zakres od 0 do 11, (wyświetlacz 12-pozycyjny) i znajduje się w R0. Kod znaku znajduje się w @R1.

```
PUSH DPH
PUSH DPL
PUSH ACC
MOV DPH,80h
MOV DPL,R0
MOV A,@R1
MOVX @DPTR,A
POP ACC
POP DPL
POP DPH
```

Przykład.
Zakładamy, że w mniej znaczącym półbajcie bajtu o adresie 21h w pamięci wewnętrznej (obszar adresowany bitowo) znajduje się dokładnie jeden ustawiony bit (wiersz w tablicy klawiszy 4×4). W zależności od niego mamy wykonać program obsługi w jednej z czterech wersji.

```
lines   EQU 21h ;obszar od 20h do 2Fh adresowany bitowo
;------------------------
lkey:   ;podprogram obsługi wierszy klawiatury;
        JB lines.0,l0   ;bit 8
        JB lines.1,l1   ;    9
        JB lines.2,l2   ;   10
        JB lines.3,l3   ;   11
```

```
l0:     ;skok do obsługi wiersza nr 0
        LJMP ll0
l1:     ;skok do obsługi wiersza nr 1
        LJMP ll1
l2:     ;skok do obsługi wiersza nr 2
        LJMP ll2
l3:     ;skok do obsługi wiersza nr 3
        LJMP ll3
;-------------------------

;tutaj właściwa obsługa wierszy
```

```
ll0:    ;obsługa wiersza nr 0
        RET
ll1:    ;obsługa wiersza nr 1
        RET
ll2:    ;obsługa wiersza nr 2
        RET
ll3:    ;obsługa wiersza nr 3
        RET
```

Przykład.
Zakładamy, że w mniej znaczącym półbajcie bajtu o adresie 31h w pamięci wewnętrznej znajduje się numer wiersza w tablicy klawiszy 4×4). W zależności od niego mamy wykonać program obsługi w jednej z czterech wersji.

```
lines   EQU 31h
;---------------
lkey:   ;podprogram obsługi wierszy klawiatury;
        MOV DPTR,#l_proc
        MOV A,lines

        ;teraz mnożymy przez 4:
```

```
        RL A    ;x2
        RL A    ;x2, razem x4
        JMP @A+DPTR
l_proc: LJMP l0         ;3 bajty
        NOP             ;1 bajt
        LJMP l1
        NOP
        LJMP l2
        NOP
        LJMP l3
        NOP
```

```
l0:     ;obsługa wiersza nr 0
        RET
l1:     ;obsługa wiersza nr 1
        RET
l2:     ;obsługa wiersza nr 2
        RET
l3:     ;obsługa wiersza nr 3
        RET
```

Przykład.
Restart systemy po naciśnięciu klawisza STOP, podłączonego do przerwania INT0.

```
add_proc  EQU 1 ;do asemblacji warunkowej

       LJMP start
;OBSŁUGA PRZERWAŃ
       ORG 0003h      ;INT0
       LJMP halt
;--------------------------------
      ORG 0030h
```

```
halt: ;powrót do start
      MOV IE,#0
      MOV TMOD,#0
      MOV TCON,#0
      POP ACC  ;zdjęcie niepotrzebnego adresu powrotu
      POP ACC
      MOV DPTR,#haltproc
      PUSH DPL
      PUSH DPH
      RETI     ;"powrót" pod nowy adres
;--------------------
```

```
haltproc: ;ew. dodatkowa obsługa stopu
    IF add_proc
       ;tylko w pewnych wersjach
    ENDIF
       ;coś jeszcze robimy zawsze
      LJMP start
;--------------------
      ;inne procedury obsługi przerwań
;--------------------
start:    ;inicjacja systemu
          ;dalszy ciąg programu
```

Przykład.
Odczytywanie bitów z portu P.1 (przycisków 0 – 7) i zapalanie ledów podłączonych pod adres 1000h, bez ich gaszenia. Gdy wszystkie są zapalone, to wszystkie się gasi. Zakładamy, że stan ledów jest zapisany w R2.

```
        MOV P1,#0FFh      ;aby można było czytać
        MOV A,P1
        ORL A,R2
        CPL A
        JNZ nzero
        SJMP next
nzero:  CPL A
```

```
next:   MOV DPTR,#1000h
        MOVX @DPTR,A
```

Przykład.

Program działa z dużą czestotliwością, większą od częstotliwości odświeżania wyświetlacza (przetwornik 1kHz). Czas obsługi przetwornika jest krytyczny.

```
clk1      BIT 0     ;ustawiany co 1 ms.
clk5      BIT 1     ;ustawiany co 5 ms.
count     EQU 30h   ;licznik do 5

display   MACRO   ;wyświetla co 1/50 sek. cyfry
                ;na kolejnych 4 wyświetlaczach
          ENDM
```

```
          LJMP start
          ORG 000Bh
          LJMP clk_step

          ORG 0030h
clk_step: SETB clk1
          DJNZ count,loop
          MOV count,#5
          SETB clk5
loop:     RETI
```

```
start:    ;inicjacja zegarów itp.
          MOV count,#5

main:     JNB clk5,next1
          CLR clk5
          display
next1:    JNB clk1,$   ;czekamy na następny krok
          ;tu obsługujemy przetwornik 1KHz
          LJMP main
```

Przykład.
Program obsługuje wolnodziałające urządzenie mechaniczne.
Dokładność czasowa jest rzędu 1/10 sekundy.

```
clk5      BIT 1     ;ustawiany co 5 ms.

display   MACRO   ;wyświetla co 1/50 sek. cyfry
                ;na kolejnych 4 wyświetlaczach
          ENDM

          LJMP start
          ORG 000Bh
          LJMP clk_step
```

```
          ORG 0030h
clk_step: display
loop:     RETI

start:    ;inicjacja zegarów itp.

main:     ;tu główny, wolnodziałający program
          LJMP main
```

9 Procesory rodziny 80x86

9.1 Procesor i jego rejestry

Procesor Intel 8086 posiada 16-bitowe rejestry.
Rejestry ogólnego przeznaczenia:

- AX – do operacji arytmetycznych i logicznych,
- BX – rejestr bazowy, adresowanie pamięci,
- CX – licznik,
- DX – rejestr danych, mnożenie i dzielenie, wysyłanie i odbieranie danych z portów.

- SI – rejestr indeksujący pamięć, wskazuje obszar skąd przesyła się dane,
- DI – rejestr indeksujący pamięć, wskazuje obszar dokąd przesyła się dane,
- SP – wskaźnik stosu,
- BP – rejestr do adresowania pamięci.

Rejestry te można traktować jako dwa rejestry 8-bitowe: AX=AH+AL, BX=BH+BL, CX=CH+CL, DX=DH+DL.
Rejestr IP (wskaźnik rozkazu) zawiera numer w pamięci aktualnie wykonywanego rozkazu. Nie jest dostępny dla programisty.

Adresowanie pamięci (wspólna pamięć programu i danych) wg wzoru:

$$\text{adres} = 16 \cdot \text{segment} + \text{offset}$$

Adres zapisywany jest w postaci segment:offset, np. 1000:7C00 i 1700:0C00 oznaczają ten sam adres 17C00. Oznacza to, że adresy są 20-bitowe, czyli adresować można do 1 MB.
Rejestry segmentowe:

- CS – segment aktualnie wykonywanego rozkazu,
- DS – segment z danymi,
- ES – segment dodatkowy np. przy przesyłaniu łańcuchów
- SS –segment stosu.

Rejestry te są używane niejawnie, np. rozkaz skoku pod adres 0 jest skokiem pod adres CS:0.
Rejestr znaczników (flag):
– – – – O D I T S Z – A – P – C

- O – nadmiar,
- D – kierunek przesyłu danych,
- I – zezwolenie na przerwania,
- T – praca krokowa,
- S – znak wyniku operacji arytmetycznej,
- Z – znacznik zera j.w.,
- A – przeniesienie połówkowe,
- P – parzystość,
- C – przeniesienie

9.2 Asembler – pierwsze programy

Pierwszy program

```
.MODEL tiny ;model
.STACK 100h ;pamięć na stos
.DATA       ;dane
Komunikat db 'Witamy na programowaniu niskopoziomowym',13,10,'$'
.CODE       ;program
start:
mov ax,seg komunikat
mov ds,ax
mov ah,9    ;argument przerwania - numer funkcji
            ;wyświetlenie ciągu znaków
```

```
mov dx, offset komunikat ;j.w.
int 21h     ;przerwanie
mov ah,4Ch  ;argument przerwania - koniec programu
int 21h     ;przerwanie
END start   ;program zaczynamy od start
```

Stos rośnie w kierunku malejących adresów (stos wiszący).
SS wskazuje segment stosu, SP – wierzchołek stosu.

Modele

tiny	program i dane mieszczą się w jednym segmencie 64 KB,
small	program w jednym segmencie, dane w jednym segmencie,
medium	dane w jednym segmencie,
compact	program w jednym segmencie,
large	pojedyncza dana w jednym segmencie,
huge	dowolny rozmiar programu i danych.

9.3 Najważniejsze rozkazy

Dane i adresowanie

Stała, np. 1 (bez #, a więc MOV AX,1 zamiast MOV A,#1 jak 8051).
Dane rejestrowe – zawartość rejestru
Dane pamięciowe (adres pośredni) – rejestr w nawiasie kwadratowym,
np. [BX] – dana pod adresem wskazywanym przez rejestr (rejestry).
Zmienna – numer komórki pamięci dany przez etykietę, np.

```
jeden db 1
dwa   dw 1000
napis db 'napis'
```

Przesłania

MOV (prawie wszystkie kombinacje dwóch argumentów)

XCHG arg1,arg2 (zamiana między rejestrami lub rejestrem i adresem

IN rejestr-a,adres (adres¡100h)

IN rejestr-a,dx (rejestr-a, to AX lub AL)

OUT adres,rejestr-a

OUT adres,dx

Operacje arytmetyczne

ADD rejestr,adres i.in,
SUB rejestr,adres i.in, (bez pożyczki)
ADC rejestr,adres i.in,
SBB rejestr,adres i.in, (z pożyczką) INC rejestr
DEC rejestr
MUL rejestr$I$adres (rejestr-a mnożony przez argument)
IMUL rejestr (j.w. ze znakiem)
DIV rejestr (rejestr-a mnożony przez argument)
IDIV rejestr (j.w. ze znakiem)
NEG rejestr$I$adres (zmiana znaku)

Przesunięcia i obroty

SHL adres$I$rejestr,CLI1 (logiczne)
SAL adres$I$rejestr,CLI1 (arytmetyczne)
SHR adres$I$rejestr,CLI1 (logiczne)
SAR adres$I$rejestr,CLI1 (arytmetyczne)
SHR zeruje najbardziej znaczący bit, SAR – powiela go.
Bit wysunięty poaza bajt do C.
ROL, RCL, ROR, RCR – obroty w lewo i prawo bez przeniesienia lub z przeniesieniem C, argumenty jak poprzednio.

Operacje logiczne

NOT rejestr$I$adres (negacja bit po bicie)

AND rejestr$I$adres,rejestr$I$adres$I$wartość

OR rejestr$I$adres,rejestr$I$adres$I$wartość

XOR rejestr$I$adres,rejestr$I$adres$I$wartość

TEST rejestr$I$adres,rejestr$I$adres$I$wartość

TEST nie zmienia wartości pierwszego argumentu, tylko ustawia znaczniki.

Procedury, stos, przerwania

JMP

CALL

wywołania procedur i skoki bezwarunkowe – zostaną omówione później.

LOOP etykieta

– zmniejsza CX o 1, skacze do etykiety (tak jak SJMP) gdy CX¿0.

LOOPZ – jak LOOP, ale skok gdy CX¿0 oraz Z=1

LOOPNZ – jak LOOP, ale skok gdy CX¿0 oraz Z=0

INT nr

– wywołanie przerwania programowego o numerze nr ($0 < \text{nr} < 255$)

RET (powrót z procedury)
IRET (powrót z przerwania)
PUSH arg
POP arg
Argument arg jest adresem danej dwubajtowej albo jednym z rejestrów AX, BX, CX, DX, BP, SP, SI, DI, CS, DS, SS, ES. Nie jest możliwe ani położenie ani zdjęcie ze stosu danej jednobajtowej.

10 Procesory 80x86 – wybrane zagadnienia

10.1 Przerwania

W segmencie 0 znajduje się tabela skoków, mająca 256 pozycji. Każda pozycja to jeden adres segment:offset, czyli 4 bajty. Razem 1024 bajty. Każdy adres to *wektor przerwań*. W czasie startu komputera BIOS i DOS wpisują tam adresy swoich procedur.
Instrukcja INT wywołuje odpowiednie przerwanie. Jej argumentem jest pozycja w tabeli przerwań – numer przerwania. Wywołanie procedury może być też wywołane automatycznie (tak jak w 8051) tzn. sprzętowo. Procedury obsługi przerwania mogą mieć różne funkcje. Numer fukcji wraz z ewentualnymi argumnetami musi być podany w rejestrze AX: w

AH numer funkcji, w AL numer podfunkcji.

Przerwanie INT 21

Program podobny do poprzedniego, ale w konwencji MASM.
Wykorzystuje funkcję numer 9 – wyprowadzenie na standardowe wyjście łańcucha znaków. Łańcuch kończy sie znakiem '$'.

```
mstos SEGMENT STACK
     DB 100h DUP (?)    ;pamięć na stos
mstos ENDS
dane SEGMENT
kom1 db 'Witamy na programowaniu niskopoziomowym',13,10,'drugi r
dane ENDS
```

```
prog SEGMENT        ;program
     assume CS:prog,DS:dane
pocz PROC
start:
mov ax,dane
mov ds,ax
lea DX,kom1
mov ah,9     ;argument przerwania - numer funkcji
             ;wyświetlenie ciągu znaków
int 21h      ;przerwanie
mov ah,4Ch   ;argument przerwania - koniec programu
int 21h      ;przerwanie
```

```
pocz ENDP
prog ENDS
END start
```

Znowu TASM. Czytanie znaku ze standardowego wejścia z echem na standardowym wyjściu. W tym przypadku wejście=klawiatura, wyjście=ekran.

```
.MODEL tiny ;model
.STACK 100h ;pamięć na stos
.CODE       ;program
start:
mov ah,1    ;argument przerwania - numer funkcji
```

```
            ;czytanie znaku z echem
mov cx,10   ;licznik, 10 znaków
l: int 21h  ;przerwanie, czeka na znak z echem
loop l      ;pętla 10 razy, aż CX=0
mov ah,4Ch  ;argument przerwania - koniec programu
int 21h     ;przerwanie
END start
```

Opisy przerwań i szczegółowo opis funkcji przerwania INT 21 w książce [2].

10.2 Łańcuchy

Przystępny opis w rozdzale 10 książki [1]. Łańcuch – spójny obszar pamięci. Można operować na dwóch łańcuchach jednakowej długości równocześnie.
Założenia:

- łańcuch źródłowy wskazywany jest przez DS:SI,
- łańcuch docelowy wskazywany jest przez ES:DI,
- wspólna długość łańcuchów wskazywana jest przez CX,
- dane pobierane z łańcucha źródłowego lub przekazywane z do łańcucha docelowego muszą przechodzić przez rejestr AX.

Wypełnienie obszaru pamięci. Adres segmentu w ES, przesunięcie (offset) w DI, słowo umieszczone w AX, liczba słów w CX.

Przykład

Zakładamy, że ES, DI, AX, CX mają już prawidłowe wartości.

```
kopiuj: mov es:[di],ax  ;kopiuj AX pod adres ES:DI
inc di                  ;adres zwiększamy o 2
inc di
dec cx                  ;licznik zmniejszamy o 1
jnz kopiuj              ;sprawdzamy, czy koniec
```

Inaczej

```
rep stosw
```

Instrukcja STOSW:

- słowo z AX kopiowane jest pod adres ES:DI,
- DI jest zwiększane (gdy DF=0) lub zmniejszane (gdy DF=1) o 2.

Przedrostek REP powoduje, że zawartość CX zmiejszana jest o 1, a STOSW jest powtarzane, aż do osiągnięcia CX=0.

Porównywanie dwóch łańcuchów może być wykonane instrukcją CMPS, CMPSB, CMPSW.

Przykład Porównanie pierwszych 50 znaków tablic umieszczonych pod adresami tab1 i tab2.

```
mov si,OFFSET tab1  ;kompletujemy adres tab1
mov ax,SEG tab1
mov ds,ax
mov di,OFFSET tab2  ;kompletujemy adres tab2
mov ax,SEG tab2
mov es,ax
mov cx,50           ;pierwszych 50 znaków
cld                 ;wyzerowanie znacznika kierunku
repe cmpsw          ;porównujemy póki są równe
jne tab_ne
............
```

```
tab_ne:             ;wracamy z adresami na pierwszy
dec si              ;element nierówny
dec si
dec di
dec di
```

10.3 Procesory 386 i 486

Procesory 386 i 486 są zgodne ze swoimi poprzednikami. Rejestry AX, BX, CX, DX, SI, DI, BP, SP zostały rozszerzone do 32-bitowych, jako EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. Ich mlodsze połówki (słowa) są dostępne pod starymi nazwami, starsze połówki nie są dostępne oddzielnie. Rejestry segmentowe CS, DS, ES, SS pozostały jako 16-bitowe. Doszły dwa nowe 32-bitowe rejestry segmentowe FS i GS. Lista rozkazów została istotnie rozszerzona, np. PUSHAD kładzie, a POPAD zdejmuje ze stosu wszystkie rejestry ogólnego przeznaczenia. PUSHFD kładzie, a POPFD zdejmuje ze stosu rejestry znaczników. Wiele instrukcji operujących na bajtach i słowach, ma wersje działające na podwójnych słowach, np. do STOSB i STOSW doszła STOSD.

11 Koprocesor numeryczny

11.1 Typy zmiennoprzecinkowe

Liczby zmiennoprzecinkowe są podzbiorem liczb wymiernych postaci

$$z = (-1)^s m p^w, \tag{11.1.1}$$

- p – podstawa (liczba naturalna > 1),
- m – mantysa (nieujemna liczba wymierna o skończonym rozwinięciu przy podstawie p),
- s – znak ($s = 0$ lub $s = 1$),
- w – wykładnik (liczba całkowita).

Jeżeli

$$1 \leq m < p, \tag{11.1.2}$$

to przedstawienie liczby $z \neq 0$ (11.1.1) jest jednoznaczne.
W komputerze zawsze jest $p = 2$ (notacja binarna) natomiast do „normalnego" użytku jest $p = 10$ (notacja dziesiętna). W notacji dziesiętniej stosuje się zapis $z = x\,\mathrm{E}\,y = x10^y$, gdzie y jest liczbą całkowitą, a x jest liczbą wymierną o skończonym rozwinięciu dziesiętnym.
Dla $p = 2$, jeżeli spełniony jest warunek (11.1.2), to część całkowita mantysy jest równa 1. Można wtedy przyjąć ją za domyślną i nie zapamiętywać, a za pamiętywać tylko część ułamkową. Dla takich m jest to liczba znormalizowana.

Wykładnik spełnia nierówność $w_{\min} < w < w_{\max}$.
Liczby

$$w_{\min} + 1 \text{ oraz } w_{\max} - 1$$

określają najmniejszy i największy wykładnik liczby.
Dla liczb znormalizowanych z

$$2^{w_{\min}+1} \leq |z| < 2 \cdot 2^{w_{\max}-1} = 2^{\cdot \max} \tag{11.1.3}$$

Wartość $w_{\max}$ jest zarezerwowana dla reprezentacji symboli, a $w_{\min}$ – dla zera i liczb zdenormalizowanych.

Zero i liczby zdenormalizowane

Jeżeli $w = w_{\min}$, to z założenia $m < 1$ i wtedy $z = (-1)^s m 2^{w_{\min}+1}$.
Wobec tego

$$z = 0 \iff m = 0 \wedge w = w_{\min}, \tag{11.1.4}$$

oraz dla $|z| < 2^{w_{\min}+1}$

$$z = (-1)^s m \cdot 2^{w_{\min}+1} \iff m < 1 \wedge w = w_{\min}. \tag{11.1.5}$$

Symbole

Nieskończoności:

$$+\infty \iff m = 1.0 \wedge w = w_{\max} \wedge s = 0,$$
$$+\infty \iff m = 1.0 \wedge w = w_{\max} \wedge s = 1.$$

Nieliczby (NaN (ang. *Not a Number*):

$$m \neq 1 \wedge w = w_{\max}.$$

Formaty zmiennoprzecinkowe:

- pojedynczy, 32 bity,
- podwójny, 64 bity,
- chwilowy, 80 bitów.

Zapis liczby w trzech polach od najstarszego do najmłodszego bitu: pole znaku s, pole wykładnika w, pole mantysy m.
Pole znaku ma zawsze 1 bit.
Pole wykładnika zawiera wykładnik w dwójkowym kodzie spolaryzowanym w_B (polaryzacja=ang. $bias$), wg wzoru:

$$w = w_B - B$$

gdzie B jest polaryzacją wyznaczoną tak, że

$$w_{\min} = w_B - B \text{ dla } w_B = 000\ldots000\text{B},$$
$$w_{\max} = w_B - B \text{ dla } w_B = 111\ldots111\text{B}.$$

Pole mantysy m zawiera część ułamkową m' mantysy w naturalnym kodzie dwójkowym. W formacie chwilowym również jeden bit części całkowitej.

Format	s	Wykładnik w_B	I	Część ułamk. m'
32 bity	31	30 23		22 0
64 bity	63	62 52		51 0
80 bitów	79	78 64	63	62 0

Daje to następujące rozmiary pól:

Format	Wykładnik	Część ułamk.
32 bity	8	23
64 bity	11	52
80 bitów	15	63

Koprocesor obsługuje też typy całkowite: dwu- cztero i ośmiobajtowe w kodzie uzupełnień do dwóch oraz liczby w kodzie BDC – dziewięć bajtów (18 cyfr dziesiętnych) oraz bajt znaku (najstarszy bit w najstarszym bajcie).

11.2 Architektura koprocesora 80x87

Koprocesor 80x87 zawiera stos ośmiu rejestrów 80 bitowych, rejestry sterowania, stanu koprocesora, i stanu stosu oraz rejestry instrukcji i argumentu.
Stos rejestrów R0 – R7 rośnie dół, adres wierzchołka stosu zajmuje jako wskaźnik stosu rejestrów ST, trzy bity rejestrze stanu koprocesora. Rejestry adresowane są wyłącznie względem wierzchołka stosu. Rejestr na wierzchołku stosu ST(0). Przez ST(n) oznaczony jest rejestr leżący o n pod wierzchołkiem, czyli mający adres rejestru o n większy.
Po położeniu na stos nowej wartości, rejestr będący poprzednio ST(n) staje się rejestrem ST($n + 1 \mod 8$).

Po inicjacji, wskaźnik stosu rejestrów ma wartość 0, czyli wartość zostanie wpisana do R7.
Z każdym rejestrem stosu związane jest dwubitowe pole stanu rejestru Rn w rejestrze stanu stosu koprocesora.

- $Rn = 00$ – liczba poprawna,
- $Rn = 01$ – zero,
- $Rn = 10$ – liczba niepoprawna lub nieskończoność,
- $Rn = 11$ – rejestr pusty.

11.3 Wybrane instrukcje koprocesora 80x87

Definicje liczb w formacie koprocesora:

DW – liczba całkowita dwubajtowa,

DD – liczba całkowita czterobajtowa,

DQ – liczba całkowita ośmiobajtowa,

DT – liczba całkowita dziesięciobajtowa w BDC,

DD – liczba zmiennoprzecinkowa w formacie pojedynczym,

DQ – liczba zmiennoprzecinkowa w formacie podwójnym,

DT – liczba zmiennoprzecinkowa w formacie chwilowym.

zmp-poj – liczba zmiennoprzecinkowa w formacie pojedynczym
zmp-pod – liczba zmiennoprzecinkowa w formacie podwójnym
zmp-chw – liczba zmiennoprzecinkowa w formacie chwilowym
zmp = zmp-poj|zmp-pod|zmp-chw
calk2 – liczba całkowita (binarna) dwubajtowa
calk4 – liczba całkowita (binarna) czterobajtowa
calk8 – liczba całkowita (binarna) ośmiobajtowa
calk = calk2|calk4|calk8

Instrukcje przesłania.

FLD ST(i)|zmp

umieszcza na stosie argument i zmniejsza wskaźnik stosu.

FST ST(i)|zmp-poj|zmp-podw

zapisuje wartość z wiechołka stosu w argumencie.

FSTP ST(i)|zmp

zapisuje wartość z wiechołka stosu w argumencie i zdejmuje ją ze stosu (zwiększa wskaźnik stosu).

FXCH ST(i)

wymienia wartość z wierzchołka stosu z zawartością rejestru.

FILD całk

umieszcza na stosie argument i zmniejsza wskaźnik stosu.

FIST calk2|calk4
zapisuje wartość z wiechołka stosu w argumencie.
FISTP calk
zapisuje wartość z wiechołka stosu w argumencie i zdejmuje ją ze stosu (zwiększa wskaźnik stosu).

Instrukcje arytmetyczne.
Format stosowy: Fop – wykonuje operacje na argumentach ST i ST(1), wynik w ST(1), zdejmuje ST ze stosu.
Format rejestrowy: Fop ST(i),ST|Fop ST,ST(i) – wykonuje operacje na argumentach ST i ST(i), wynik w pierwszym, nie ma zdjęcia ze stosu.
Format rejestrowy ze zdjęciem ze stosu: FopP ST(i),ST – wykonuje operacje na argumentach ST i ST(i), zdejmuje ST ze stosu.
Format z argumentem zmiennoprzecinkowym: Fop zmp-poj|zmp-pod – wykonuje operacje na argumentach ST i zmp-poj|zmp-pod.
Format z argumentem całkowitym: FIop calk2|calk4 – wykonuje operacje na argumentach ST i calk2|calk4.
Operacjami op mogą być ADD,SUB,SUBR, MUL, DIV, DIVR.

Przykład.
FADD
dodaje ST do ST(1) i umieszcza w ST(1), zdejmuje ST ze stosu.
FADD ST(i),ST|ST,ST(i)
dadaje argumenty, wynik w pierwszym.
itd.
FSQRT – oblicza pierwiastek kwadratowy z wartości na wierzchołku stosu i wpisuje tam wynik.
FRNDINT – zaokrągla liczbę z wierzchołka stosu do całkowitej i wpisuje tam wynik.

Instrukcje obliczania funkcji przestepnych.

FPTAN – oblicza tangens z liczby ST(0) w postaci y/x, umieszcza y w ST(0), po czym kładzie x na stos.

Inne: FPATAN, FSIN, FCOS, FSINCOS i.in.

Na przykład FYL2X oblicza wartość wyrażenia $y \log_2 x$, gdzie y to ST(1), x to ST(0). Następnie x jest zdejmowane ze stosu, a wynik do ST.

12 Wstawki w asemblerze

12.1 Turbo Pascal

Wstawka w asemblerze w postaci

```
asm
    {tu treść programu w asemblerze}
end;
```

Zasady działania na przykładach.

Funkcja lub procedura, której część lub całość jest napisana w asemblerze.

```
program minus;
var x,y:word;

procedure x_minus_y;
begin
asm
   mov cx,ds:x
   sub cx,ds:y
   mov ds:x,cx
end;
```

```
end;

begin
     writeln('Podaj liczby: ');
     repeat
          write('x=');read(x);
          write('y=');read(y);
          x_minus_y;
          writeln(x);
     until false;
end.
```

Funkcja lub procedura, której całość jest napisana w asemblerze.

```
program minusasm;
var x,y:word;

procedure x_minus_y;assembler;
asm
   mov cx,ds:x
   sub cx,ds:y
   mov ds:x,cx
end;
```

```
begin
     writeln('Podaj liczby: ');
     repeat
          write('x=');read(x);
          write('y=');read(y);
          x_minus_y;
          writeln(x);
     until false;
end.
```

Różnice w deklaracji i użyciu stałych i zmiennych.
Deklaracja

```
const
x=10;
y=20;
var
z:integer;
```

Fragment w asemblerze

```
asm
mov z,x+y
end;
```

Dodawanie stałych x i x na etapie kompilacji.
Natomiast, gdy x i y są zmiennymi, to

```
var
x,y,z:integer;
```

Fragment w asemblerze

```
asm
mov ax,z
add ax,y
mov z,ax
end;
```

Dodawanie stałych x i x na etapie wykonania.

Procedura w asemblerze, zasemblowana przez TASM do postaci `*.obj`.

```
.MODEL TPASCAL
        .DATA
        EXTRN x:word,y:word
        .CODE
PUBLIC x_minus_y
x_minus_y PROC NEAR
mov cx,ds:x
sub cx,ds:y
mov ds:x,cx
ret
x_minus_y ENDP
```

```
END
```

Uwaga.
Chronione muszą być rejestry BP, SP, SS i DS. Inne mogą być dowolnie modyfikowane.

```
procedure x_minus_y;
begin
asm
   push ds {bez tego}
   mov ds,ax
   pop ds  {i bez tego nie działa}
   mov cx,ds:x
   sub cx,ds:y
   mov ds:x,cx
end;
end;
```

12.2 C i C++

W Borland C i Borland C++ (w tym CBuilder) tak samo jak w Pascalu

13 Programy dla MS Windows

13.1 Win API

Przykłady są modyfikacjami programów z pliku `asm_kursasm.zip` *ze strony* `www.programik.com`
Przy programowaniu w Windows nie posługujemy się przerwaniami i funkcjami DOS, tylko funkcjami API.
Funkcji tych używa się również programując w C, C++, Pascalu, Delphi itd. Opisy tych funkcji można więc znaleźć w podręcznikach tych języków i ich implementacji dla MS Windows.

Przykład.

```
int MessageBox(
HWND hWnd,  // uchwyt właściciela okna
LPCTSTR lpText,  // adres tekstu w oknie komunikatu
LPCTSTR lpCaption,  // adres tytułu w oknie komunikatu
UINT uType  // styl okna komunikatu
);
```

Styl okna komunikatu jest flagą, np.

MB_ABORTRETRYIGNORE	przerwij, ponów próbę, zignoruj
MB_OK	ok
MB_OKCANCEL	ok, anuluj
MB_RETRYCANCEL	ponów próbę, anuluj
MB_YESNO	tak, nie
MB_YESNOCANCEL	tak, nie, anuluj

Aby dodatkowo wyświetlic ikonkę, dopisuje się do kodu przycisku "or" oraz odpowiednią wartość z tabeli poniżej.

MB_ICONEXCLAMATION	
MB_ICONWARNING	wykrzyknik
MB_ICONINFORMATION	
MB_ICONASTERISK	mała litera "i" w niebieskim kółku
MB_ICONQUESTION	znak zapytania
MB_ICONSTOP	
MB_ICONERROR	
MB_ICONHAND	krzyżyk

13.2 Struktura programu

Program w asemblerze dla Windows ma następującą strukturę:

```
.386
.model flat, stdcall
option casemap:none

include \masm32\include\windows.inc
;deklaracje bibliotek, których chcemy użyć

.const
;stałe
```

```
.data
;zmienne, które są dane od początku działania programu

.data?
;zmienne niezdefiniowane (podajemy tylko typ zmiennej)

.code
start:
;kod assemblerowy
end start
```

Używane funkcje API znajdują się (dla MASM) w bibliotekach kernel32 i user32, trzeba je więc zadeklarować

```
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
```

13.3 Przykłady programów

Przykład 1.

```
.386
.model flat, stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
```

```
.data
msgTytul db "Programowanie niskopoziomowe",0
msgTekst db "Ostatni wykład z programowania niskopoziomowego",0
; (*)
.code
start:
 invoke MessageBox,0,addr msgTekst,addr msgTytul,MB_OK
 invoke ExitProcess,0
end start
```

Linia (*) może mieć postać

```
"Ostatni wykład",10,13, "z programowania niskopoziomowego",0
```

Funkcje API zwracają wartości w rejestrze EAX. Sprawdza się to w następujący sposób:

```
.IF warunek
    polecenia
.ELSEIF warunek
    polecenia
.ELSE
    polecenia
.ENDIF
```

Przykład 2.

```
.386
.model flat, stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

.data
msgTytul db "Programowanie niskopoziomowe",0
msgTekst db "Wciśnij tak lub nie.",0
msgYesTekst db "Wcisnąłeś ",34,"Tak",34,"!",0
msgNoTekst db "Wcisnąłeś ",34,"Nie",34,"!",0

.code
start:
  invoke MessageBox,0,addr msgTekst,addr msgTytul,MB_YESNO or MB_ICONQUESTION
  .IF eax==IDYES
```

```
    invoke MessageBox,0,addr msgYesTekst,addr msgTytul,MB_OK or MB_ICONEXCLAMATION
  .ELSE
    invoke MessageBox,0,addr msgNoTekst,addr msgTytul,MB_OK or MB_ICONEXCLAMATION
  .ENDIF
  invoke ExitProcess,0
end start
```