Mikrokontrolery? To takie proste...

Część 7
Asembler – język maszynowy procesora

W kolejnym odcinku poświęconym naszym wspólnym staraniom mającym na celu ujarzmienie mikrokontrolera 8051 postaram się zapoznać Was drodzy Czytelnicy w przystępny sposób z listą instrukcji tego procesora. Na końcu tego odcinka czeka na Was druga już lekcja – czyli kolejny praktyczny krok w nauce z wykorzystaniem naszego komputerka edukacyjnego z 8051. Dziś wspólnie napiszemy i przeanalizujemy krótki ale już prawdziwie asemblerowy program

W poprzednim odcinku poznałeś już ideę tworzenia programów na mikrokontrolery 8051. Wiesz że do tego celu niezbędny jest zestaw instrukcji danego procesora (u nas jest to rodzina MCS-51, która ma wspólny język programowania) oraz znajomość kodów numerycznych poszczególnych instrukcji w przypadku kiedy nie masz dostępu do komputera i wszystkie czynności musisz wykonać ręcznie. W przypadku kiedy do dyspozycji programisty jest komputer, procedurę tłumaczenia instrukcji zapisanych jawnie – w języku asemblera – automatycznie wykonuje komputer korzystający z programu zwanego kompilatorem. Autor cyklu zadbał, aby każdy z Was drodzy Czytelnicy, niskim kosztem mógł stać się posiadaczem takiego kompilatora. Jest on dostępny na dyskietkach 3,5" w ofercie handlowej AVT pod nazwą AVT-2250/D. Ważną informacją jest to że zamieszczono tam dwie wersje kompilatora: wersję angielską oraz polską!. Jest to chyba pierwszy program tego typu komunikujący się w naszym ojczystym języku z programistą. Dzięki temu osoby nie znające angielskiego będą mogły bez problemów korzystać z takiej wersji kompilatora. Funkcjonalnie obie wersje są takie same, to znaczy że wykonują wszystkie czynności identycznie, jedynie komunikaty zgłaszane przez program występują w dwóch rożnych językach, jak wspomniałem wcześniej. Na dyskietce znajduje się plik tekstowy z rozszerzeniem .DOC, w którym zawarte są informacje o kompilatorze i jego możliwościach niezbędne do prawidłowego posługiwania się nim. Dlatego nie zbędę opisywał szczegółowo tych spraw, ponieważ wśród naszych czytelników są osoby nie mające komputera PC a poza tym każdy zainteresowany PC-towiec będzie miał sam okazję na zapoznanie się z instrukcją użytkowania programu.

Ze względu na mocno ograniczone możliwości „ręcznej" kompilacji tworzonych przez Ciebie programów do postaci maszynowej, powinieneś już teraz zastanowić się nad możliwością nabycia lub przynajmniej korzystania z komputera PC, nawet poczciwej AT czy XT. Efektywne, pozbawione niepotrzebnych pomyłek, tworzenie nawet mało skomplikowanych programów jest możliwe tylko przy wykorzystaniu komputera oraz kompilatora, który jest dostępny dla wszystkich zainteresowanych po przystępnej cenie.

W tym miejscy chcę uspokoić wszystkich antykomputerowców. Wszystkie przedstawiane w cyklu przykładowe programy będą drukowane w postaci czytelnej i jasnej także dla tego grona czytelników. Ułatwi to analizę i pokaże jak w praktyce tłumaczy się komendy asemblera na język maszynowy.

W tym odcinku szkoły mikroprocesorowej zapoznamy się z listą instrukcji procesora, oraz dodatkowo zbierzemy „w całość" wiadomości dotyczące wszystkich rejestrów specjalnych SFR – także tych nie omawianych (na razie). Wszystko to jest umieszczone dodatkowo we wkładce wewnątrz numeru w postaci kartki A4 z nadrukowanymi dwustronnie skrótowo wszysktimi informacjami niezbędnymi do rozpoczęcie pisania własnych programów oraz ich tłumaczenia (asemblacji) w przypadku osób które musza to zrobić ręcznie. Taka „ściąga" powinna być przez Ciebie drogi Czytelniku wycięta a następnie zafoliowana, by mogła ci służyć przez cały czas zabawy z procesorem 8051. Zawieszenie jej na ścianie nad twoim biurkiem z pewnością ułatwi Ci poznanie i zapamiętanie wszystkich instrukcji procesora, tak abyś w przyszłości mógł władać asemblerem tak ja własnym ojczystym językiem – gwarantuję Ci – jest to możliwe!

Przejdźmy zatem do zapoznania się i wyjaśnienia działania wszystkich poleceń kontrolerów 8051.

Lista instrukcji

Informacje zawarte w tej części artykułu są rozszerzeniem listy przedstawionej we wkładce wewnątrz numeru. Dlatego analizują opisy poszczególnych instrukcji powinieneś mieć także przed oczyma wspomnianą „ściągę". Kiedy w przyszłości nabierzesz nieco wprawy w posługiwaniu się poleceniami asemblera, potrzebna będzie Ci tylko strona z wkładki,

a do niniejszego opisu będzie mógł zawsze wrócić w przypadku niejasności, szczególnie wtedy jeżeli będziesz chciał tworzyć programy nie mając dostępu do komputera PC. Tak więc zaczynamy.

Opis każdej instrukcji składa się zasadniczo z następujących elementów:

– nazwy angielskiej i polskiej instrukcji: pkt. a)
– krótkiego opisu działania instrukcji: pkt. b)
– wyszczególnienia znaczników na które działa instrukcja: pkt. c)
– opisu szczegółowego instrukcji lub jej rodzajów: pkt. d), wraz z podaniem formatu i kodów maszynowych instrukcji, w zapisanych binarnie i heksadecymalnie, z podaniem ilości cykli i bajtów kodu oraz ewentualnie poparte przykładem lub uwagami dotyczącymi efektów użycia danej instrukcji.

Większość z tych informacji znajduje się także w tabeli zestawieniowej instrukcji we wkładce wewnątrz numeru.

Operacje arytmetyczne

"ADD"

a) ang. „add to acummulator" – dodaj do akumulatora
b) Do wartości przechowywanej w akumulatorze dodawany jest wskazany argument, a wynik zostaje wpisany do akumulatora.
c) znaczniki: C, AC i OV
d) rodzaje instrukcji:

– **ADD A , Rn**
do akumulatora dodawana jest zawartość rejestru Rn
A <- A + Rn gdzie Rn = R0...R7 (jeden z rejestrów roboczych)
kod: 0 0 1 0 1 n2 n1 n0, gdzie n2...n0 – wskazują na R0...7 stąd: 28h-2Fh
cykle: 1 bajty: 1
przykład: ADD A , R2

– **ADD A , adres**
do akumulatora dodawana jest zawartość komórki w wewn. RAM
o adresie „adres"
A <- A + (adres)
kod: 0 0 1 0 0 1 0 1 25h
cykle: 1
bajty: 2 (kod instrukcji 25h + adres)
przykład: ADD A , 2Fh (dodanie do A zawartości komórki o adresie 2Fh)

– **ADD A , @Ri**
do akumulatora dodawana jest zawartość komórki w wewn. RAM
o adresie wskazywanym przez rejestr Ri (R0 lub R1)
A <- A + (Ri).
kod: 0 0 1 0 0 1 1 i gdzie i wskazuje na R0 (i=0) lub R1 (i=1) stąd: 26h, 27h
cykle: 1 bajty: 1
przykład: ADD A , @R0 (dodanie do A zawartości komórki o adresie w R0)

– **ADD A , #dana**
do akumulatora dodawany jest argument stały (8-bitowa liczba)
A <- A + dana
kod: 0 0 1 0 0 1 0 0
cykle: 1
bajty: 2 (kod instrukcji + dana)
przykład: ADD A , #120 (dodanie do A liczby 120)

"ADDC"

a) ang. „add to acummulator with carry" – dodaj do akumulatora z przeniesieniem
b) Do wartości przechowywanej w akumulatorze dodawany jest wskazany argument oraz zawartość znacznika przeniesienia C, a wynik zostaje wpisany do akumulatora.
c) znaczniki: C, AC i OV
d) rodzaje instrukcji:

– **ADDC A , Rn**
do akumulatora dodawana jest zawartość rejestru Rn oraz C
A <- A + Rn + C gdzie Rn = R0...R7 (jeden z rejestrów roboczych)
kod: 0 0 1 1 1 n2 n1 n0 gdzie n2...n0 – wskazują na R0...7 stąd: 38h-3Fh
cykle: 1 bajty: 1
przykład: ADDC A , R4

– **ADDC A , adres**
do akumulatora dodawana jest zawartość komórki w wewn. RAM
o adresie „adres" oraz znacznik C
A <- A + (adres) + C
kod: 0 0 1 1 0 1 0 1 35h
cykle: 1
bajty: 2 (kod instrukcji 35h + adres)
przykład: ADDC A , 7Eh (dodanie do A zawartości komórki o adresie 7Eh)

– **ADDC A , @Ri**
do akumulatora dodawana jest zawartość komórki w wewn. RAM
o adresie wskazywanym przez rejestr Ri. (R0 lub R1) oraz C
A <- A + (Ri) + C
kod: 0 0 1 1 0 1 1 i gdzie i wskazuje na R0 (i=0) lub R1 (i=1) stąd: 36h, 37h
cykle: 1 bajty: 1
przykład: ADDC A , @R1 (dodanie do A zawartości komórki o adresie w R1)

– **ADDC A , #dana**
do akumulatora dodawany jest argument stały (8-bitowa liczba) i C
A <- A + dana + C
kod: 0 0 1 1 0 1 0 0
cykle: 1
bajty: 2 (kod instrukcji + dana)
przykład: ADDC A , #120 (dodanie do A liczby 120)

"SUBB"

a) ang. „substract from acummulator with borrow" – odejmij od akumulatora z pożyczką
b) Od wartości przechowywanej w akumulatorze odejmowany jest wskazany argument oraz zawartość znacznika przeniesienia C, a wynik zostaje wpisany do akumulatora.
c) znaczniki: C, AC i OV
d) rodzaje instrukcji:

– **SUBB A , Rn**
od akumulatora odejmowana jest zawartość rejestru Rn oraz C
A <- A – Rn – C gdzie Rn = R0...R7 (jeden z rejestrów roboczych)
kod: 1 0 0 1 1 n2 n1 n0 gdzie n2...n0 – wskazują na R0...7 stąd: 98h-9Fh
cykle: 1 bajty: 1
przykład: SUBB A , R6

– **SUBB A , adres**
od akumulatora odejmowana jest zawartość komórki w wewn. RAM
o adresie „adres" oraz znacznik C
A <- A – (adres) – C
kod: 1 0 0 1 0 1 0 1 95h
cykle: 1
bajty: 2 (kod instrukcji 95h + adres)
przykład: SUBB A , 45h (odjęcie od A zawartości komórki o adresie 45h i znacznika C)

– **SUBB A , @Ri**
od akumulatora odejmowana jest zawartość komórki w wewn. RAM
o adresie wskazywanym przez rejestr Ri. (R0 lub R1) oraz C
A <- A – (Ri) – C
kod: 1 0 0 1 0 1 1 i gdzie i wskazuje na R0 (i=0) lub R1 (i=1) stąd: 96h, 97h
cykle: 1 bajty: 1
przykład: SUBB A , @R1 (odjęcie od A zawartości komórki o adresie w R1oraz C)

– **SUBB A , #dana**
od akumulatora odejmowany jest argument stały (8-bitowa liczba) oraz C
A <- A – dana – C
kod: 1 0 0 1 0 1 0 0 94h
cykle: 1
bajty: 2 (kod instrukcji + dana)
przykład: ADDC A , #86h (odjęcie od A liczby 86h i znacznika C)

"INC"

a) ang. „increment" – zwiększenie o 1
b) do wskazanego argumentu jest dodawana jedynka
c) znaczniki: bez zmian
d) rodzaje instrukcji:

– **INC A**
do akumulatora dodawana jest jedynka
A <- A + 1
kod: 0 0 0 0 0 1 0 0 04h
cykle: 1 bajty: 1

– **INC Rn**
do zawartości rejestru Rn dodawana jest jedynka
Rn <- Rn + 1 gdzie Rn = R0...R7 (jeden z rejestrów roboczych)
kod: 0 0 0 0 1 n2 n1 n0 gdzie n2...n0 – wskazują na R0...7 stąd: 08h-0Fh
cykle: 1 bajty: 1
przykład: INC R3

– **INC adres**
do zawartości komórki o adresie „adres" dodawana jest jedynka
(adres) <- (adres) + 1
kod: 0 0 0 0 0 1 0 1 05h
cykle: 1
bajty: 2 (kod instrukcji 05h + adres)
przykład: INC 12h (inkrementacja zawartości komórki o adresie 12h)

– **INC @Ri**
do zawartości komórki o adresie wskazywanym przez Ri dodawana jest jedynka
(Ri) <- (Ri) + 1
kod: 0 0 0 0 0 1 1 i gdzie i wskazuje na R0 (i=0) lub R1 (i=1) stąd: 06h, 07h
cykle: 1 bajty: 1
przykład: INC @R1

– **INC DPTR**
do 16-bitowego wskaźnika danych DPTR złożonego z rejestrów SFR: DPH (bardziej znaczący bajt) i DPL (mniej znaczący bajt) dodawana jest jedynka.
Znaczniki nie ulegają zmianie.
DPTR <- DPTR + 1
kod: 1 0 1 0 0 0 1 1 A3h
cykle: 2 bajty: 1

"DEC"

a) ang. „decrement" – zmniejszenie o 1
b) od wskazanego argumentu odejmowana jest jedynka
c) znaczniki: bez zmian
d) rodzaje instrukcji:

– **DEC A**
od akumulatora odejmowana jest jedynka
A <- A – 1
kod: 0 0 0 1 0 1 0 0 14h
cykle: 1 bajty: 1

– **DEC Rn**
od zawartości rejestru Rn odejmowana jest jedynka
Rn <- Rn – 1 gdzie Rn = R0...R7 (jeden z rejestrów roboczych)

kod: 0 0 0 1 1 n2 n1 n0 gdzie n2...n0 – wskazują na R0...7 stąd: 18h-1Fh
cykle: 1 bajty: 1
przykład: DEC R5

– **DEC adres**
od zawartości komórki o adresie „adres" odejmowana jest jedynka
(adres) <- (adres) – 1
kod: 0 0 0 1 0 1 0 1 15h
cykle: 1
bajty: 2 (kod instrukcji 15h + adres)
przykład: DEC 3Fh (inkrementacja zawartości komórki o adresie 3Fh)

– **DEC @Ri**
od zawartości komórki o adresie wskazywanym przez Ri odejmowana jest jedynka
(Ri) <- (Ri) – 1
kod: 0 0 0 1 0 1 1 i gdzie i wskazuje na R0 (i=0) lub R1 (i=1) stąd: 16h, 17h
cykle: 1 bajty: 1
przykład: DEC @R0

"MUL AB"

a) ang. „multiply" – pomnóż
b) 8-bitowa liczba bez znaku znajdująca się w akumulatorze jest mnożona przez 8-bitową liczbę bez znaku z rejestru B. 16-bitowy wynik wpisywany jest do rejestrów B i A (bardziej znaczący bajt do B, mniej znaczący bajt do A)
c) znaczniki: jeśli wynik mnożenia jest > 255 to ustawiany jest znacznik OV, w przeciwnym razie OV jest zerowany. znacznik C jest zerowany.
d) B.A <- A x B
kod: 1 0 1 0 0 1 0 0 A4h
cykle: 4 bajty: 1

"DIV AB"

a) ang. „divide" – podziel
b) 8-bitowa liczba bez znaku, znajdująca się w akumulatorze jest dzielona przez 8-bitową liczbę z rejestru B. Cześć całkowita ilorazu wpisywana jest do akumulatora, a reszta z dzielenia do rejestru B. W przypadku gdy dzielnik jest równy 0 (B=0) to po wykonaniu operacji zawartość akumulatorze i rejestru B jest nieokreślona oraz dodatkowo ustawiony zostaje znacznik OV.
c) znaczniki: C = 0, OV =0 (zerowane)
d) A <- A : B B <- reszta (A : B)
kod: 1 0 0 0 0 1 0 0 84h
cykle: 4 bajty: 1

"DA A"

a) ang. „decimal adjust" – korekcja dziesiętna
b) wykonywana jest korekcja dziesiętna wyniku dodawania. Operacja ta sprowadza wynik do postaci dwóch cyfr dziesiętnych w kodzie BCD, jeżeli argumenty były w kodzie BCD. Rozkaz ten powinien być używany jedynie w połączeniu z rozkazem dodawania (ADD, ADDC). Także inkrementacja powinna odbywać się poprzez instrukcję ADD A, #1, a nie INC A, bowiem w tym drugim przypadku nie są ustawianie znaczniki C i AC, tak więc nie może być wykonana korekcja dziesiętna. Korekcja polega na tym że w przypadku kiedy po wykonanej na akumulatorze operacji dodawania (ADD, ADDC) zawartość jego bitów 3...0 jest większa od 9 lub jest ustawiony znacznik AC, to do wartości akumulatora dodawana jest liczba 6. Po tym jeżeli okaże się że zawartość bitów 7...4 jest większa od 9 lub jest ustawiony znacznik C to do tych bitów dodawana jest także liczba 6. Jeżeli podczas tej ostatniej operacji wystąpiło przeniesienie to do znacznika wpisywana jest 1, w przeciwnym wypadku stan znacznika C nie zmienia się.
c) znaczniki: C, OV (patrz pkt.a)
d) A <- korekcja dziesiętna (A)
kod: 1 1 0 1 0 1 0 0 D4h
cykle: 1 bajty: 1
Przykład: jeżeli w wyniku dodawania w akumulatorze jest liczba 6Ah, to po korekcji dziesiętnej akumulator będzie zawierał liczbę 70h, patrz listing:

```
MOV   A,#69h  ;wpisanie liczby 69h
              do akumulatora     (1)
ADD   A,#1    ;inkrementacja
              akumulatora poprzez
              dodawanie          (2)
              ;w wyniku tego
              w A będzie liczba 6Ah
da    A       ;korekcja dziesiętna A,
              w A będzie po tym
              70h                (3).
```

Uwaga: jeżeli w przykładzie w linii (2) użyjemy instrukcji INC A zamiast dodania jedyni, to korekcja dziesiętna będzie nieprawidłowa.

Operacje logiczne

"ANL"

a) ang. „logical AND" – pomnóż logicznie
b) wykonywany jest iloczyn logiczny AND (mnożenie bitów „bit po bicie") wskazanych w instrukcji dwóch argumentów. Wynik operacji jest wpisywany do argumentu pierwszego instrukcji
c) znaczniki: nie zmieniają się
d) rodzaje instrukcji:

– **ANL A , Rn**
wymnożona logicznie zostaje zawartość akumulatora i rejestru Rn, wynik w A
A <- A ∩ Rn gdzie Rn = R0...R7 (jeden z rejestrów roboczych)
kod: 0 1 0 1 1 n2 n1 n0 gdzie n2...n0 – wskazują na R0...7 stąd: 58h-5Fh
cykle: 1 bajty: 1
przykład: ANL A , R3

– **ANL A , adres**
wymnożona logicznie zostaje zawartość akumulatora i komórki o podanym adresie „adres", wynik zostaje umieszczony w A
A <- A ∩ (adres)
kod: 0 1 0 1 0 1 0 1 55h
cykle: 1
bajty: 2 (kod instrukcji 55h + adres)
przykład: ANL A , 45h (mnożenie logicznie A i zawartości komórki pod adresem 45h)

– **ANL A , @Ri**
wymnożona logicznie zostaje zawartość akumulatora komórki w wewn. RAM o adresie wskazywanym przez rejestr Ri. (R0 lub R1)
A <- A ∩ (Ri)
kod: 0 1 0 1 0 1 1 i gdzie i wskazuje na R0 (i=0) lub R1 (i=1) stąd: 56h, 57h
cykle: 1 bajty: 1
przykład: ANL A , @R1 (wymnożenie. logiczne A i zawartości komórki o adresie w R1)

– **ANL A , #dana**
wymnożona logicznie zostaje zawartość akumulatora przez argument stały (8-bitowa liczba)
A <- A ∩ dana
kod: 0 1 0 1 0 1 0 0 54h
cykle: 1
bajty: 2 (kod instrukcji 54h + dana)
przykład: ANL A , #23 (mnożenie logiczne A i liczby 23)

– **ANL adres , A**
wymnożona logicznie zostaje zawartość akumulatora i komórki o podanym adresie „adres", wynik zostaje umieszczony w komórce pamięci o adresie "adres"
(adres) <- (adres) ∩ A
kod: 0 1 0 1 0 0 1 1 52h
cykle: 1 bajty: 2 (kod instrukcji 55h + adres)
przykład: ANL A , 45h (mnożenie logicznie A i zawartości komórki pod adresem 45h)

– **ANL adres , #dana**
wymnożona logicznie zostaje zawartość komórki o adresie „adres" przez argument stały (8-bitowa liczba)
(adres) <- (adres) ∩ dana
kod: 0 1 0 1 0 0 1 1 53h
cykle: 2 bajty: 3 (kod instrukcji 54h + adres + dana)
przykład: ANL 45h , #23 (mnożenie logiczne zawartości komórki 45h i liczby 23)

"ORL"

a) ang. „logical OR" – zsumuj logicznie
b) wykonywana jest suma logiczna OR (dodawanie bitów „bit po bicie") wskazanych w instrukcji dwóch argumentów. Wynik operacji jest wpisywany do argumentu pierwszego instrukcji
c) znaczniki: nie zmieniają się
d) rodzaje instrukcji:

– **ORL A , Rn**
dodana logicznie zostaje zawartość akumulatora i rejestru Rn, wynik w A
A <- A ∪ Rn gdzie Rn = R0...R7 (jeden z rejestrów roboczych)
kod: 0 1 0 0 1 n2 n1 n0 gdzie n2...n0 – wskazują na R0...7 stąd: 48h-4Fh
cykle: 1 bajty: 1
przykład: ORL A , R7

– **ORL A , adres**
dodana logicznie zostaje zawartość akumulatora i komórki o podanym adresie „adres", wynik zostaje umieszczony w A
A <- A ∪ (adres)
kod: 0 1 0 0 0 1 0 1 45h
cykle: 1 bajty: 2 (kod instrukcji 45h + adres)
przykład: ORL A , 19h (dodanie logiczne A i zawartości komórki pod adresem 19h)

– **ORL A , @Ri**
dodana logicznie zostaje zawartość akumulatora komórki w wewn. RAM o adresie wskazywanym przez rejestr Ri. (R0 lub R1)
A <- A ∪ (Ri)
kod: 0 1 0 0 0 1 1 i gdzie i wskazuje na R0 (i=0) lub R1 (i=1) stąd: 46h, 47h
cykle: 1 bajty: 1
przykład: ORL A , @R0 (dodanie logiczne A i zawartości komórki o adresie w R0)

– **ORL A , #dana**
dodana logicznie zostaje zawartość akumulatora przez argument stały (8-bitowa liczba)
A <- A ∪ dana
kod: 0 1 0 0 0 1 0 0 44h
cykle: 1 bajty: 2 (kod instrukcji 44h + dana)
przykład: ORL A , #23 (dodanie logiczne A i liczby 23)

– **ORL adres , A**
dodana logicznie zostaje zawartość akumulatora i komórki o podanym adresie „adres", wynik zostaje umieszczony w komórce pamięci o adresie "adres"
(adres) <- (adres) ∪ A
kod: 0 1 0 0 0 0 1 1 42h
cykle: 1 bajty: 2 (kod instrukcji 42h + adres)
przykład: ORL A , 20h (dodanie logicznie A i zawartości komórki pod adresem 20h)

– **ORL adres , #dana**
dodana logicznie zostaje zawartość komórki o adresie „adres" oraz argument stały (8-bitowa liczba)
(adres) <- (adres) ∪ dana
kod: 0 1 0 0 0 0 1 1 43h
cykle: 2 bajty: 3 (kod instrukcji 43h + adres + dana)
przykład: ORL 12h , #99 (dodanie logiczne zawartości komórki 12h i liczby 99)

"XRL"

a) ang. „logical XOR" – zsumuj mod 2 (różnica symetryczna)
b) wykonywana jest suma mod 2 XOR wskazanych w instrukcji dwóch argumentów. Wynik operacji jest wpisywany do argumentu pierwszego instrukcji
c) znaczniki: nie zmieniają się
d) rodzaje instrukcji:

– **XRL A , Rn**
zsumowana (mod 2) zostaje zawartość akumulatora i rejestru Rn, wynik w A
A <- A ⊕ Rn gdzie Rn = R0...R7 (jeden z rejestrów roboczych)
kod: 0 1 1 0 1 n2 n1 n0 gdzie n2...n0 – wskazują na R0...7 stąd: 68h-6Fh
cykle: 1 bajty: 1
przykład: XRL A , R7

– **XRL A , adres**
zsumowana (mod 2) logicznie zostaje zawartość akumulatora i komórki o podanym adresie „adres", wynik zostaje umieszczony w A
A <- A ⊕ (adres)
kod: 0 1 1 0 0 1 0 1 65h
cykle: 1 bajty: 2 (kod instrukcji 65h + adres)
przykład: XRL A , 19h (dodanie logiczne A i zawartości komórki pod adresem 19h)

– **XRL A , @Ri**
zsumowana (mod 2) logicznie zostaje zawartość akumulatora komórki w wewn. RAM o adresie wskazywanym przez rejestr Ri. (R0 lub R1)
A <- A ⊕ (Ri)
kod: 0 1 1 0 0 1 1 i gdzie i wskazuje na R0 (i=0) lub R1 (i=1) stąd: 66h, 67h
cykle: 1 bajty: 1
przykład: XRL A , @R0 (zsumowanie (mod 2) logiczne A i zawartości komórki o adresie w R0)

– **XRL A , #dana**
zsumowana (mod 2) logicznie zostaje zawartość akumulatora przez argument stały (8-bitowa liczba)
A <- A ⊕ dana
kod: 0 1 1 0 0 1 0 0 64h
cykle: 1 bajty: 2 (kod instrukcji 64h + dana)
przykład: XRL A , #23 (zsumowanie (mod 2) logiczne A i liczby 23)

– **XRL adres , A**
zsumowana (mod 2) logicznie zostaje zawartość akumulatora i komórki o podanym adresie „adres", wynik zostaje umieszczony w komórce pamięci o adresie „adres"
(adres) <- (adres) ⊕ A
kod: 0 1 1 0 0 0 1 1 62h
cykle: 1 bajty: 2 (kod instrukcji 62h + adres)
przykład: XRL A , 20h (zsumowanie (mod 2) logicznie A i zawartości komórki pod adresem 20h)

– **XRL adres , #dana**
zsumowana (mod 2) logicznie zostaje zawartość komórki o adresie „adres" oraz argument stały (8-bitowa liczba)
(adres) <- (adres) ⊕ dana
kod: 0 1 1 0 0 0 1 1 63h
cykle: 2 bajty: 3 (kod instrukcji 63h + adres + dana)
przykład: XRL 12h , #99 (dodanie logiczne zawartości komórki 12h i liczby 99)

"CLR A"

a) ang. „clear accumulator" – zeruj akumulator
b) do akumulatora zostaje wpisana wartość 0.
c) znaczniki: bez zmian
d) A <- 0
kod: 1 1 1 0 0 1 0 0 E4h
cykle: 1 bajty: 1
Przykład: efekt wyzerowania akumulatora można uzyskać stosując instrukcję:
MOV A , #0
lecz w tym przypadku instrukcja ma długość 2 bajtów (a CLR A tylko 1), co w efekcie skraca długość kodu programu i oszczędza pamięć.

"CPL A"

a) ang. „complement accumulator" – zaneguj akumulator
b) wartość akumulatora zostaje zanegowana, wynik wpisany zostaje do akumulatora
c) znaczniki: bez zmian
d) A <- / A
kod: 1 1 1 1 0 1 0 0 F4h
cykle: 1 bajty: 1
Przykład: aby np. zmienić znak liczby zapisanej w akumulatorze w kodzie U2 należy wykonać sekwencję instrukcji:

```
CPL   A     ;negacja akumulatora
INC   A     ;inkrementacja akumulatora
.....
```

"RL A"

a) ang. „rotate left" – przesuń w lewo
b) zawartość akumulatora zostaje przesunięta w lewo o 1 pozycję (o 1 bit), to znaczy że:
bit 1 przyjmuje wartość bitu 0
bit 2 przyjmuje wartość bitu 1
itd.....
bit 7 przyjmuje wartość bitu 6
a
bit 0 przyjmuje wartość bitu 7
c) znaczniki: bez zmian
d) A <- rotacja w lewo (A)
kod: 0 0 1 0 0 0 1 1 23h
cykle: 1 bajty: 1
Przykład: jeżeli w A jest liczba 43h (01000011 binarnie) to po wykonaniu instrukcji:
RL A
akumulator będzie zawierał liczbę: 86h (10000110 binarnie).

"RLC A"

a) ang. „rotate left through carry" – przesuń cyklicznie w lewo ze znacznikiem C
b) zawartość akumulatora zostaje przesunięta w lewo o 1 pozycję (o 1 bit) z uwzględnieniem znacznika C, to znaczy że: znacznik C przyjmuje wartość bitu 7 (akumulatora oczywiście)
bit 1 przyjmuje wartość bitu 0
bit 2 przyjmuje wartość bitu 1
itd.....
bit 7 przyjmuje wartość bitu 6
a
bit 0 przyjmuje wartość znacznika C
c) znaczniki: C jest ustawiany zgodnie z wynikiem operacji
d) A <- rotacja w lewo (A) z C
kod: 0 0 1 1 0 0 1 1 33h
cykle: 1 bajty: 1
Przykład: jeżeli w A jest liczba 54h (01010100 binarnie) to wykonanie instrukcji:

```
CLR   C     ;wyzeruje znacznik C
RLC   A     ;przesuń w lewo z C
            ;akumulator
....
```

spowoduje wymnożenie przez 2 liczby zapisanej w naturalnym kodzie binarnym w akumulatorze.

"RR A"

a) ang. „rotate right" – przesuń w prawo
b) zawartość akumulatora zostaje przesunięta w prawo o 1 pozycję (o 1 bit), to znaczy że:
bit 0 przyjmuje wartość bitu 1
bit 1 przyjmuje wartość bitu 2
itd.....
bit 6 przyjmuje wartość bitu 7
a
bit 7 przyjmuje wartość bitu 0
c) znaczniki: bez zmian
d) A <- rotacja w prawo (A)
kod: 0 0 0 0 0 0 1 1 03h
cykle: 1 bajty: 1
Przykład: jeżeli w A jest liczba 43h (01000011 binarnie) to po wykonaniu instrukcji
RR A
akumulator będzie zawierał liczbę: A1h (10100001 binarnie).

"RRC A"

a) ang. „rotate right through carry" – przesuń cyklicznie w prawo ze znacznikiem C
b) zawartość akumulatora zostaje przesunięta w prawo o 1 pozycję (o 1 bit) z uwzględnieniem znacznika C, to znaczy że: znacznik C przyjmuje wartość bitu 0 (akumulatora oczywiście)
bit 0 przyjmuje wartość bitu 1
bit 1 przyjmuje wartość bitu 2
itd.....
bit 6 przyjmuje wartość bitu 7
a
bit 7 przyjmuje wartość znacznika C
c) znaczniki: C jest ustawiany zgodnie z wynikiem operacji
d) A <- rotacja w prawo (A) z C
kod: 0 0 0 1 0 0 1 1 13h
cykle: 1 bajty: 1
Przykład: jeżeli w A jest liczba 54h (01010100 binarnie) to wykonanie instrukcji:

```
CLR   C     ;wyzeruje znacznik C
RLC   A     ;przesuń w lewo
            z C akumulator
....
```

spowoduje podzielenie przez 2 liczby zapisanej w naturalnym kodzie binarnym w akumulatorze.

"SWAP A"

a) ang. „swap nibbles within accumulator" – wymień półbajty w akumulatorze
b) w wyniku tej instrukcji wymieniona zostaje zawartość bitów 3...0 (mniej znaczący półbajt) i bitów 7...4 (bardziej znaczący półbajt) akumulatora. Operacja ta jest równoważna 4-krotnemu przesunięciu zawartości akumulatora.
c) znaczniki: bez zmian
d) A_{3-0} <-> A_{7-4}
kod: 1 1 0 0 0 1 0 0 C4h
cykle: 1 bajty: 1
Przykład: sekwencja podana poniżej powoduje zamianę półbajtów akumulatora

```
MOV   A , #52h  ;wpisanie do akumulatora
liczby 52h
SWAP A          ;wykonanie polecenia
                zamiany
.....           ;w akumulatorze
                znajduje się teraz
                 liczba 25h
```

Uff! Na razie to tyle w następnym odcinku dokończenie listy instrukcji, a więc pozostałe komendy dotyczące:
- operacji przemieszczania danych
- operacji na bitach (znacznikach)
- skoki i pozostałe

oraz krótki opis asemblera ASM51 przeznaczony szczególnie dla komputerowców.

Sławomir Surowiński

Lekcja 2

W dzisiejszej lekcji sprawdzimy działanie niektórych spośród omówionych wcześniej instrukcji arytmetycznych i logicznych procesora na podstawie przykładowego programu. Działanie programu jest bardzo proste, otóż:

a) najpierw program prosi o wprowadzenie dwóch liczb 8-bitowych w postaci heksadecymalnej, czyli z zakresu 0...FFh (0...255 dziesiętnie). Pierwsza liczba wyświetlana jest na wyświetlaczach DL1 i DL2, druga na DL4 i DL5
b) następnie wykonywana jest wybrana przez Ciebie operacja arytmetyczna lub logiczna (o tym jak ją wybrać – za chwilę
c) w efekcie na wyświetlaczach DL7 i DL8 wypisywany jest wynik operacji, który możesz sprawdzić ręcznie (na papierze) lub korzystając z kalkulatora wyposażonego w konwerter liczb zapisanych dziesiętnie i szesnastkowo (np. taki z MS-Windows).

Program w postaci listingu – czyli w zapisie źródłowym z dodatkowymi informacjami istotnymi szczególnie dla tych którzy nie mają komputera jest następujący:

```
                ;Program do lekcji nr 2
                ;testowanie komend: ADD, SUBB, ANL, ORL, XRL, SWAP
                ;z wykorzystaniem instrukcji BIOS'a

                ;*********************************
8000                    org     8000h           ;pocztek zewn. pamieci programu
                ;*********************************

8000 120274     znowu:  lcall   CLS             ;wyczyszczenie wyswietlacza
8003 75F001             mov     B,#1            ;pozycja 1 na displeju
8006 757840             mov     DL1,#_minus     ;znak "—" na pozycji wprowadzenia
8009 757940             mov     DL2,#_minus     ;pierwszego skladnika
800C 1203A7             lcall   GETACC          ;pobranie skladnika 1 dodawania
800F 128036             lcall   wait1           ;odczekaj sekunde
8012 C0E0               push    Acc             ;i przechowanie go na stosie
8014 75F004             mov     B,#4            ;pozycja 4 na displeju
8017 757B40             mov     DL4,#_minus     ;znak "—" na pozycji wprowadzeniaa
801A 757C40             mov     DL5,#_minus     ;drugiego skladnika
801D 1203A7             lcall   GETACC          ;pobranie skladnika 2 dodawania
8020 128036             lcall   wait1           ;odczekaj sekunde
8023 D0F0               pop     B               ;sciagniecie skladnika 1 ze stosu do rej.B
8025 C3                 clr     C               ;potrzebne do testowania instrukcji SUBB
8026 25F0               add     A,B             ;komenda dodania skladnikow - tu wstaw inne komendy
8028 75F007             mov     B,#7            ;pozycja 7 na displeju
802B 12024E             lcall   A2HEX           ;wypisanie wyniku dodawania
802E 128036             lcall   wait1           ;odczekaj sekunde
8031 128036             lcall   wait1           ;odczekaj sekunde
8034 80CA               sjmp    znowu           ;i nastepne skladniki

                ;*********************************
8036 C0E0       wait1:  push    Acc             ;przechowanie A na stosie
8038 74FF               mov     A,#255
803A 120295             lcall   DELAY           ;odczekanie 0,5 sek
803D 74FF               mov     A,#255
803F 120295             lcall   DELAY           ;odczekanie 0,5 sek (w sumie 1 sek.)
8042 D0E0               pop     Acc             ;odtworzenie A (ze stosu)
8044 22                 ret                     ;powrot do programu glownego
                ;*********************************
8045                    END
```

Szczegółowy opis listingu nie jest tematem niniejszej lekcji (a przyszłego odcinka szkoły mikroprocesorowej), toteż przedstawię tylko istotne informacje potrzebne do wykonania zadania z naszej dzisiejszej lekcji. Informacje podzielę na te istotne dla komputerowców oraz dla „ręczniaków" (o ile mogę posłużyć się takim skrótem), tak więc, patrzymy na listing powyżej i wyjaśniamy sobie:

a) w pierwszej kolumnie podany jest adres początkowy danej linii programu z zawartą w niej instrukcją. U nas adres początkowy to 8000h – początek pamięci SRAM w komputerku. Zauważmy że cały program zajmuje 45 bajtów, bo ostatnim adresem jest 8045h – ostatnia linia listingu.
b) w każdej zawierającej instrukcję linii tuż za adresem znajduje się ciąg bajtów będący odpowiednikiem maszynowym instrukcji zapisanej w dalszej części linii w sposób jawny. Dzięki temu „niekomputerowcy" będą mogli po prostu wklepać te dane „ciurkiem" od adresu 8000h bez mozolnego tłumaczenia z postaci źródłowej znajdującej się w trzeciej kolumnie listingu). Warto jednak przy tym chociaż chwilę zastanowić się i przetłumaczyć już teraz (korzystając z tabeli we wkładce) znane i nieznane instrukcje w kolej-

nych liniach a następnie porównać je z danymi z kolumny drugiej.

c) w listingu występują odwołania do procedur umieszczonych w programie monitora są to:
GETACC, A2HEX i DELAY. Nie wdając się w szczegóły (na razie) wyjaśniam że nazwom tym przypisane są adresy w przestrzeni programu monitora (EPROM) od których zaczynają się kody tych procedur. Ich działanie jest następujące:
"GETACC" : procedura pobrania, z klawiatury, 8-bitowej liczby zapisanej w postaci szesnastkowej i umieszczenie jej w akumulatorze z jednoczesnym wyświetleniem wpisywanej przez użytkownika wartości na wyświetlaczu. Pozycja na której wypisywana jest wartość na displeju powinna być określona przed jej wywołaniem w rejestrze B. W naszym przykładzie dzięki tej procedurze możesz wprowadzić składniki testowanego działania.
"A2HEX" : procedura wyświetlenia na displeju w postaci szesnastkowej (na 2 wyświetlaczach) aktualnej zawartości akumulatora. Podobnie jak poprzednio pozycja od której ma być wypisana liczba musi być określona w rejestrze B. W naszym przykładzie dzięki tej procedurze wyświetlany jest wynik operacji na wyświetlaczach DL7 i DL8.
"DELAY" : procedura opóźnienia, czas opóźnienia jest podawany w akumulatorze przed wywołaniem procedury a mnożnik wynosi około 1,95 ms. Jeżeli zatem wpiszemy do akumulatora wartość 255 to po wywołanie procedury DELAY będzie trwało ok. 255 x 1,95 ms = 497 ms czyli około 0,5 sekundy. Zastosowanie tej procedury w naszym przykładzie ma umożliwić Ci obserwację wykonywania programu „krok po kroku".

d) dowolny tekst znajdujący się za znakiem średnika „;" jest traktowany jako komentarz i nie jest brany pod uwagę podczas kompilacji programu w przypadku korzystania z kompilatora na komputer PC. Uwagi zawarte w komentarzu są bardzo przydatne podczas analizy programu.

e) w przykładowym listingu większość instrukcji jest Ci jeszcze nie znana, są one jednak niezbędne do wykonania tej lekcji, toteż proszę traktuj je jako domyślne, więcej informacji na ich temat w kolejnym numerze EdW.

Dla „niekomputerowców":

Korzystając z funkcji monitora „Edit" – należy wklepać kod programu, korzystając z listingu powyżej, od adresu 8000h. Dla ułatwienia podaję że pierwsze bajty kodu to:

12, 02, 74, 75, F0, 01, 75, 78, 40, 75, 79, 40, 12, 03, A7, 12, 80, 36, C0, E0 itd.... .

Wytrwałym proponuję analizę i przetłumaczenie (tabela instrukcji we wkładce) kilku pierwszych lub całego listingu programu a następnie porównanie efektów swojej pracy z kodami podanymi w naszym przykładzie.

f) `w naszym listingu w linii o adresie 8026h znajduje się właściwa instrukcja testująca działanie danej funkcji arytmetyczno-logicznej (zaciemniona linia). W przykładzie naszym znajduje się instrukcja ADD – dodawania.

W przypadku chęci zastosowanie innej funkcji należy w miejsce kodu „25 F0" (pod adresem 801A) wpisać odpowiednie dla poszczególnych instrukcji, ciągi bajtów:

dla SUBB wpisać: 95 F0
dla ANL wpisać: 55 F0
dla ORL wpisać: 45 F0
dla XRL wpisać: 65 F0

korzystając z funkcji „Edit" monitora.

Uwaga, w przypadku testowania instrukcji SWAP A należy wpisać liczby: C4 00.

Zauważmy wszakże że poprzednie instrukcje były dwubajtowe, ta ostatnia zaś jest 1-bajtowa. dlatego na pozycji drugiego bajtu wpisałem 00 co jest kodem maszynowym instrukcji „NOP" – „nie rób nic". Instrukcję NOP poznasz dokładnie w kolejnym numerze EdW. Na razie powiem Ci tylko że podczas wykonywania instrukcji NOP procesor nie robi nic – czyli de facto leniuchuje przez jeden cykl maszynowy (12 cykli zegara). Zastosowanie tej instrukcji przy modyfikacji kodu z poziomu monitora (funkcja Edit) jest uzasadnione, bowiem jest ona niejako „wypełniaczem" brakującego bajtu kodu o adresie: 8027h. Przy wprowadzaniu danych podczas testu instrukcji SWAP pierwszy składnik nie jest brany pod uwagę (bo instrukcja SWAP jest 1-argumentowa) , toteż można wpisać dowolną wartość najlepiej 00.

Po modyfikacjach należy opuścić funkcję „Edit" i uruchomić ponownie program – komenda monitora „Jump", a następnie sprawdzić działanie nowo wprowadzonej instrukcji.

Dla komputerowców:

Uwaga: przy przeglądaniu i modyfikacjach naszego przykładu korzystaj z DOSowego Nortona Commandera! Zanim zaczniesz zabawę przeczytaj uważnie plik informacyjny ASM51.DOC.

Na dyskietce z kompilatorem ASM51 znajduje się zbiór źródłowy z naszym przykładem pod nazwą „LEKCJA2.S03". Powinieneś wykonać następujące czynności:

– skompilować przykład do postaci maszynowej , skorzystaj z programu wsadowego „DO.BAT" , wydaj komendę:
> DO LEKCJA2 {Enter}
– załaduj program do komputerka (komenda „Load" monitora)
– uruchom program (komenda „Jump") najpierw z instrukcją arytmetyczną ADD (domyślnie znajduje się w pliku LEKCJA2.S03)
– wykonaj kilka działań na przykładowych liczbach
– zmodyfikuj plik źródłowy (klawisz F4 w Nortonie) czyli linię z instrukcją ADD zamień na inne instrukcje podane wcześniej w ćwiczeniu: SUBB, ANL, ORL, XRL, wpisując je w miejsce ADD (uwaga: wielkość liter nie ma znaczenia)
– skompiluj ponownie program i załaduj do komputerka
– uruchom ponownie program i wykonaj kilka działań sprawdzając na kartce papieru lub kalkulatorze poprawność działania poszczególnych instrukcji.
Przy okazji zajrzyj do powstałych w wyniku kompilacji zbiorów:
– listingu : LEKCJA2.LST
– zbioru : LEKCJA2.HEX zapisanego w formacie Intel HEX. Więcej na temat tego formatu danych możesz dowiedzieć się z artykułu w naszym bratnim piśmie „Elektronika Praktyczna" nr 10/97 na stronie 75. Na łamach naszego kursu wrócimy przy innej okazji do tego tematu.

Postaraj się zapoznać ze zbiorem typu listing. Przekształć „ręcznie" dowolne linie programu (korzystając z tabeli we wkładce w tym numerze EdW) i porównaj otrzymane kody poszczególnych linii z tymi w zbiorze LEKCJA2.LST .

Jako przykłady proponuję wykonać następujące operacje:

a) test funkcji ADD:
argumenty: 12h, 67h wynik: 79h
argumenty: FEh, 02h wynik: 00h

b) test funkcji SUBB: (uwaga: tu 1–szy argument jest odejmowany od 2–go !)
argumenty: 12h, 67h wynik: 55h
argumenty: F0h, 05h wynik: 15h

c) test funkcji ANL:
argumenty: 1Fh, EEh wynik: 0Eh
argumenty: F0h, 0Fh wynik: 00h

d) test funkcji ORL:
argumenty: 7Eh, 80h wynik: FEh
argumenty: 70h, 09h wynik: 79h

e) test funkcji XRL:
argumenty: 25h, 6Bh wynik: 4Eh
argumenty: 55h, AAh wynik: FFh

f) test funkcji SWAP:
argumenty: pierwszy nie istotny , 78h
wynik: 87h
argumenty: pierwszy nie istotny , 39h
wynik: 93h

Zauważ że wszystkie instrukcje działają na liczbach 8–bitowych, toteż w przypadku przekroczenia zakresu tych liczb informacja o wyniku jest częściowo tracona.

Życzę wesołej zabawy i dużo wytrwałości !

Sławomir Surowiński