AKADEMIA GÓRNICZO – HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I ELEKTRONIKI

KATEDRA AUTOMATYKI NAPĘDU I URZĄDZEŃ PRZEMYSŁOWYCH

MIKROPROCESOROWE METODY STEROWANIA

Mikrokontrolery rodziny MCS-51

Cz. II.

Autor:

Dr inż. Zbigniew Waradzyn

Kraków 2005

12. ZEROWANIE, USTAWIANIE I ODCZYT STANU LINII PORTU

Poniżej zostaną omówione sposoby *zerowania*, *ustawiania* i *odczytu* stanu linii portu przy użyciu rozkazów **asemblera** (języka programowania niskiego poziomu) mikrokontrolerów rodziny 8051. Rozkazy należy umieścić *w pliku tekstowym*. Bardziej szczegółowe informacje na temat asemblera oraz kroków, jakie należy wykonać, aby tego typu rozkazy przekształcić do postaci „zrozumiałej" dla mikrokontrolera zostaną podane w dalszej części opracowania.

12.1. Zapalenie diody świecącej

Jak wynika z rysunku 9.1 celem zapalenia diody należy **wyzerować** odpowiednie wyjście cyfrowe (odpowiednią linię portu). Niech naszym zadaniem będzie zapalenie diody ***D1*** – trzeba więc wyzerować linię **P1.1**, gdyż do niej przyłączona jest dioda *D1*.

Do **wyzerowania** pojedynczej linii portu (i ogólnie do wyzerowania bitu) służy instrukcja (rozkaz) o postaci ogólnej ***clr bit*** (ang. ***cl***ea***r*** *bit*), w której **clr** jest **symboliczną nazwą** instrukcji, a symbol **bit** oznacza adres bitu, który należy wyzerować. Ten adres bitu jest w przypadku omawianej instrukcji **argumentem instrukcji**.

Tak więc zapalamy diodę *D1* instrukcją:

clr P1.1 (zeruj bit *P1.1*), w której *P1.1* oznacza adres linii portu.

Wobec powyższego diodę ***D4*** zapalimy instrukcją ***clr P1.4***, itd.

Prawda, że proste?

12.2. Zgaszenie diody świecącej

W celu zgaszenia diody należy **ustawić** odpowiednie wyjście cyfrowe (odpowiednią linię portu). Niech naszym zadaniem będzie zgaszenie diody **D2** – trzeba ustawić linię **P1.2**.

Do **ustawienia** pojedynczej linii portu (i ogólnie do ustawienia bitu) służy instrukcja o postaci ogólnej ***setb bit*** (ang. ***set b****it*), w której **setb** jest *symboliczną nazwą* instrukcji (w dalszej części będzie używane pojęcie *nazwa instrukcji*), a symbol *bit* oznacza adres bitu, który należy ustawić (argument instrukcji).

Tak więc gasimy diodę *D2* instrukcją:

setb P1.2 (ustaw bit *P1.2*), w której *P1.2* oznacza adres linii portu.

Podobnie diodę *D3* zgasimy instrukcją ***setb P1.3***, itd.

Uwaga*:* W podanych powyżej instrukcjach zerowania i ustawiania linii portów oznaczenia *P1.1*, *P1.4*, *P1.3*, itd. nie są dokładnie adresami bitów, tylko ich nazwami. Wpisywanie nazw bitów zamiast ich adresów bardzo ułatwia pracę programiście: przecież znacznie łatwiej zapamiętać nazwę, jak np. *P1.1*, niż adres, np. *91H*. Adresy bitów zostaną omówione w dalszej części opracowania.

12.3. Modyfikacja całego portu (wszystkich jego linii równocześnie) za pomocą jednej instrukcji.

Przedstawione powyżej instrukcje *clr bit* i *setb bit* wykonują operacje na pojedynczych bitach. Aby więc wymusić odpowiednie stany wszystkich linii portu, należy w programie umieścić 8 takich instrukcji. Jednakże istnieje możliwość modyfikacji wszystkich linii wybranego portu za pomocą jednej instrukcji, np. instrukcji o postaci ogólnej ***mov ad, #n*** (ang. ***mov****e data*), w której **mov** jest nazwą instrukcji, **ad** (pierwszy argument instrukcji) – ogólnie adresem komórki pamięci (rejestru), u nas adresem rejestru odpowiadającego portowi (adresem rejestru portu), zaś **#n** oznacza liczbę (drugi argument instrukcji). Efektem wykonania tej instrukcji będzie wpisanie liczby podanej jako drugi argument do rejestru, którego adres podano jako pierwszy argument.

Liczbę tę możemy przedstawić w postaci dziesiętnej, szesnastkowej lub dwójkowej. Aby więc uzyskać taki stan diod, jakiego wymagamy przy otwartym *W1* (rys.11.1), należy wykonać instrukcję:

mov P1, #1110 1101B

Przy zapisie liczby w postaci dwójkowej, jak powyżej, od razu widać, który linie portu zostaną wyzerowane (*P1.1* i *P1.4*), a które ustawione (pozostałe). Otrzymany stan linii portu odpowiada przedstawionemu w p. 10 d), s. 27.

Ponieważ *1110 1101B = EDh = 237* (sprawdź!), tę samą instrukcję można zapisać używając zapisu szesnastkowego liczby

mov P1, #0EDh

lub dziesiętnego

mov P1, #237d lub ***mov P1, #237***

Zapis dziesiętny daje tu najmniejszą przejrzystość (trzeba przeliczać), więc o wiele wygodniej jest korzystać z zapisu dwójkowego (binarnego) lub szesnastkowego (heksadecymalnego).

Uwaga 1*:* W podanych powyżej instrukcjach oznaczenie *P1* nie jest *adresem rejestru*, tylko jego **nazwą**, analogicznie jak w podanych wcześniej instrukcjach zerowania i ustawiania linii portów. Adresy rejestrów zostaną omówione w dalszej części opracowania.

Uwaga 2*:* Jeśli liczba szesnastkowa zaczyna się od litery (*A, B, C, D, E* lub *F*), musi w programie być poprzedzona zerem, jak w podanej wyżej instrukcji *mov P1, #**0**EDh*. Stosowane w tym opracowaniu zapisy typu *EDh* oraz *0EDh* są równoważne.

Uwaga 3*:* Znak „#” przed liczbą (w drugim argumencie instrukcji) oznacza, że do rejestru ma być wpisana ta właśnie liczba. Gdyby, na przykład, nie podano znaku „#” przed liczbą *0EDh*, do rejestru *P1* zamiast **liczby** *0EDh* zostałyby wpisana **zawartość komórki o adresie** *0EDh*.

Uwaga 4*:* W przypadku zapisu liczby w systemie dziesiątkowym możemy po tej liczbie umieścić literę ***d*** (np. *mov P1, #237**d***) lub jej nie umieszczać (*mov P1, #237*).

Zapamiętaj*:* Instrukcje ***clr*** *P1.1* i ***setb*** *P1.2* powodują wykonanie operacji na bitach, zaś instrukcja ***mov*** *P1, #0DEh* powoduje wykonanie operacji na całym rejestrze (8-bitowej komórce pamięci).

12.4. Odczyt stanu styku przyłączonego do wejścia cyfrowego

Jak widać z rysunku 9.1, styk *W1* przyłączony jest do linii *4* portu *P3*, czyli do linii *P3.4* (wyprowadzenie 14 układu scalonego) i odczytując stan tej linii otrzymujemy informację o stanie styku: stan *0* – styk zamknięty, stan *1* – styk otwarty. Podobnie można sprawdzić stan styku *W2* (linia *P3. 5*, wyprowadzenie 15).

Stan linii portu może być odczytany programowo. Istnieją instrukcje dokonujące tylko odczytu linii portu, jak i instrukcje dokonujące odczytu linii, testowania jej stanu i realizujące skok w różne miejsca w programie zależnie od wyniku testu.

12.4.1. Odczyt pojedynczej linii portu

Do **odczytu** pojedynczej linii portu (i ogólnie do odczytu bitu) służy instrukcja o postaci ogólnej ***mov C, bit*** w której **C** oznacza wspomniany już znacznik przeniesienia *CY* (p.8.3d), a symbol **bit** oznacza adres bitu, który należy odczytać (skopiować do znacznika *CY*).

Tak więc stan styku *W1* możemy sprawdzić przez odczyt stanu linii *P3.4* instrukcją

mov C, P3.4 – skopiuj stan bitu (w tym przypadku stan końcówki *P3.4* mikrokontrolera) do znacznika *CY*

Znacznik *CY* służy w powyższym przypadku do chwilowego przechowania stanu **bitu** przed jego przetestowaniem za pomocą kolejnej instrukcji.

12.4.2. Odczytu pojedynczej linii portu z testowaniem jej stanu.

Omawiane w niniejszym podpunkcie instrukcje dokonują odczytu stanu linii portu (ogólnie – stanu bitu), testowania tego stanu i realizują następnie przejście w odpowiednie miejsce w programie: inne, gdy bit jest ustawiony, a inne – gdy wyzerowany. Mówiąc ściślej jest tak, że w jednym z powyższych przypadków następuje skok w inne miejsce w programie, a w drugim wykonywana jest kolejna instrukcja z pamięci programu. Istnieją dwie możliwości:

a) skok następuje, gdy bit jest **wyzerowany**

Do realizacji powyższego służy instrukcja o postaci ogólnej ***jnb bit, d,*** w której, *bit* oznacza adres bitu, którego stan należy odczytać i przetestować, a *d* określa **względny adres skoku**. Instrukcja ta wykonuje skok w inne miejsce programu (określone wartością argumentu *d*), jeśli

bit jest wyzerowany (ang. ***j**ump if **b**it is **n**ot set* – skocz, gdy bit nie jest ustawiony), w przeciwnym razie (bit ustawiony) zostanie pobrana kolejna instrukcja z pamięci programu.

W naszym programie instrukcją odczytującą stan styku *W1* (linii portu *P3.4* – rys. 9.1) i dokonującą skoku do adresu o etykiecie *A2*, jeśli ten styk jest zamknięty (rys. 11.1) będzie instrukcja

jnb P3.4, A2 - skok do adresu o etykiecie *A2*, gdy bit *P3.4* (końcówka *P3.4* mikrokontrolera) przyjmuje stan *0*.

Następną instrukcją w programie musi być instrukcja wykonywana w przypadku, gdy styk *W1* jest otwarty, czyli instrukcja oznaczona etykietą *A3* (patrz rys. 11.1).

b) skok następuje, gdy bit jest **ustawiony**,

Tutaj można zastosować instrukcję o postaci ogólnej ***jb bit, d,*** w której, analogicznie jak w instrukcji poprzedniej, *bit* oznacza adres bitu, którego stan należy odczytać i przetestować, zaś *d* określa względny adres skoku. Instrukcja ta jest bardzo podobna do instrukcji *jnb bit, d*; różni się od niej tym, że w tym przypadku skok w inne miejsce programu wykonywany jest wtedy, gdy bit jest ustawiony (ang. ***j**ump if **b**it is set* – skocz, gdy bit jest ustawiony).

W naszym programie instrukcją odczytującą stan styku *W2* (czyli stan linii portu *P3.5* – rys. 9.1) i dokonującą skoku do adresu o etykiecie *A4*, jeśli ten styk jest otwarty (rys. 11.1) będzie instrukcja

jb P3.5, A4 - skok do adresu o etykiecie *A4*, gdy bit *P3.5* (końcówka *P3.5* mikrokontrolera) przyjmuje stan *1*.

Następną instrukcją w programie musi być instrukcja, którą należy wykonać, gdy styk *W2* jest zamknięty, czyli instrukcja oznaczona etykietą *A5* (patrz rys. 11.1).

Uwaga*:* W przedstawionych instrukcjach zamiast adresu bitu podano jego *nazwę* (*P3.4*, *P3.5*), a zamiast względnego adresu skoku – *etykietę* tego miejsca w programie, do którego ma być wykonany skok, gdy bit jest wyzerowany (*A2* – punkt a), względnie ustawiony (*A4* – punkt b). Po „przetłumaczeniu" instrukcji na język „zrozumiały" dla mikrokontrolera nazwa bitu zostanie przekształcona w adres bitu, zaś etykieta zostanie zapisana w inny sposób, ale to będzie wyjaśnione w dalszej części opracowania.

12.4.3. Równoczesny odczyt wszystkich linii portu

Przedstawione powyżej instrukcje *mov C, bit; jnb bit, d* i *jb bit, d* mają jako argumenty pojedyncze bity, czyli umożliwiają odczyt pojedynczej linii portu. Jednakże istnieje także możliwość odczytu wszystkich linii portu za pomocą jednej instrukcji, np. ***mov A, ad***, w której

mov jest nazwą instrukcji, **A** oznacza akumulator, zaś **ad** jest ogólnie adresem komórki pamięci (rejestru), u nas adresem rejestru portu. Efektem wykonania tej instrukcji będzie skopiowanie zawartości komórki o adresie *ad* do akumulatora. Odczytu portu *P3* możemy dokonać instrukcją

mov A, P3 (skopiuj stan portu *P3* do akumulatora).

Po wykonaniu odczytu stanu portu, któremu teraz odpowiada również stan akumulatora, możemy za pomocą kolejnych instrukcji testować stan jego poszczególnych bitów lub wykonać operację na całej zawartości akumulatora.

Uwaga*:* Z porównania przedstawionych instrukcji *mov C, bit* (kopiowanie bitu – p.12.4.1) oraz *mov A, ad* (kopiowanie całego bajta) widać, że znacznik *CY* (tu *C*) może spełniać analogiczną rolę w stosunku do bitów, jak akumulator (*A*) w stosunku do bajtów. Znacznik *CY* pełni tu rolę tzw. **akumulatora boolowskiego** (porównaj p. 8.3.d, s. 24).

12.4.4. Odczyt stanu końcówki, czy odczyt stanu przerzutnika?

Jak podano wcześniej (p.10e, s.29) przy odczycie stanu linii portu niektóre instrukcje dokonują odczytu końcówki (wyprowadzenia układu scalonego), a niektóre odczytują stan przerzutnika (bufora) (porównaj rys. 10.1 – *„Odczytywanie końcówki”* lub *„Odczytywanie bufora”*).

Przypomnienie: Jeśli dana linia portu ma być wejściem, do przerzutnika musi być wpisana „*1*”.

Pytanie: Linie *P3.4* i *P3.5* są u nas wejściami, więc do odpowiednich przerzutników muszą być wpisane „*1*”. Przy odczycie stanu wejścia, które zależnie od wymuszenia zewnętrznego może przyjmować stan *0* lub *1*, nie miałoby więc sensu sprawdzanie stanu przerzutnika. Czy podane wyżej instrukcje odczytu linii portu rzeczywiście odczytują stan końcówki?

Odpowiedź: Podane wyżej instrukcje *mov C, bit*, *jnb bit, d*, *jb bit, d* oraz *mov A, ad* **zawsze odczytują stan końcówki**, mogą więc być stosowane do odczytu wejść cyfrowych. Instrukcje te nie modyfikują stanu odczytanego bitu (dotyczy pierwszych trzech powyższych instrukcji) ani rejestru (dotyczy czwartej z zamieszczonych powyżej instrukcji).

Uwaga*:* Odczytu *wyjścia przerzutnika* dokonują te instrukcje, które po odczycie stanu bitu lub rejestru mogą go **zmodyfikować**. Przykładami takich instrukcji są:

- ***jbc bit, d*** (ang. ***j****ump if* ***b****it is set and* ***c****lear bit*) – instrukcja podobna do poznanej już instrukcji *jb bit, d*, ale realizująca dodatkowo zerowanie bitu po przetestowaniu jego stanu,
- ***inc ad*** (ang. ***inc****rement*) – jeśli jako adres podamy nazwę portu, instrukcja spowoduje **odczyt rejestru** portu jako liczby w naturalnym kodzie dwójkowym, zwiększenie jej o 1 oraz zapis tak otrzymanej liczby z powrotem do rejestru portu.

13. ZARYS PROGRAMU

Bazując na schemacie układu z rys. 9.1, algorytmie programu z rys. 11.1 i podanych powyżej informacjach można już napisać zarys programu w **asemblerze**.

```
; Zarys programu przes_a
; zapalenie diod D1 i D4
    clr   P1.1
    clr   P1.4
; zgaszenie diod D2, D3, D5 i D6
    setb  P1.2
    setb  P1.3
    setb  P1.5
    setb  P1.6
; Czy ma być przesuw? ( testowanie stanu przycisku W1)
A1:
    jnb P3.4, A2
; Nie ma przesuwu  - zapalenie diod D1 i D4 oraz zgaszenie diod D2, D3, D5 i D6 za pomocą
; jednej instrukcji
A3:
    mov P1, #0EDh
    sjmp A1                 ; skok względny bezwarunkowy do etykiety A1
; Kontrola kierunku przesuwu ( testowanie stanu przycisku W2)
A2:
    jb P3.5, A4
A5:
;  tu należy podać instrukcje realizujące przesuw w prawo
    sjmp A6                 ; skok względny bezwarunkowy do etykiety A6
A4:
;  tu należy podać instrukcje realizujące przesuw w lewo
A6:
;  tu należy podać instrukcje realizujące opóźnienie
    sjmp A1                 ; skok względny bezwarunkowy do etykiety A1
end                         ; ta instrukcja zawsze konieczna na końcu programu
```

Właściwą treść programu stanowią instrukcji podane czcionką pogrubioną oraz etykiety A1 ÷ A6. Wszystkie teksty umieszczone w danym wierszu na prawo od średnika „**;**” traktowane są jak komentarz. Komentarze są przydatne dla samego programisty, jak i dla innych osób chcących zrozumieć działanie programu, natomiast są całkowicie pomijane przy „tłumaczeniu” programu do postaci „zrozumiałej” przez mikrokontroler.

W naszym programie pojawiła się nowa instrukcja o postaci ogólnej ***sjmp d*** (ang. ***j**u**mp** relative* (***s**hort*) – skok względny (krótki)) realizująca bezwarunkowy skok „krótki" w miejsce programu określone argumentem *d* (względny adres skoku). Programista jako ten argument podaje odpowiednią etykietę, podobnie jak w instrukcjach *jnb bit, d* oraz *jb bit, d*.

Uwaga*:* Pewnego wyjaśnienia wymagają pojęcia *skok względny, bezwarunkowy* i *„krótki"*:

- **skok względny** jest wtedy, gdy argument instrukcji nie podaje bezpośrednio adresu komórki pamięci programu, do której należy wykonać skok, a jedynie zawiera informację o tym, o ile komórek w pamięci programu należy się przemieścić, licząc od pozycji bieżącej. Natomiast dokładny adres komórki pamięci programu, do której należy wykonać skok, zawarty jest w argumencie instrukcji *skoku bezwzględnego*, np. *ljmp ad16*, z którą spotkamy się później.
- ze **skokiem bezwarunkowym**, np. *sjmp d*, mamy do czynienia wtedy, gdy po napotkaniu takiej instrukcji **zawsze** następuje skok w określone miejsce w programie. W odróżnieniu od instrukcji skoku bezwarunkowego istnieje także wiele instrukcji realizujących *skok warunkowy*, w których skok następuje **tylko** przy spełnieniu określonego warunku, np. poznane już instrukcje *jnb bit, d* oraz *jb bit, d*.
- **skok „krótki"** oznacza, że skoku można dokonać tylko w ograniczonym zakresie, czyli o określoną ilość komórek do przodu i określoną ilość komórek do tyłu. Zostanie to wyjaśnione szczegółowo w dalszej części materiałów.

Należy zauważyć, że w naszym programie wykonanie operacji zapalenia i zgaszenia odpowiednich diod dokonywane jest dwoma sposobami:

- w części inicjalizacyjnej programu każda dioda zostaje zapalana bądź zgaszona oddzielnie (instrukcje *clr P1.1*; *setb P1.2*, itd. – porównaj p. 12.1 i 12.2),
- w dalszej części programu (etykieta *A3*) odpowiedni stan diod uzyskuje się za pomocą jednej instrukcji *mov P1, #0EDh* (porównaj p. 12.3).

Użycia różnych instrukcji dokonano w celu zademonstrowania, że ten sam efekt można niejednokrotnie uzyskać różnymi sposobami.

Na końcu programu musi znajdować się instrukcja **end,** informująca o **końcu programu.**

Nie jest to rozkaz (instrukcja) mikrokontrolera 8051 – jest to tzw. instrukcja **makroasemblera**, ale mowa o tym będzie później.

Celem dokończenia programu trzeba jeszcze zrealizować „przesuw" w lewo i w prawo oraz opóźnienie, co zostanie przedstawione w następnych punktach.

14. REALIZACJA PRZESUWU

14.1. Przesuw w lewo

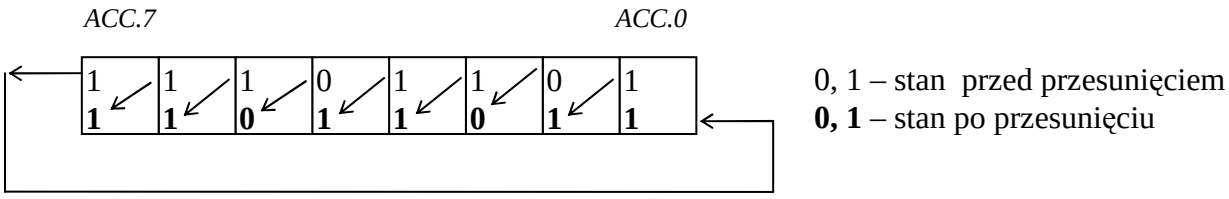
Załóżmy, że załączamy nasz układ „przesuwu" przy otwartych stykach *W1* i *W2*. Świecą diody *D1* i *D4*, a pozostałe są zgaszone. Taki stan diod zapewnia następujący stan rejestru *P1*: *1**110 110**1B*. Po zamknięciu *W1* powinniśmy uzyskać przesuw w lewo (rys. 11.1), co oznacza, że:

- po upływie pierwszego okresu opóźnienia stan portu *P1* powinien wynosić *1**101 101**1B*,
- po upływie drugiego okresu opóźnienia - *1**011 011**1B*,
- po upływie trzeciego okresu opóźnienia - *0**110 111**1B*, itd.

Uwaga: Czcionką pogrubioną zaznaczono te bity, które decydują o stanie diod (bity najbardziej znaczący *(MSB)* i najmniej znaczący *(LSB)* nie wpływają na stan diod – rys. 9.1).

Pytanie: Jak zrealizować zmianę stanu portu w opisany wyżej sposób?

Odp.: Istnieje instrukcja realizująca tego typu przesuw bitów w akumulatorze (ale **tylko w akumulatorze,** nie w innych rejestrach). Jej postać jest następująca: ***rl A***: **rl** (ang. ***r****otate* ***l****eft* – obróć w lewo) to symboliczna nazwa instrukcji, zaś **A** oznacza akumulator. Jak widać na poniższym rysunku, w efekcie wykonania omawianej instrukcji każdy bit zostaje przesunięty o jedna pozycję w lewo, przy czym bit najstarszy, czyli *ACC.7*, zostaje przepisany w miejsce bitu najmłodszego, czyli *ACC.0.*



Pytanie: Przedstawiona instrukcja realizuje przesuw w akumulatorze, a my potrzebujemy dokonać przesuwu w rejestrze *P1*. Czy da się więc wykorzystać tę instrukcję?

Odp.: Tak. Należy najpierw skopiować zawartość rejestru *P1* do akumulatora, zrealizować przesuw bitów w akumulatorze, a następnie skopiować zawartość akumulatora do rejestru *P1*.

```
mov A, P1      ; skopiowanie stanów wyjść do akumulatora,
rl A           ; przesuw w lewo,
mov P1, A      ; skopiowanie zawartości akumulatora do rejestru P1.
```

Jak widać, dwukrotnie zastosowano tu przedstawioną już wcześniej instrukcję o nazwie ***mov ....*** .

W pierwszym przypadku postać tej instrukcji to ***mov A, ad*** – kopiowanie zawartości komórki

o adresie **ad** do akumulatora, a w drugim ***mov ad, A*** – kopiowanie zawartości akumulatora do komórki o adresie **ad**. Jak widać, w wyniku wykonania tej instrukcji następuje kopiowanie zawartości komórki, która podana jest **na ostatnim miejscu** do komórki podanej **przed nią**.

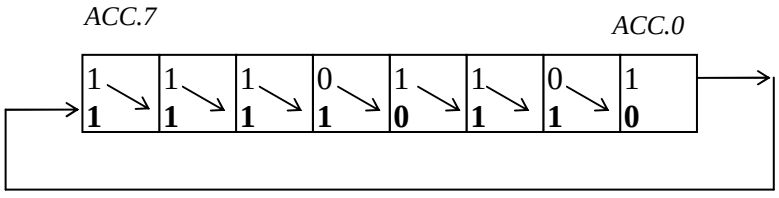
*Uwagi co do instrukcji **mov ...***:

- ogólna postać tej instrukcji to ***mov a, b***, czyli poza nazwą instrukcji występują w niej zawsze **2 dodatkowe** elementy,
- w efekcie jej wykonania następuje zawsze kopiowanie (zapis) w kierunku **a ← b**,
- jeśli jako *a* lub *b* wpiszę się liczbę, jest ona traktowana jako adres komórki (wyjątkowo adres bitu – patrz niżej). Jeśli do komórki chcemy wpisać konkretną liczbę, musimy ją poprzedzić znakiem „#". Porównaj:
  - *mov 30H, 20H* – skopiowanie **zawartości komórki** o adresie *20H* do komórki o adresie *30H*,
  - *mov 30H, #20H* – wpisanie **liczby** *20H* do komórki o adresie *30H*,
- istnieje **wiele kombinacji** elementów występujących w tej instrukcji; dotychczas podano kilka: „*mov ad, #n*", „*mov A, ad*", „*mov ad, A"* i „*mov C, bit*",

  przy czym:
  - *ad* – adres rejestru (komórki pamięci) z wewnętrznej pamięci danych mikrokontrolera,
  - *n* – liczba,
  - *A* – akumulator,
  - *C* – znacznik przeniesienia,
  - *bit* – adres bitu z wewnętrznej pamięci danych mikrokontrolera.
- instrukcja ta dokonuje **zwykle** operacji na **komórkach** pamięci; wyjątkiem są instrukcje, w której występuje *C* (znacznik przeniesienia), jak w instrukcji „*mov C, bit*". W tym przypadku kopiowany jest stan pojedynczego bitu. Podobnie jest w przypadku instrukcji „*mov bit, C*".

14.2. Przesuw w prawo

Przesuw w prawo realizowany jest analogicznie, jak przesuw w lewo, z tym, że należy tu zastosować instrukcję ***rr A*** (***r**otate **r**ight* – obróć w prawo).

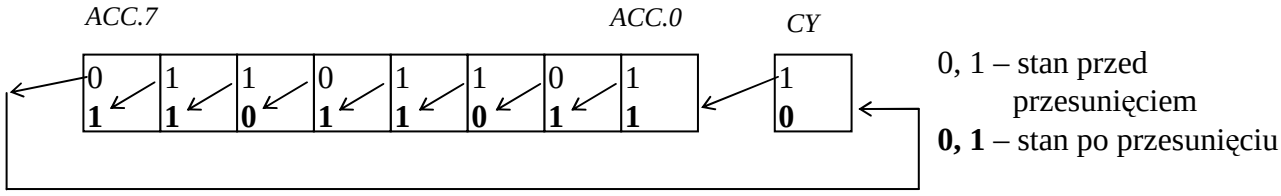


Przesuw w prawo uzyskamy więc wykorzystując instrukcje:

```
mov A, P1      ; skopiowanie stanów wyjść do akumulatora,
rr A           ; przesuw w prawo,
mov P1, A      ; skopiowanie zawartości akumulatora do rejestru P1.
```

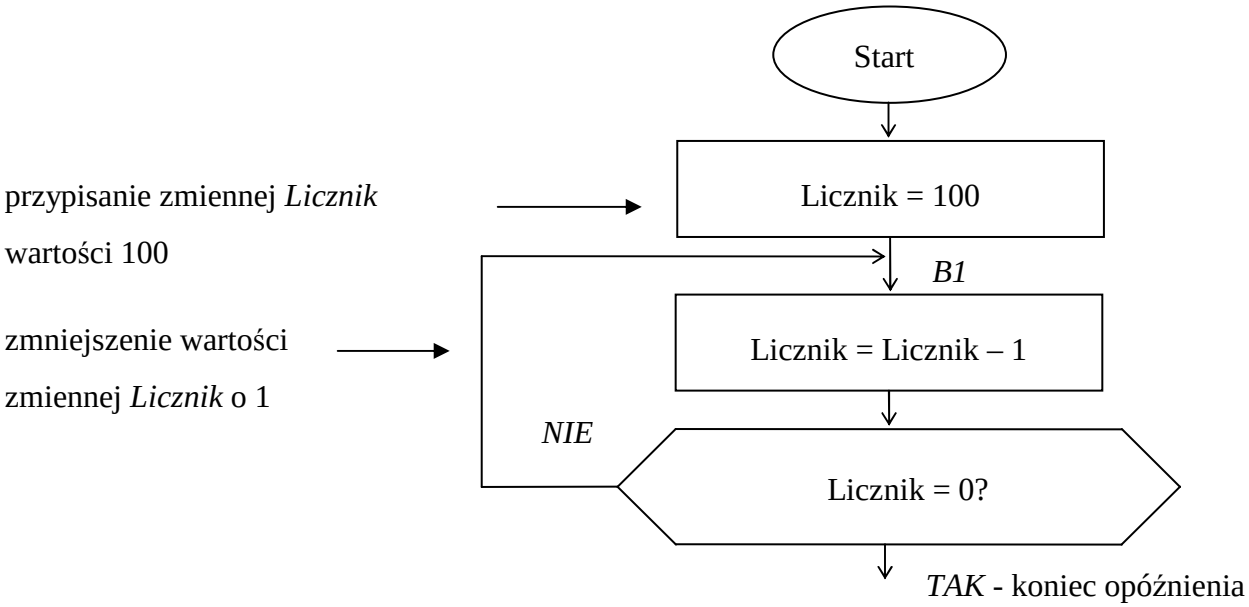
14.3. Przesuw z udziałem znacznika przeniesienia *CY*

Istnieją też instrukcje przesuwu w lewo i w prawo, odpowiednio ***rlc A*** i ***rrc A***, w których bierze udział także znacznik *CY*. Poniżej przedstawiono przykładowy efekt wykonania instrukcji *rlc A*.



15. REALIZACJA OPÓŹNIENIA

Opóźnienia czasowe można realizować na wiele sposobów. Na rysunku 15.1 przedstawiono algorytm prostego sposobu realizacji takiego opóźnienia. Wykorzystywana jest tu zmienna (komórka wewnętrznej pamięci danych mikrokontrolera), którą nazwiemy **Licznik**. O tym, jakie komórki mogą pełnić rolę tej zmiennej będzie mowa później.



Rys. 15.1. Prosty sposób realizacji opóźnienia

***Uwaga**:*

W obu powyższych „bloczkach" znak „**=**" nie oznacza wcale, że lewa strona równa się prawej. Znak ten pełni tu rolę tzw. **operatora przypisania**, co oznacza, że wartość z prawej strony ma być przepisana do zmiennej znajdującej się po lewej stronie.

Przedstawiony algorytm można zrealizować programowo tak:

```
        mov Licznik, #100          ; wartość początkowa
B1:
        djnz Licznik, B1
        ......................
```

Pierwsza instrukcja powoduje wpis wartości 100 (postać dziesiętna) do komórki pamięci o nazwie *Licznik*. Jest to znana nam już instrukcja postaci *mov ad, #n*.

Następną zastosowaną tu instrukcją jest ***djnz ad, d*** (ang. ***d**ecrement and **j**ump if **n**ot **z**ero* – zmniejsz o jeden i skacz, jeśli nie zero). Instrukcja ta wykonuje dwie czynności:

a) dekrementuje (zmniejsza o 1) liczbę zawartą w komórce o adresie *ad* (tę komórkę nazwaliśmy tu *Licznik*),

b) sprawdza, czy ta liczba (już po dekrementacji) jest równa zeru:
- jeśli nie – wykonywany jest skok do adresu wskazanego przez argument *d* (programista wpisuje tu etykietę: u nas jest to *B1*),
- jeśli tak – wykonywana jest następna instrukcja (znajdująca się w pamięci programu bezpośrednio za instrukcją *djnz ad, d*).

Uwaga: Argument *d* występujący w instrukcji *djnz ad, d* to znany nam już *względny adres skoku*, z którym spotkaliśmy się w instrukcjach *jnb bit, d*; *jb bit, d* i *sjmp d.*

Patrząc na podany powyżej algorytm łatwo zauważyć, że instrukcja ***djnz ad, d***, wykonuje operacje przedstawione w algorytmie w dwu bloczkach:

- dekrementacja zawartości komórki (co w prostokącie zapisaliśmy jako *Licznik = Licznik - 1*),
- testowanie zawartości komórki z wykonaniem ewentualnego skoku w inne miejsce w pamięci programu (przełącznik z zapytaniem *Licznik = 0?*).

W konsekwencji, w powyższym fragmencie programu następuje 100-krotne zmniejszanie wartości zmiennej *Licznik* – co trwa ok. 200 cykli maszynowych, gdyż instrukcja *djnz ad, d* zajmuje 2 cykle maszynowe. Przy założeniu, że jeden cykl maszynowy trwa 1μs uzyskuje się w powyższy sposób opóźnienie ok. 200 μs.

Uwaga*:* Jeden cykl maszynowy to 12 taktów sygnału zegarowego. Jeżeli zegar pracuje z częstotliwością 12 MHz to jeden cykl maszynowy trwa 1μs (porównaj rys. 8.4). Założenie, że **cykl maszynowy trwa 1μs** będziemy przyjmować także w dalszych rozważaniach.

Pytanie 1: Jakie maksymalne opóźnienie można uzyskać w sposób opisany powyżej?

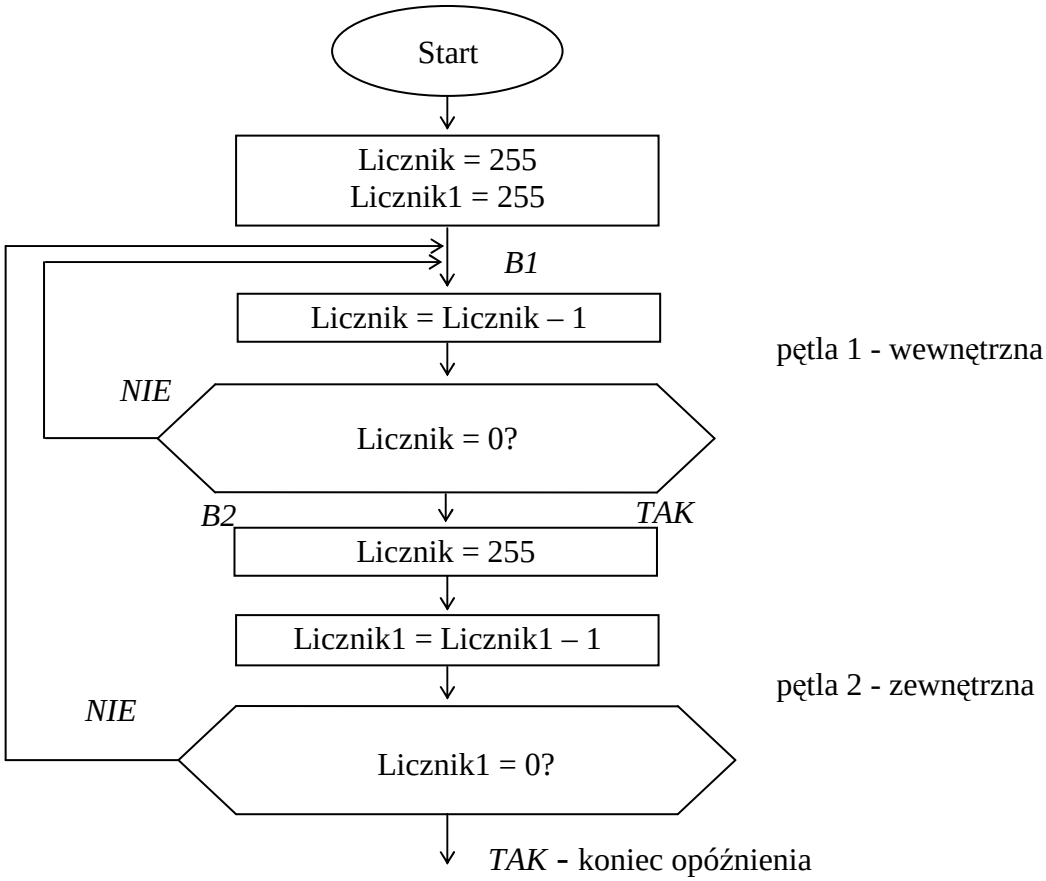
Odp.: Trochę ponad 500 μs, jeśli wartością początkową wpisaną do zmiennej *Licznik* będzie 0. W takim przypadku przed pierwszym testowaniem stanu zmiennej *Licznik,* jej zawartość zostanie zmieniona na 255 i w konsekwencji dopiero po jej dalszym 255-krotnym zmniejszeniu nastąpi zakończenie opóźnienia. Jeśli natomiast zamiast 0 wpiszemy na początku do zmiennej *Licznik* wartość 255, otrzymamy minimalnie krótsze opóźnienie (zmienna zostanie zmniejszona w sumie 255 razy).

Uwaga*:* Jeśli zmienna 8-bitowa jest równa zero, to w wyniku jej dekrementacji (zmniejszenia o jeden) instrukcją *djnz ad, d* wartość zmiennej przyjmie maksymalną wartość 8-bitową, czyli 255.

Pytanie 2.: Jak uzyskać większe opóźnienia?

Odp.: Na przykład przez stworzenie 2 lub więcej takich pętli – *pętla w pętli*. Inne sposoby tworzenia opóźnień, np. przy wykorzystaniu układu czasowo-licznikowego, zostaną podane w dalszej części opracowania.

Na rysunku 15.2 przedstawiono algorytm z dwoma pętlami.



Rys. 15.2. Algorytm do realizacji większego opóźnienia (pętla w pętli)

Po zakończeniu realizacji pętli 1(etykieta *B2*), która trwa ok. 255*2 μs (przyjmijmy dla uproszczenia 500 μs), do zmiennej *Licznik* ponownie zostaje wpisana wartość początkowa i zmienna *Licznik1* jest dekrementowana. Jeśli jej wartość jest różna od zera, następuje powrót do miejsca o etykiecie *B1* i pętla 1 jest realizowana ponownie. W konsekwencji do miejsca o etykiecie *B2* program „wchodzi" co ok. 500 μs (czas realizacji pętli 1), zaś pętla 2 zostaje wykonana 254 razy. Otrzymamy więc całkowite opóźnienie ok. 255*500μs ≈ 125 ms.

Realizacja programowa opóźnienia na podstawie powyższego algorytmu jest następująca:

```
; początek bloku opóźniającego
        mov Licznik, #255          ; wartość początkowa
        mov Licznik1, #255         ; wartość początkowa
B1:
        djnz Licznik, B1
B2:
        mov Licznik, #255          ; ponowne wpisanie wartości początkowej
        djnz Licznik1, B1
; koniec bloku opóźniającego
```

Uwaga: W algorytmie na rys. 15.2 etykieta *B2* nie jest konieczna.

16. KOD ŹRÓDŁOWY PROGRAMU *przes_a*

Poniżej podana jest treść programu ***przes_a*** realizującego ciągłe świecenie lub „przesuw" świecących diod w układzie z rys. 9.1 na podstawie algorytmu z rys. 11.1 oraz zarysu programu przedstawionego w p. 13.

```
;*********************************************************************
; Program   przes_a.s03
;*********************************************************************
;  W zaleznosci od stanu wejsc P3.4 i P3.5 nastepuje:
;  - stabilne swiecenie dwu diod,
;  - swiecenie jednej lub dwu diod z przesuwem w prawo,
;  - swiecenie jednej lub dwu diod z przesuwem w lewo.
;---------------------------------------------------------------------
; Teksty zaczynajace sie od srednika sa to komentarze i sa ignorowane
; przez kompilator
;*********************************************************************
; Poczatek programu od adresu 0000h
;*********************************************************************
; Swieca diody D1 i D4, pozostale zgaszone
     clr   P1.1          ; zapalenie diody D1
     clr   P1.4          ; zapalenie diody D4
     setb  P1.2          ; zgaszenie diody D2
     setb  P1.3          ; zgaszenie diody D3
     setb  P1.5          ; zgaszenie diody D5
     setb  P1.6          ; zgaszenie diody D6

;*************************************************************************
; POCZATEK PETLI GLOWNEJ PROGRAMU
; Sprawdzanie, czy ma byc przesuw
A1:
     jnb P3.4, A2

;---------------------------------------------------------------------
; Nie ma przesuwu - swieca diody 1 i 4, pozostale zgaszone
A3:
     mov P1, #0EDh      ; zapalenie i zgaszenie diod jedna instrukcja
     sjmp A1

;*********************************************************************
; Jest przesuw - testowanie kierunku przesuwu
A2:
     jb P3.5, A4

;---------------------------------------------------------------------
; Przesuw  w  prawo
A5:
     mov A, P1
     rr A
     mov P1, A
     sjmp A6
```

```
;---------------------------------------------------------------------
; Przesuw w lewo
A4:
     mov A, P1
     rl A
     mov P1, A

;*********************************************************************
;   Blok opozniajacy - ok. 0.5 sek.
;---------------------------------------------------------------------
; Wpis wartosci poczatkowych do komorek pomocniczych (licznikow opozn.)
A6:
     mov 30h, #255          ; licznik opozn. 1
     mov 31h, #255          ; licznik opozn. 2
     mov 32h, #4            ; licznik opozn. 3
; Pierwsza petla opozniajaca (ok. 500 mikrosekund)
B1:
     djnz 30h, B1

; Skok w to miejsce co ok. 500 mikrosekund
     mov 30h, #255  ; ponowne wpisanie wartosci poczatk. do licznika opozn.1
; Druga petla opozniajaca
     djnz 31h, B1

; Skok w to miejsce co ok. 0.125 sekundy
     mov 31h, #255  ; ponowne wpisanie wartosci poczatk. do licznika opozn.2

; Trzecia petla opozniajaca
     djnz 32h, B1
; Skok w to miejsce co ok. 0.5 sekundy  - koniec opoznienia
;---------------------------------------------------------------------
     sjmp A1
; Koniec programu
     end        ; ta instrukcja ZAWSZE konieczna na koncu programu
;---------------------------------------------------------------------
```

Taki program należy przygotować **w dowolnym pliku tekstowym**. Wszystkie użyte w programie rozkazy mikrokontrolera zostały podane **czcionką pogrubioną,** celem ich wyróżnienia.

Tego typu program nazywany jest **programem źródłowym**. Co należy zrobić, aby przekształcić go do postaci „zrozumiałej” dla mikrokontrolera, zostanie opisane w dalszej części opracowania.

Jak już wcześniej wspomniano, wszystkie teksty w danym wierszu na prawo od średnika traktowane są jako komentarz i nie będą uwzględniane podczas przekształcania naszego programu na postać „zrozumiałą” dla mikrokontrolera. Zaleca się stosowanie obszernych komentarzy, gdyż dobrze napisany komentarz bardzo ułatwia zrozumienie programu innym

osobom. Także programista, który podczas pisania jest „w transie" i wszystko rozumie, po powrocie do programu po dłuższej przerwie może się długo zastanawiać nad tym, „co też autor chciał przez to powiedzieć".

Opóźnienie zrealizowano stosując 3 pętle i uzyskując czas opóźnienia ok. 0,5 s. Jako zmienne pomocnicze zastosowano 3 komórki: o adresach *30h, 31h i 32h*. Mimo że algorytm opóźnienia zastosowanego w programie jest nieco bardziej rozbudowany, niż algorytm przedstawiony na rys. 15.2, można poczynić pewne analogie: użyta w programie komórka *30h* odpowiada zmiennej *Licznik* z algorytmu, zaś komórka *31h* - zmiennej *Licznik1*. O tym, gdzie te komórki się znajdują (wewnętrzna pamięć *RAM* mikrokontrolera) i jak można z nich korzystać, będzie mowa w dalszej części materiałów.

Krótkie opóźnienia można realizować przy użyciu instrukcji ***nop*** (***n****o* ***op****eration*). Ta instrukcja (jest to **instrukcja bezargumentowa**) nic nie „robi", tylko zajmuje czas, czyli generuje opóźnienie. Przy przyjęciu czasu trwania jednego cyklu maszynowego równego 1 μs, za pomocą pojedynczej instrukcji *nop* uzyskuje się opóźnienie wynoszące właśnie 1 μs.

Uwaga: Nazwy instrukcji można pisać zarówno małymi, jak i dużymi literami.

17. PAMIĘĆ SFR – obszar rejestrów specjalnych

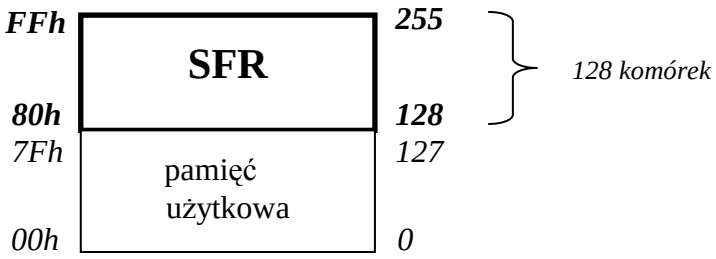
Wymieniając niektóre rejestry podano wcześniej (np. p.8.3), że znajdują się one w tzw. obszarze ***SFR***. Celem wyjaśnienia, co to za obszar, należy krótko omówić wewnętrzną pamięć danych mikrokontrolera.

Każdy mikrokontroler omawianej rodziny 8051 posiada **wewnętrzną pamięć danych** (ang. *internal data memory*) zawierającą 256 słów 8-bitowych o adresach od *0* do *255* (szesnastkowo od *00h* do *FFh*). Większość instrukcji wykonujących operacje na bajtach operuje właśnie na komórkach z tego obszaru. Pamięć tę przedstawiono na rys. 8.1 jako *Pamięć danych RAM*.

Wewnętrzną pamięć danych mikrokontrolera można podzielić na dwa obszary (rys. 17.1):

a) **pamięć użytkowa** o adresach *00h ÷ 7Fh* (pamięć tę przedstawiono na rys. 8.5 jako *Pamięć danych 128 RAM*),

b) **obszar rejestrów specjalnych *SFR*** (ang. ***S**pecial **F**unction **R**egisters*) o adresach od ***128*** do ***255*** (szesnastkowo od ***80h*** do ***FFh***). Pamięć ta na rysunku 8.5 nie jest przedstawiona jako oddzielny blok, natomiast podane są poszczególne rejestry w niej zawarte.



Rys. 17.1. Wewnętrzna pamięć danych mikrokontrolera

Na rysunku 17.2 przedstawiono tzw. **mapę pamięci** obszaru *SFR*, na której podane są nazwy rejestrów oraz ich adresy w pamięci. Każdy prostokącik oznacza jeden rejestr 8-bitowy, czyli jedną komórkę pamięci. Każdy rejestr posiada oddzielny adres. Przykładowo, rejestr *P0* (rejestr portu *P0*) znajdujący się na rysunku w lewym dolnym rogu, ma adres *80H*, rejestr *SP* znajdujący się na prawo od niego – adres *81H*, zaś rejestr w prawym dolnym rogu (*PCON*) – adres *87H*.

Pytanie: Gdzie znajdują się rejestry *P1* i *P3*, które są wykorzystywane w naszym programie?
Odp.: Wszystkie rejestry odpowiadające **portom mikrokontrolera**, czyli rejestry *P0 ÷ P3* znajdują się omawianym obszarze *SFR*. Rejestr portu *P1* ma adres *90H*, a rejestr portu *P3* – adres *0B0H*.

Rejestry Specjalne SFR (*8052/32*):

0F8H								
0F0H	B							
0E8H								
0E0H	ACC							
0D8H								
0D0H	PSW							
0C8H	*T2CON*		*RCAP2L*	*RCAP2H*	*TL2*	*TH2*		
0C0H								
0B8H	IP							
0B0H	P3							
0A8H	IE							
0A0H	P2							
98H	SCON	SBUF						
90H	P1							
88H	TCON	TMOD	TL0	TL1	TH0	TH1		
80H	P0	SP	DPL	DPH				PCON

↑ rejestry adresowane bitowo i bajtowo

Rys. 17.2. Obszar rejestrów specjalnych SFR [2]

W obszarze *SFR* znajdują się m. in. także akumulator (*ACC*), rejestr *B* i wymieniony wcześniej rejestr *PSW*. Tu umieszczone są także inne rejestry, odpowiedzialne za pracę różnych bloków funkcjonalnych, np. układu czasowo-licznikowego, systemu przerwań, itp. Są to rejestry, które znajdują się na przedstawionym wcześniej schemacie blokowym mikrokontrolera (rys.8.5).

Pytanie: Co oznaczają liczby umieszczone z lewej strony opisywanych rejestrów: liczba *D0H* przy rejestrze *PSW* przedstawionym w punkcie 8.3d na s.23, oraz liczba *90H* przy rejestrze *P1* (s.27, p. 10d)?
Odp.: Liczby te są właśnie adresami omawianych rejestrów w obszarze *SFR* wewnętrznej pamięci danych mikrokontrolera (porównaj z rys. 17.2).

Na mapie obszaru *SFR* (rys. 17.2) widać wiele wolnych miejsc (adresów). Tych adresów niestety **nie da się** wykorzystać jako **dodatkowe** komórki pamięci. Wiele z nich zostaje zapełnionych przez dodatkowe rejestry w różnych nowszych wersjach mikrokontrolerów, ale w omawianej przez nas wersji podstawowej te adresy są „nieczynne"; jest tak, jakby ich w ogóle nie było.

***Uwaga**:* Podczas pisania programu można używać bezpośrednio **adresów** komórek z pamięci *SFR* **zamiast** ich **symboli**, np.

- zamiast pisać ***mov A, P1***, można napisać ***mov A, 90H***,
- zamiast pisać ***mov P3, B***, można napisać ***mov 0B0H, B*** lub nawet ***mov 0B0H, 0F0H*** (*0F0H* to adres rejestru *B*),

lecz jednak o wiele wygodniej jest używać nazw rejestrów, gdyż są one o wiele łatwiejsze do zapamiętania, niż ich adresy, a także i program jest wtedy bardziej czytelny.

Przypomnienie: Poszczególne bity rejestrów *P0÷P3* odpowiadają liniom portów z rys. 10.1. Jeśli np. użyjemy instrukcji *setb P1.1*, to stan *1* pojawi się na wejściu *D* przerzutnika odpowiadającego linii *P1.1*, zostanie przepisany na wyjście *Q* tegoż przerzutnika (rys. 10.1b) i zapamiętany. Instrukcja *mov C, P1.1* (odczyt ***końcówki!***) spowoduje skopiowanie stanu *wyprowadzenia P1.1* mikrokontrolera do znacznika *CY*, zaś instrukcja *jbc P1.1, d* (odczyt ***wyjścia przerzutnika!***) odczyta stan wyjścia *Q* przerzutnika odpowiadającego linii *P1.1*, wyzeruje go (stan *0* pojawi się na wejściu *D* przerzutnika, zostanie przepisany na jego wyjście *Q* i zapamiętany) oraz ewentualnie zrealizuje skok w inne miejsce programu – porównaj p. 12.4.4.

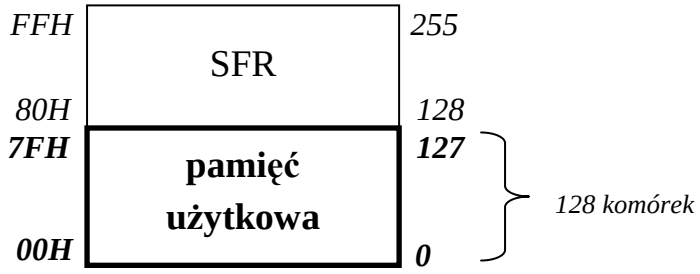
Uwaga: Rejestry portów *P0÷P3* z punktu widzenia programisty traktujemy tak samo, jak każdy inny rejestr adresowalny bitowo.

18. PAMIĘĆ UŻYTKOWA

Pytanie: Przy realizacji opóźnienia w naszym programie ***przes_a*** wykorzystujemy komórki pamięci o adresach *30h*, *31h* i *32h*. Gdzie te komórki się znajdują?

Odp.: Komórki te znajdują się w obszarze ***pamięci użytkowej*** w wewnętrznej pamięci danych mikrokontrolera.

Pamięć użytkowa to fragment wspomnianej w poprzednim punkcie wewnętrznej pamięci danych mikrokontrolera. Pamięć użytkowa zajmuje w niej komórki o adresach od ***0*** **do** ***127*** (szesnastkowo od ***00h*** **do** ***7Fh***) – rys. 18.1 (porównaj z rys. 17.1).

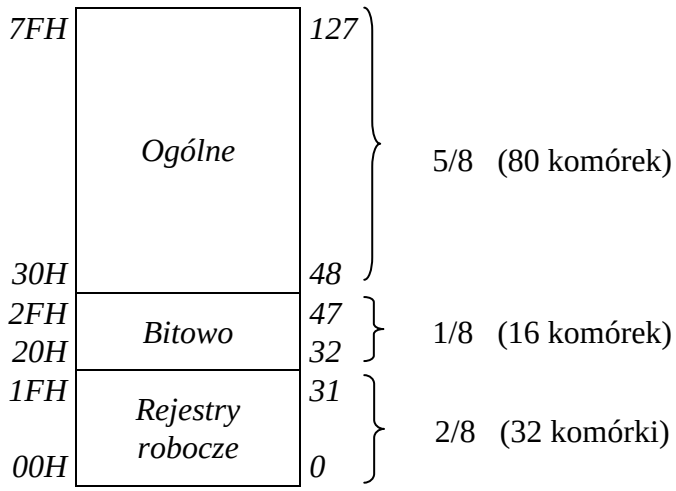


Rys. 18.1. Wewnętrzna pamięć danych mikrokontrolera

Dla przypomnienia*:* Pamięć użytkową przedstawiono na rys. 8.5 jako *Pamięć danych 128 RAM* oraz stanowi ona część *Pamięci danych RAM* z rys. 8.1.

Komórki pamięci użytkowej (128 bajtów) można podzielić na **3 obszary**:

- obszar rejestrów roboczych (*0 ... 1FH*, dziesiętnie *0 ... 31*) - 32 bajty,
- obszar adresowany bitowo (*20H ... 2FH*, dziesiętnie *32 ... 47*) - 16 bajtów (128 bitów),
- obszar ogólnego stosowania (*30H ... 7FH*, dziesiętnie *48 ...127*) - 80 bajtów.



Rys. 18.2. Pamięć użytkowa – poszczególne obszary w tej samej skali

Na rysunku 18.3 pokazano szczegółowo mapę pamięci użytkowej, przy czym nie jest tu zachowana skala – dokładnie rozrysowane są komórki z dwu pierwszych obszarów (adresy *00h ... 2Fh*), przy czym w komórkach z obszaru *20H ... 2FH* wyszczególnione są pojedyncze bity.

Poniżej opisano dokładniej poszczególne obszary pamięci użytkowej.

18.1. Obszar rejestrów roboczych – komórki ***0 ... 31*** (***00H ... 1FH***) - 32 bajty

Komórki te, zgodnie z rys. 18.3, można podzielić na 4 grupy po 8 bajtów zwane **bankami rejestrów** *RB0 ... RB3*, każdy po 8 **uniwersalnych rejestrów roboczych** (ang. *working registers*). Rejestry te nazywają się *R0, R1, ... , R7*. W danym momencie aktywny jest **tylko** jeden bank rejestrów. Po załączeniu mikrokontrolera lub jego wyzerowaniu aktywny jest bank ***RB0*** i w tym przypadku rejestr *R0* to komórka *0*, rejestr *R1* – komórka *1*, ... oraz rejestr *R7* – komórka *7*). Jeśli w programie korzystamy tylko z jednego banku rejestrów, to pozostałe komórki z omawianego obszaru mogą być użyte do innych celów. Jeśli więc aktywny jest bank *RB0*, to komórki od *08h* do *1Fh* można wykorzystać do innych celów.

Omawiane rejestry stosuje się często jako pomocnicze komórki pamięci, np. do chwilowego przechowania jakiejś wartości. I tak przykładowo, instrukcja ***mov R1, P1*** powoduje zapamiętanie w rejestrze *R1* stanu portu *P1*, zaś instrukcja ***mov R3, A*** – skopiowanie zawartości akumulatora do rejestru *R3*.

18.2. Obszar adresowalny bitowo - komórki ***32 ... 47*** (***20H ... 2FH***) – **128 bitów** (16 bajtów)

18.2.1. Podstawowe informacje

W wielu instrukcjach podaje się adres elementu pamięci **(rejestru (komórki)** lub **bitu),** który chce się zmodyfikować, odczytać bądź przetestować, np.:

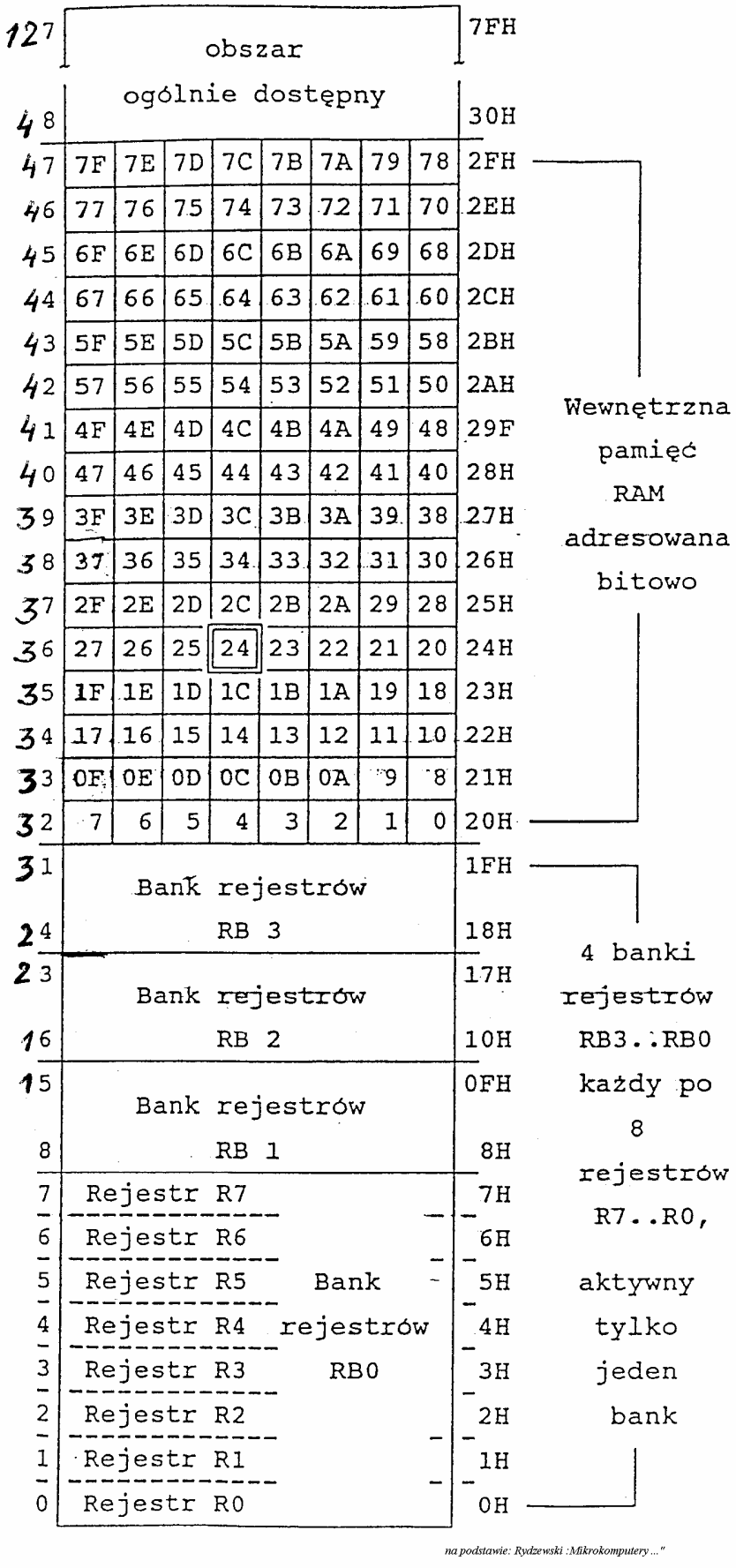
mov A, 30H,
setb 30H.

Pytanie: Jakich fragmentów pamięci dotyczą te instrukcje?

Odp.: *mov A, 30H* – kopiuj zawartość **komórki (**bajt**)** o adresie ***30H*** do akumulatora,
setb 30H – ustaw **bit** o adresie ***30H*.**

Przypomnienie: Każda instrukcja wykonuje operacje albo na **bitach** albo na **bajtach**, ale nigdy równocześnie.

Pytanie: Wiemy już, gdzie znajdują się komórki pamięci o adresach z zakresu *00h ... FFh*. Ale gdzie są **bity** o takich adresach?



Rys. 18.3. Młodsza część wewnętrznej pamięci RAM (pamięć użytkowa) [2]

Odp.: Bity o adresach ***0 ... 127*** (***00H ... 7FH***) znajdują się właśnie w omawianym obszarze komórek *20H ... 2FH*. Przyporządkowanie bitów do komórek przedstawia rys. 18.3. Bit o adresie *00H* to najmniej znaczący (zerowy) bit komórki *20H*, możemy go też zapisać jako *20H.0*. Bit o adresie *01H* to pierwszy bit komórki *20H*, czyli *20H.1*. Bit o adresie *08H* to z kolei zerowy bit komórki *21H*, czyli *21H.0;* bit *30H* to bit *0* komórki *26H, itd.* Ostatni bit z tego zakresu, czyli bit o adresie *127* (*7FH*), to najbardziej znaczący (siódmy) bit komórki *2FH*, czyli bit *2FH.7*.

Podsumowując: każda komórka z tego zakresu zawiera 8 bitów; komórek jest 16, wiec daje to razem **128 bitów**.

Z powyższego wynika, ze zamiast pisać np. ***setb 30H*** (*adres bitu*), możemy też zapisać ***setb 26H.0*** (*adres bajta* oraz *pozycja bitu* w bajcie).

Ciekawostka: Wyróżniony na rys. 18.3 bit o adresie *24H* to czwarty bit komórki o adresie *24H*, jego ustawienia można więc dokonać instrukcją *setb 24H* lub *setb 24H.4*.

18.2.2. Zastosowanie bitów

Takie bity mogą służyć w programie jako **znaczniki** do zapamiętywania pewnych stanów, np.

- dioda ma migać → bit ustawiony,
- dioda ma być zgaszona → bit wyzerowany.

Inny przykład:

- *zmienna A* ≥ *zmienna B* → bit wyzerowany,
- *zmienna A* < *zmienna B* → bit ustawiony.

Programista ma pełną swobodę w wyborze tego, co który bit będzie oznaczał.

18.2.3. Operacje na bajtach

To, że w omawianym obszarze pamięci (adresowalnym bitowo) można dokonywać operacji na pojedynczych bitach nie oznacza wcale, że nie można tu dokonywać operacji na całych komórkach:

- można dokonywać operacji na całych komórkach, których bity **są** stosowane jako znaczniki, analogicznie jak wcześniej opisano dla rejestru *P1* (p. 12.3),
- komórki z omawianego obszaru, których bity **nie są** wykorzystywane jako znaczniki, mogą być **adresowane bajtowo** i dowolnie użyte przez programistę.

<u>UWAGA</u>: Ostrożnie z równoczesnym stosowaniem bitów oraz bajtów z omawianego obszaru!!!

Dlaczego? Patrz poniższe pytanie i odpowiedź.

<u>Pytanie</u>: Co się stanie, jeśli użyjemy w tym samym programie np. bitu *30H* w jednej instrukcji, a adresu komórki *26H* (**w której ten bit się znajduje**) w innej instrukcji do zupełnie innego celu, np.:

```
setb 30H        ; wyłączenie silnika
mov 26H, A      ; zapamiętanie prędkości obrotowej silnika
```

<u>Odp.</u>: **<u>Katastrofa</u>.** Nastąpi kolizja – bit ustawiony lub wyzerowany jedną z instrukcji może być zmodyfikowany drugą. Przykładowo instrukcja ***setb 30H*** wyłączyła silnik, zaś wpis wartości do komórki *26H* (która służy tu do przechowania zmierzonej prędkości obrotowej) instrukcją ***mov 26H, A*** może spowodować wyzerowanie bitu *30H* i **niezamierzone załączenie silnika** (**!!!**). Jakie skutki może to wywołać – łatwo sobie wyobrazić. Podczas pisania programu należy więc bardzo skrupulatnie kontrolować, które bity i komórki do czego służą.

18.3. Obszar ogólnego stosowania – komórki *48 ... 127* (*30H ... 7FH*) - 80 bajtów

Obszar ten adresowalny jest ***tylko bajtowo***, co oznacza, że poszczególne bity z tego obszaru nie są bezpośrednio dostępne. Błędne byłyby więc instrukcje np. ***setb 30H.0***, czy ***clr 7FH.1***.

Omawiany obszar służy do przechowywania danych – i nie pełni żadnych dodatkowych specjalnych funkcji poza tym, że można w nim umieścić *stos*.

<u>Uwaga</u>: Komórki pamięci ***30H***, ***31H*** i ***32H*** wykorzystane w naszym programie do generacji opóźnienia pochodzą właśnie z tego obszaru.

18.4. Stos

W obszarze pamięci użytkowej (adresy *00H ... 7FH*) należy także zarezerwować miejsce na **stos**, czyli fragment pamięci wykorzystywany do przechowywania (odbywa się to automatycznie) tego adresu z pamięci programu, do którego trzeba będzie wrócić po zakończeniu wykonywania *procedury użytkownika* (wywołanej np. instrukcją ***lcall adr16***) lub *procedury obsługi przerwania* (wymienione procedury zostaną omówione w dalszej części opracowania). Na stosie można także przechować chwilowo zawartość dowolnej – adresowanej *bezpośrednio* – komórki wewnętrznej pamięci danych: zapisać ją za pomocą instrukcji ***push ad***, zaś zdjąć ze stosu instrukcją ***pop ad***.

Ważnym pojęciem jest **wierzchołek stosu.** Jest to adres komórki, **powyżej której** będą umieszczane kolejne bajty odkładane na stos. Po załączeniu lub wyzerowaniu mikrokontrolera wierzchołek stosu to adres *07H*, co oznacza, że pierwszy bajt odłożony na stos znajdzie się w komórce *08H*, następny – w komórce *09H*, itd. Komórka *07H*, która odpowiada rejestrowi *R7* (p. 18.1) nie zostanie więc zmodyfikowana. Bieżąca wartość wierzchołka stosu przechowywana jest w rejestrze ***SP*** zwanym **wskaźnikiem stosu** (ang. ***S****tack* ***P****ointer*) i jest modyfikowana automatycznie po każdej operacji odłożenia bajta na stos lub zdjęcia bajta ze stosu. Rejestr *SP* znajduje się w obszarze *SFR* i posiada adres *81H* – rys. 17.2.

Jak wynika z powyższego, początkowa wartość zawarta we wskaźniku *SP* po uruchomieniu mikrokontrolera wynosi *07*. Przed każdym zapisaniem na stos wskaźnik *SP* jest zwiększany o 1, a po każdym odczytaniu ze stosu – zmniejszany o 1. W konsekwencji pierwsza dana zapisana na stos zostanie umieszczona w komórce *08*.

Stos można przenieść w inne miejsce wpisując nowy adres do rejestru *SP*. Realizuje to instrukcja ***mov SP, #n.*** Na przykład instrukcja *mov SP, #60H* przypisze wierzchołkowi stosu adres *60H*. Przenoszenie stosu jest czasem stosowane, ale *jedynie* w części inicjalizacyjnej programu. Programowa zmiana zawartości rejestru *SP* w chwili, gdy na stos zostało już coś odłożone spowoduje błędne działanie programu.

Nasz program *przes_a* w ogóle nie korzysta ze stosu, ale niektóre jego kolejne wersje będą stos wykorzystywać.

19. BITY W OBSZARZE *SFR*

Pytanie: Powyżej (p.18.2) omówiono bity o adresach *0 ... 127* (*00H ... 7FH*):

a) czy istnieją też bity *128 - 255* (*80H ... 0FFH*), a jeśli tak, to gdzie one są?

b) jeśli tak, to jaki będzie efekt instrukcji np. *setb 94H*?

Odp.: Bity o adresach ***128 ... 255*** (***80H ... 0FFH***), znajdują się w obszarze rejestrów specjalnych ***SFR***. Są to bity komórek z lewej kolumny tablicy z rys. 17.2, czyli komórek o adresach **podzielnych przez 8**, a więc *80H, 88H, 90H, 98H, 0A0H, 0A8H*, itd., aż do *0F0H*.

Uwaga: Kilka komórek z tego obszaru nie jest wykorzystanych.

Przyjrzyjmy się bitom rejestru *P0* – jest to najniższy adres z omawianego obszaru.

Adresy bitów		*87H*	*86H*	*85H*	*84H*	*83H*	*82H*	*81H*	*80H*	
Adres rejestru	***80H***	*P0.7*	*P0.6*	*P05*	*P0.4*	*P0.3*	*P0.2*	*P0.1*	*P0.0*	***P0***
		(MSB)							*(LSB)*	

Jak widać z powyższego, najmłodszy bit rejestru *P0*, czyli bit *P0.0* ma adres *80H*, bit *P0.1* – adres *81H*, itd., a najstarszy bit *P0.7* - adres *87H*.

Następne 8 adresów, czyli adresy od *88H* do *8FH*, przypisanych jest bitom rejestru o adresie *88H*, czyli rejestru *TCON*.

Adresy bitów		*8FH*	*8EH*	*8DH*	*8CH*	*8BH*	*8AH*	*89H*	*88H*	
Adres rejestru	***88H***	*TCON.7*	*TCON.6*	*TCON.5*	*TCON.4*	*TCON.3*	*TCON.2*	*TCON.1*	*TCON.0*	***TCON***
		(MSB)							*(LSB)*	

Kolejne 8 adresów przypada na bity rejestru *P1*, jak poniżej, itd.

Adresy bitów		*97H*	*96H*	*95H*	*94H*	*93H*	*92H*	*91H*	*90H*	
Adres rejestru	***90H***	*P1.7*	*P1.6*	*P15*	*P1.4*	*P1.3*	*P1.2*	*P1.1*	*P1.0*	***P1***
		(MSB)							*(LSB)*	

W naszym programie ***przes_a*** diodę *D1* zapalaliśmy instrukcją ***clr P1.1***. Teraz już wiadomo, że zamiast tego można zapisać ***clr 91H***, gdyż argumentem instrukcji *clr bit* jest adres bitu, a adresem bitu odpowiadającego linii *P1.1* jest *91H*.

Instrukcja ***setb 94H*** (z pytania powyżej) jest równoważna instrukcji ***setb P1.4***, spowoduje więc ustawienie czwartej linii portu *P1*, czyli bitu *P1.4*. W przypadku naszego układu wykonanie tej instrukcji spowoduje zgaszenie diody *D4*.

Zadanie: Porównać adresy najmłodszych bitów rejestrów *P0, P1, TCON* i *P3* z adresami tychże rejestrów.

Rozw.: adresy bitów *P0.0 = 80H, P1.0 = 90H, TCON.0 = 88H, P3.0 = 0B0H*

adresy rejestrów *P0 = 80H, P1 = 90H, TCON = 88H, P3 = 0B0H*

Wniosek: Adres najmniej znaczącego bitu rejestru z obszaru *SFR* jest zawsze równy bajtowemu adresowi rejestru.

Na rysunku 19.1 podano zestawienie rejestrów specjalnych z obszaru *SFR* z wyszczególnieniem pojedynczych bitów. Jak wynika z rysunku wiele bitów posiada swoje odrębne nazwy. Z nazwami bitów (znaczników) spotkaliśmy się już przy omawianiu rejestru *PSW* (p. 8.3d. s.23). Niektóre z przedstawionych rejestrów zostaną omówione bardziej szczegółowo w dalszej części opracowania.

Adres	Symbol	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
F0	**B**								
E0	**ACC**								
D0	**PSW**	CY	AC	F0	RS1	RS0	OV	–	P
B8	**IP**	–	–	–	PS	PT1	PX1	PT0	PX0
B0	**P3**	/RD	/WR	T1	T0	/INT1	/INT0	TxD	RxD
A8	**IE**	EA	–	–	ES	ET1	EX1	ET0	EX0
A0	**P2**								
99	**SBUF**								
98	**SCON**	SM0	SM1	SM2	REN	TB8	RB8	TI	RI
90	**P1**								
8D	**TH1**								
8C	**TH0**								
8B	**TL1**								
8A	**TL0**								
89	**TMOD**	GATE	C/T	M1	M0	GATE	C/T	M1	M0
			Timer 1				Timer 0		
88	**TCON**	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
87	**PCON**	SMOD	–	–	–	GF1	GF0	PD	IDL
83	**DPH**								
82	**DPL**								
81	**SP**								
80	**P0**								

Rys. 19.1. Rejestry specjalne mikrokontrolera 8051 (*źródło: Gałka*: *„Podstawy ...”*)

Uwaga:

Rejestry z obszaru *SFR* umieszczone na rys. 17.2 poza lewą kolumną (czyli rejestry, których adres nie kończy się na 0 lub 8) nie są adresowalne bitowo; niepoprawne byłyby więc instrukcje typu **~~clr TMOD.1~~** (rejestr TMOD ma adres 89H) lub **~~setb PCON.5~~** (rejestr PCON ma adres 87H), mimo że poszczególne bity tych rejestrów są wyszczególnione (rys. 19.1). Aby zmienić stan bitu w którymś z tych rejestrów, konieczne jest wykonanie operacji na całym rejestrze (porównaj p.12.3). Dodatkowe instrukcje (*orl ...*, *anl ...* i *xlr ...*), które to umożliwiają (poza omówioną już instrukcją *mov ...*) zostaną podane w dalszej części opracowania.

Poniżej podano możliwe sposoby adresowania znaczników.

Przykład 1.
Drugi (*przypominam – liczymy od zera!*) bit rejestru *IE* (adres rejestru to *A8H*) nazywa się **EX1** i jak się później dowiemy służy do załączania / wyłączania przerwań od sygnału zewnętrznego. Adres tego bitu (znacznika) to *AAH*. Załączanie przerwań można zrealizować przez ustawienie wspomnianego wyżej znacznika jedną z poniższych instrukcji:

- *setb EX1* – nazwa bitu,
- *setb 0AAH* – adres bitu,
- *setb IE.2* – nazwa rejestru z numerem bitu,
- *setb 0A8H.2* – adres rejestru z numerem bitu.

Przykład 2.
Czwarty bit rejestru *TCON* (adres rejestru to *88H*) nazywa się **TR0** i jak się później dowiemy służy do sterowania licznika / timera *T0* (bit ustawiony – licznik załączony, bit wyzerowany – licznik zatrzymany). Adres tego bitu (znacznika) to *8CH*. Uruchomienie licznika można zrealizować przez ustawienie wymienionego wyżej znacznika jedną z poniższych instrukcji:

- *setb TR0* – nazwa bitu,
- *setb 8CH* – adres bitu
- *setb TCON.4* – nazwa rejestru z numerem bitu
- *setb 88H.4* – adres rejestru z numerem bitu

Mimo, że wszystkie powyższe sposoby adresowania bitów są poprawne, używa się głównie tych, w których występują nazwy, bo łatwiej zapamiętać nazwę, niż adres.

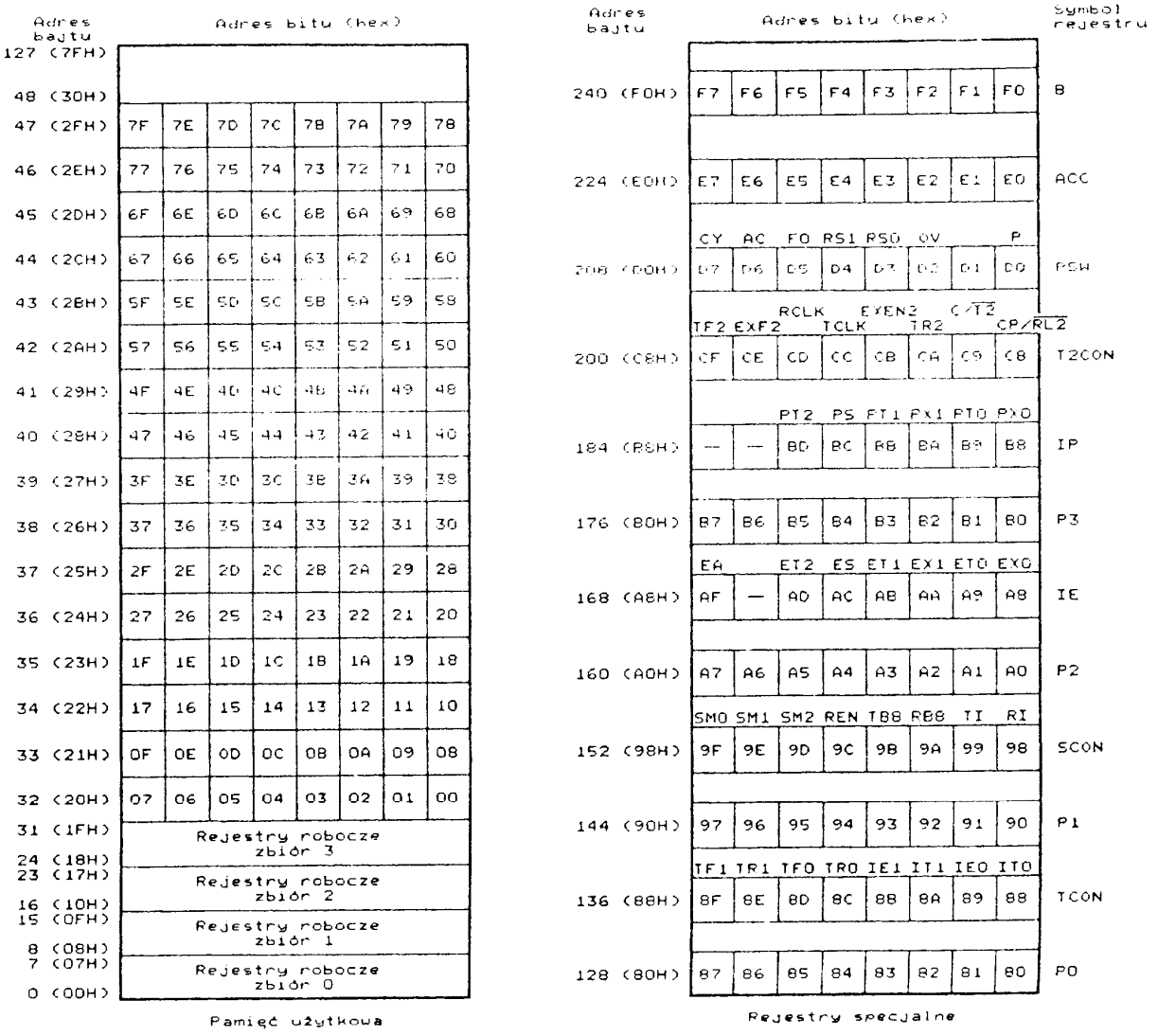
Uwaga*:* Stan dowolnego bitu można także zmieniać wykonując operację na całej komórce pamięci (całym rejestrze), w której dany bit jest umieszczony (przykład podano w p.12.3 omawiając ustawianie lub zerowanie linii portu). Ten sam efekt, co w powyższych przykładach można osiągnąć także za pomocą następujących instrukcji:

- *mov IE, #xxxxx1xxb* – wpisanie *1* do drugiego bitu rejestru *IE* (ustawienie bitu),
- *mov TCON, #xxx0xxxxb*– wpisanie *0* do czwartego bitu rejestru *TCON* (wyzerowanie bitu),

przy czym *x* oznacza dowolny ze stanów *0* oraz *1*. Różnica polega na tym, że instrukcje podane w powyższych *Przykładzie 1* i *Przykładzie 2* modyfikują tylko jeden bit, nie zmieniając stanu pozostałych bitów w rejestrze, zaś instrukcje *mov* ... wpływają na stan wszystkich bitów rejestru. Modyfikację pojedynczego bitu lub kilku bitów bez zmiany stanu pozostałych bitów komórki (rejestru) umożliwiają także wspomniane nieco wyżej instrukcje *orl* ..., *anl* ... i *xlr* ..., wykonujące operacje na całych komórkach pamięci, ...ale o tym – trochę później.

Na rysunku 19.2 podano dodatkowo zestawienie przedstawiające lokalizację wszystkich bitów o adresach z zakresu *0H ... FFH*. Podsumowując:

- adresy *0H – 7FH* (*0 – 127*) – pamięć użytkowa, komórki *20H – 2FH* ,
- adresy *80H – FFH* (*128 – 255*) – obszar rejestrów specjalnych *SFR*, komórki o adresach podzielnych przez 8 (adresy w kodzie szesnastkowym zakończone na *0H* lub *8H*).



źródło: Rydzewski: "Mikrokomputery ..."

W dalszej części opracowania zostanie omówione postępowanie prowadzące do uzyskania kodu wynikowego programu, którym będzie można zaprogramować pamięć EPROM. Zapoznamy się także w szczegółach z postacią otrzymanego kodu wynikowego. Do jego zrozumienia potrzebna nam będzie znajomość sposobu zapisywania liczb ze znakiem, co jest przedstawione w następnym punkcie.

20. LICZBY ZE ZNAKIEM - kod uzupełnień do dwóch (kod U2)

20.1. Liczby bez znaku - przypomnienie

Liczby bez znaku w zapisie dwójkowym (binarnym) i szesnastkowym (heksadecymalnym) zostały omówione w p.1 niniejszego opracowania. Przy liczbach bez znaku zakres wynosi **0...255** (**00H...FFH**) w przypadku liczb 8-bitowych. Dla przypomnienia:

	0000 0000B	00H	0
	0000 0001B	01H	1
	.....	.....	...
	1111 1110B	FEH	254
	1111 1111B	FFH	255

W przypadku liczb 16-bitowych zakres wynosi **0...65535** (**0000H...FFFFH**).

20.2. Liczby **ze znakiem** w kodzie **uzupełnień do dwóch (U2) - zestawienie**

W omawianym kodzie o znaku liczby decyduje najstarszy bit:

- 0 - liczba dodatnia,
- 1 - liczba ujemna.

a) liczby 8-bitowe ze znakiem:

0000 0000B	00H	0
0000 0001B	01H	1
.....	.....	...
0111 1111B	7FH	127
1000 0000B	80H	-128
1000 0001B	81H	-127
.....	.....	...
1111 1110B	FEH	-2
1111 1111B	FFH	-1

Tak samo, jak dla liczb bez znaku

czyli zakres **-128...+127**.

b) liczby 16-bitowe ze znakiem:

0000 0000 0000 0000B	0000H	0
0000 0000 0000 0001B	0001H	1
.....	.....	...
0111 1111 1111 1111B	7FFFH	32767
1000 0000 0000 0000B	8000H	-32768
1000 0000 0000 0001B	8001H	-32767
.....	.....	...
1111 1111 1111 1110B	FFFEH	-2
1111 1111 1111 1111B	FFFFH	-1

Tak samo, jak dla liczb bez znaku

czyli zakres **-32768...+32767**.

Uwaga*:* Liczby 8-bitowe **od 0 do 127** (w postaci dwójkowej mają 0 na pierwszej pozycji) mają dokładnie **taką samą postać**, niezależnie od tego, czy są interpretowane jako liczby ze znakiem, czy bez znaku. Podobnie jest z liczbami 16-bitowymi **od 0 do 32767**.

20.3. Obliczanie liczby przeciwnej

Zadanie1: Obliczyć, jaką postać w kodzie U2 ma liczba – 40.

Rozwiązanie: Liczba – 40 jest liczbą przeciwną do liczby 40 (ich suma wynosi 0). Liczbę przeciwną do zadanej liczby przedstawionej w postaci binarnej liczymy następująco:

- najpierw negujemy wszystkie bity liczby, bit po bicie,
- do tak powstałej liczby dodajemy 1.

a) Najpierw załóżmy, że chodzi nam o liczby 8-bitowe. Liczba 40 ma postać 0010 1000b. Po zanegowaniu wszystkich bitów otrzymamy liczbę 1101 0111b. Na zakończenie do otrzymanej liczby dodajemy 1 otrzymując 1101 1000b.

b) Załóżmy teraz, że liczby mają być 16-bitowe. Liczba 40 ma postać 0000 0000 0010 1000b. Po zanegowaniu wszystkich bitów otrzymamy liczbę 1111 1111 1101 0111b. Na zakończenie do otrzymanej liczby dodajemy 1 otrzymując 1111 1111 1101 1000b.

Odpowiedź: Liczba – 40 w kodzie U2 to:

- ***1101 1000b*** w postaci 8-bitowej,
- ***1111 1111 1101 1000b*** w postaci 16-bitowej.

Zadanie2: Podana w kodzie U2 liczba ma postać 1011 1001b. Przedstawić tę liczbę w postaci dziesiętnej.

Rozwiązanie: Jest to liczba ujemna, gdyż na pierwszej pozycji jest 1. Aby stwierdzić, jaka to liczba należy najpierw obliczyć liczbę do niej przeciwną (przykładowo, liczba przeciwną do – 40 jest 40), a następnie zamienić otrzymaną liczbę na postać dziesiętną. Po zanegowaniu wszystkich bitów zadanej liczby otrzymujemy 0100 0110b, a po dodaniu 1 – liczbę 0100 0111b. Tak otrzymaną liczbę (przeciwną do zadanej) przeliczamy na postać dziesiętną: 1+2+4+64=71.

Odpowiedź: Szukana liczba to **– 71.**

20.4. O interpretacji rodzaju liczby decyduje programista

Mikrokontroler wykonuje operacje **dodawania** i **odejmowania** dokładnie w taki sam sposób na liczbach bez znaku, jak i na liczbach ze znakiem. Można więc powiedzieć, że mikrokontroler „nie wie” z jaką postacią liczb ma wtedy do czynienia. To programista decyduje o tym, jak interpretuje te liczby. Liczby ujemne muszą być więc tak dobrane, aby operacje dodawania i odejmowania dawały prawidłowe wyniki, niezależnie od tego, czy liczby traktujemy jako liczby ze znakiem, czy bez znaku. Warunek ten spełnia kod uzupełnień do 2 (kod U2), w którym **liczba „-1” to same jedynki**. Stąd kod ten jest powszechnie stosowany do reprezentacji liczb ze znakiem.

Poniżej podano przykłady, z których wynika, dlaczego dla liczb 8-bitowych ze znakiem powinno być ***–1 = 1111 1111B***. W przykładach pominięto znacznik pożyczki, który zostaje ustawiony przy odejmowaniu oraz znacznik przeniesienia, ustawiony przy dodawaniu.

dziesiętnie	*dwójkowo*
0	0000 0000B
–1	– 0000 0001B
–1	1111 1111B
-128	1000 0000B
+127	+ 0111 1111B
–1	1111 1111B

A oto inny przykład. Weźmy dodawanie:

szesnastkowo	*dwójkowo*
90H	1001 0000B
+ 48H	+ 0100 1000B
D8H	1101 1000B

Jeśli potraktujemy te liczby jako liczby bez znaku, mamy dodawanie 144 + 72 = 216, a jeśli, jako liczby ze znakiem: –112 + 72 = – 40 (proszę sprawdzić!). Wynik jest prawidłowy przy obu interpretacjach liczb.

A teraz przykład odejmowania:

szesnastkowo	*dwójkowo*
D8H	1101 1000B
– 37H	– 0011 0111B
A1H	1010 0001B

Jeśli potraktujemy te liczby jako liczby bez znaku, mamy odejmowanie 216 – 55 = 161, a jeśli jako liczby ze znakiem: – 40 – 55 = – 95. Również tutaj wynik jest prawidłowy przy obu interpretacjach liczb.

Uwaga 1*:* W powyższych przykładach obliczenia są proste, gdyż suma i różnica liczb mieszczą się w zakresie 8–bitowym, zarówno dla liczb ze znakiem, jak i bez znaku. W ogólnym przypadku należy jednak uwzględnić możliwość przekroczenia zakresu, np.:

- liczby bez znaku: *200 + 100 = 300* – suma poza zakresem 8-bitowym dla liczb bez znaku, mimo że składniki z zakresu 8-bitowego dla liczb bez znaku,
- liczby ze znakiem – kod U2: *100 + 50 = 150* – suma poza zakresem 8-bitowym dla kodu U2, mimo że składniki z zakresu 8-bitowego dla kodu U2.

W przypadku przekroczenia zakresu 8-bitowego dla liczb bez znaku ustawiany jest znacznik przeniesienia **CY** (**PSW.7**), zaś przy przekroczeniu zakresu 8-bitowego w kodzie U2 ustawiany jest znacznik nadmiaru **OV** (**PSW.2**).

Uwaga 2: W operacjach **mnożenia i dzielenia** argumenty traktowane są jako **liczby bez znaku**.