STEP7 - dodatno gradivo

David Nedeljković

Univerza v Ljubljani
Fakulteta za elektrotehniko

Ljubljana, 2011

KAZALO

1. STEP 7 – DODATEK

1.1 DIGITALNE LOGIČNE OPERACIJE

Digitalne logične operacije obdelujejo soležne bite akumulatorjev AKU1 in AKU2 ali pa soležne bite akumulatorja AKU1 in konstante, pri čemer dobimo rezultat na soležnih bitih v AKU1. Obdelave soležnih bitov so možne po logičnih funkcijah IN (AND), ALI (OR) in EKSKLUZIVNI ALI (XOR), prav tako pa imamo možnost izbrati podatkovno širino, nad katero se operacija izvede (spodnjih 16 bitov ali vseh 32 bitov).

Izvajanje digitalnih logičnih operacij je neodvisno od stanja bitov v statusni besedi, glede na rezultat operacije pa se v statusni besedi postavi pogojna koda CC1: če je rezultat v AKU1 (oziroma njegovi spodnji besedi) enak 0, zavzame CC1 vrednost 0, če pa je rezultat različen od 0, se CC1 postavi na 1. Pogojna koda CC0 se vedno postavi na 0, prav tako tudi bit presežka OV. Podrobnejša razlaga o statusni besedi je podana v [3, 23, 31].

Digitalne logične operacije bomo pogosto elegantno uporabili, kadar bomo želeli vsebino posameznih bitov neke besede (tudi zloga ali dvojne besede) "maskirati", če bo torej naš cilj, da do izraza pridejo le vrednosti izbranih bitov te besede.

1.1.1 AW

(and word) poveže soležne bite spodnje besede akumulatorja AKU1 in spodnje besede akumulatorja AKU2 po logični funkciji IN:

```
...     // prej       AKU1: 16#89AB_CDEF  =  2#1000_1001_1010_1011_1100_1101_1110_1111
                       AKU2: 16#0123_6543  =  2#0000_0001_0010_0011_0110_0101_0100_0011
AW
...     // potem      AKU1: 16#89AB_4543  =  2#1000_1001_1010_1011_0100_0101_0100_0011
                       AKU2: 16#0123_6543  =  2#0000_0001_0010_0011_0110_0101_0100_0011
```

Ta zgled velja za neposredno naslavljanje, kjer prej z ustreznimi operacijami nalaganja poskrbimo, da sta oba operanda v akumulatorjih.

Pri takojšnjem naslavljanju mora biti prvi operand v AKU1, drugega pa podamo kot konstanto v ukazu, npr.:

```
...     // prej       AKU1: 16#89AB_CDEF  =  2#1000_1001_1010_1011_1100_1101_1110_1111

AW  W#16#678F    //          16#678F  =                    2#0110_0111_1000_1111

...     // potem      AKU1: 16#89AB_458F  =  2#1000_1001_1010_1011_0100_0101_1000_1111
```

V obeh primerih dobimo rezultat v AKU1. Soležni bit rezultata je 1 samo v primeru, ko sta soležna bita obeh operandov 1. Vsebina AKU2 ostane nespremenjena. Zapomniti si je treba, da se operacija izvede samo na spodnjih 16 bitih; zgornja beseda akumulatorja AKU1 ravno tako ohrani svojo prejšnjo vsebino!

Zgled:

```
L    MD 40   // MD 40 v AKU1
L    IW 0    // IW 0 v AKU1, MD 40 gre v AKU2!
AW           // pozor: kaj je spodnjih 16 bitov!?
T    QW 0    // rezultat na QW 0
```

V AKU1 najprej naložimo vsebino pomožne pomnilniške dvojne besede `MD 40`; v AKU1 so torej zlogi `MB 40`, `MB 41`, `MB 42` in `MB 43`. Z naslednjim nalaganjem se ta vsebina AKU1 prenese v AKU2, v AKU1 - natančneje, v njegovi spodnji besedi - pa dobimo vsebino vhodne besede `IW 0`. Vsi biti zgornje besede AKU1 se pri tem drugem nalaganju (podatkovna širina W) postavijo na 0.
Sledi izvajanje operacije `AW`, potem pa prenos rezultata (spodnje besede AKU1) na izhodno besedo `QW 0`.

Če zgled natančneje preučimo, opazimo, da se je pravzaprav izvedla povezava po logični funkciji IN med soležnimi biti `MW 42` in `IW 0`! Res pa imamo po operaciji `AW` v zgornji besedi AKU1 vsebino iz `MW 40`.

Oglejmo si še zgled za maskiranje vhodne besede `IW 0` s konstanto, kjer želimo, da se na najvišji nibble `QW 0` prenese vsebina najvišjega nibbla `IW 0` in na najnižji nibble `QW 0` vsebina najnižjega nibbla `IW 0`; biti obeh preostalih (vmesnih) nibblov `QW 0` pa se naj postavijo na 0:

```
L    IW 0        // IW 0 v AKU1
AW   W#16#F00F   // maska: prepušča najvišji in najnižji nibble
T    QW 0        // rezultat na QW 0
```

Kaj se zgodi v programu:

```
L    IW 0        //    AKU1: 16#0000_XXXX  =  2#0000_0000_0000_0000_xxxx_xxxx_xxxx_xxxx
AW   W#16#F00F   //             16#F00F  =                    2#1111_0000_0000_1111
T    QW 0        //    AKU1: 16#0000_X00X  =  2#0000_0000_0000_0000_xxxx_0000_0000_xxxx
```

Soležne bite, ki nas zanimajo, smo maskirali z "1", pri čemer je povezava po logični funkciji IN dala za rezultat kar vrednost dotičnega bita z vhodne enote. Irelevantne bite smo maskirali z "0", saj tako povezava po IN rezultira z "0", ne glede na stanje soležnega vhodnega bita. V zgledu je z "x" ponazorjena poljubna vrednost posameznega bita (0 ali 1), z "X" pa poljubna šestnajstiška števka (0-F).

1.1.2 OW

(or word) pri neposrednem naslavljanju poveže soležne bite spodnje besede akumulatorja AKU1 in spodnje besede akumulatorja AKU2 po logični funkciji ALI:

```
...     // prej       AKU1: 16#89AB_CDEF  =  2#1000_1001_1010_1011_1100_1101_1110_1111
                      AKU2: 16#0123_6543  =  2#0000_0001_0010_0011_0110_0101_0100_0011
OW
...     // potem      AKU1: 16#89AB_EDEF  =  2#1000_1001_1010_1011_1110_1101_1110_1111
                      AKU2: 16#0123_6543  =  2#0000_0001_0010_0011_0110_0101_0100_0011
```

Drugi operand lahko podamo tudi kot konstanto v ukazu:

```
...     // prej       AKU1: 16#89AB_CDEF  =  2#1000_1001_1010_1011_1100_1101_1110_1111

OW  W#16#678F    //          16#678F  =                    2#0110_0111_1000_1111

...     // potem      AKU1: 16#89AB_EFEF  =  2#1000_1001_1010_1011_1110_1111_1110_1111
```

Tudi tukaj se rezultat v obeh primerih izoblikuje v AKU1. Soležni bit rezultata je 1 v primeru, ko je vsaj eden izmed soležnih bitov obeh operandov enak 1. Vsebina AKU2 ostane nespremenjena. Ne smemo prezreti, da se tudi ta operacija izvede samo na spodnjih 16 bitih, zato zgornja beseda akumulatorja AKU1 ohrani svojo prejšnjo vsebino!

Zgled:

```
L    IB 0    // IB 0 v AKU1
L    QW 0    // QW 0 v AKU1, IB 0 gre v AKU2!
OW           //
T    MW 16   // rezultat na MW 16
```

V AKU1 najprej naložimo vsebino vhodnega zloga `IB 0`; ta se naloži na nižji zlog spodnje besede AKU1, ostali biti (preostali trije višji zlogi) v AKU1 pa se postavijo na 0. Z naslednjim nalaganjem se ta vsebina AKU1 prenese v AKU2, v AKU1 - natančneje, v njegovi spodnji besedi - pa dobimo vsebino izhodne besede `QW 0`. Vsi biti zgornje besede AKU1 se pri tem drugem nalaganju (podatkovna širina W) postavijo na 0.
Sledi izvajanje operacije `OW`, potem pa prenos rezultata (spodnje besede AKU1) na `MW 16`.

Podobno kot prej si oglejmo še zgled za maskiranje vhodne besede `IW 0` s konstanto, kjer želimo, da se na najvišji nibble `QW 0` prenese vsebina najvišjega nibbla `IW 0` in na

najnižji nibble `QW 0` vsebina najnižjega nibbla `IW 0`; biti obeh preostalih (vmesnih) nibblov `QW 0` pa se naj tokrat postavijo na 1:

```
L    IW 0        // IW 0 v AKU1
OW   W#16#0FF0   // maska: prepušča najvišji in najnižji nibble
T    QW 0        // rezultat na QW 0
```

Kaj se zgodi v programu:

```
L    IW 0        //    AKU1: 16#0000_XXXX  =  2#0000_0000_0000_0000_xxxx_xxxx_xxxx_xxxx
OW   W#16#0FF0   //              16#0FF0   =                      2#0000_1111_1111_0000
T    QW 0        //    AKU1: 16#0000_XFFX  =  2#0000_0000_0000_0000_xxxx_1111_1111_xxxx
```

Soležne bite, ki nas zanimajo, smo maskirali z "0", pri čemer je povezava po logični funkciji ALI dala za rezultat kar vrednost pripadajočega bita z vhodne enote. Nepomembne bite smo maskirali z "1", saj tako povezava po ALI da rezultat "1", ne glede na stanje soležnega vhodnega bita. Tudi tukaj smo z "x" ponazorili poljubno vrednost posameznega bita (0 ali 1), z "X" pa poljubno šestnajstiško števko (0-F).

1.1.3 XOW

(exclusive or word) pri neposrednem naslavljanju poveže soležne bite spodnje besede akumulatorja AKU1 in spodnje besede akumulatorja AKU2 po logični funkciji EKSKLUZIVNI ALI:

```
...     // prej       AKU1: 16#89AB_CDEF  =  2#1000_1001_1010_1011_1100_1101_1110_1111
                       AKU2: 16#0123_6543  =  2#0000_0001_0010_0011_0110_0101_0100_0011
XOW
...     // potem      AKU1: 16#89AB_A8AC  =  2#1000_1001_1010_1011_1010_1000_1010_1100
                       AKU2: 16#0123_6543  =  2#0000_0001_0010_0011_0110_0101_0100_0011
```

Drugi operand lahko podamo tudi kot konstanto v ukazu:

```
...     // prej       AKU1: 16#89AB_CDEF  =  2#1000_1001_1010_1011_1100_1101_1110_1111

XOW  W#16#678F   //          16#678F  =                      2#0110_0111_1000_1111

...     // potem      AKU1: 16#89AB_AA60  =  2#1000_1001_1010_1011_1010_1010_0110_0000
```

Tudi tukaj rezultat v obeh primerih dobimo v AKU1. Soležni bit rezultata je 1 v primeru, ko je samo eden izmed soležnih bitov obeh operandov enak 1. Vsebina AKU2 ostane nespremenjena. Ne spreglejmo, da se tudi ta operacija izvede samo na spodnjih 16 bitih, zato zgornja beseda akumulatorja AKU1 ohrani svojo prejšnjo vsebino!

Zgled:

```
L    MW 0    // MW 0 v AKU1
L    IW 0    // IW 0 v AKU1, MW 0 gre v AKU2!
XOW          //
T    QW 124  // rezultat na QW 124
```

V AKU1 najprej naložimo vsebino `MW 0`; ta se naloži na spodnjo besedo AKU1, ostali biti (zgornja beseda) v AKU1 pa se postavijo na 0. Z naslednjim nalaganjem se ta vsebina AKU1 prenese v AKU2, v AKU1 - natančneje, v njegovi spodnji besedi - pa dobimo vsebino vhodne besede `IW 0`. Vsi biti zgornje besede AKU1 se tudi pri tem drugem nalaganju (podatkovna širina W) postavijo na 0.
Sledi izvajanje operacije `XOW`, potem pa prenos rezultata (spodnje besede AKU1) na izhodno besedo `QW 124`.

Tako kot pri prejšnjih ukazih si oglejmo še zgled za maskiranje vhodne besede `IW 0` s konstanto, kjer želimo, da se na najvišji nibble `QW 0` prenese vsebina najvišjega nibbla `IW 0` in na najnižji nibble `QW 0` vsebina najnižjega nibbla `IW 0`; biti obeh preostalih (vmesnih) nibblov `QW 0` pa naj zavzamejo negirano vrednost soležnih bitov z `IW 0`:

```
L    IW 0        // IW 0 v AKU1
XOW W#16#0FF0    // maska: prepušča najvišji in najnižji nibble
T    QW 0        // rezultat na QW 0
```

Kaj se zgodi v programu:

```
L    IW 0        //    AKU1: 16#0000_XXXX  =  2#0000_0000_0000_0000_xxxx_xxxx_xxxx_xxxx
XOW W#16#0FF0    //              16#0FF0   =                    2#0000_1111_1111_0000
T    QW 0        //    AKU1: 16#0000_XYYX  =  2#0000_0000_0000_0000_xxxx_yyyy_yyyy_xxxx
```

Soležne bite, ki nas zanimajo, smo maskirali z "0", pri čemer je povezava po logični funkciji EKSKLUZIVNI ALI dala za rezultat kar vrednost pripadajočega bita z vhodne enote. Preostale bite, ki jih želimo negirati, smo maskirali z "1", saj tako dobimo kot rezultat negirano vrednost soležnega vhodnega bita. V zgledu je z "x" ponazorjena poljubna vrednost posameznega bita (0 ali 1) in z "y" njegova negirana vrednost. Oznaka "X" ponazarja poljubno šestnajstiško številko (0-F), "Y" pa šestnajstiško številko, ki je eniški komplement "X".

V nadaljevanju so prikazane še digitalne logične operacije nad 32-bitnimi operandi. V splošnem veljajo tudi za njih enake ugotovitve kot pri 16-bitnih različicah operacij, seveda s to bistveno razliko, da se operacija zdaj izvaja na vseh 32 bitih. Zato moramo tudi operande in rezultat v AKU1 obravnavati temu ustrezno.

1.1.4 AD

(and double) pri neposrednem naslavljanju poveže vse soležne bite akumulatorja AKU1 in akumulatorja AKU2 po logični funkciji IN:

```
...     // prej        AKU1: 16#89AB_CDEF  =  2#1000_1001_1010_1011_1100_1101_1110_1111
                       AKU2: 16#0123_6543  =  2#0000_0001_0010_0011_0110_0101_0100_0011
AD
...     // potem       AKU1: 16#0123_4543  =  2#0000_0001_0010_0011_0100_0101_0100_0011
                       AKU2: 16#0123_6543  =  2#0000_0001_0010_0011_0110_0101_0100_0011
```

Pri takojšnjem naslavljanju je prvi operand v AKU1, drugega pa podamo kot konstanto v ukazu, npr.:

```
...     // prej        AKU1: 16#89AB_CDEF  =  2#1000_1001_1010_1011_1100_1101_1110_1111

AD  DW#16#A0C3_678F  // 16#A0C3_678F  =  2#1010_0000_1100_0011_0110_0111_1000_1111

...     // potem       AKU1: 16#8083_458F  =  2#1000_0000_1000_0011_0100_0101_1000_1111
```

Zgled:

```
L   MD 40   // MD 40 v AKU1
L   ID 0    // ID 0 v AKU1, MD 40 gre v AKU2!
AD          //
T   QD 0    // rezultat na QD 0
```

1.1.5 OD

(or double) pri neposrednem naslavljanju poveže soležne bite akumulatorja AKU1 in akumulatorja AKU2 po logični funkciji ALI:

```
...     // prej        AKU1: 16#89AB_CDEF  =  2#1000_1001_1010_1011_1100_1101_1110_1111
                       AKU2: 16#0163_6543  =  2#0000_0001_0110_0011_0110_0101_0100_0011
OD
...     // potem       AKU1: 16#89EB_EDEF  =  2#1000_1001_1110_1011_1110_1101_1110_1111
                       AKU2: 16#0163_6543  =  2#0000_0001_0110_0011_0110_0101_0100_0011
```

Drugi operand lahko podamo tudi kot konstanto v ukazu:

```
...     // prej        AKU1: 16#89AB_CDEF  =  2#1000_1001_1010_1011_1100_1101_1110_1111

OD  DW#16#A0C3_678F  // 16#A0C3_678F  =  2#1010_0000_1100_0011_0110_0111_1000_1111

...     // potem       AKU1: 16#A9EB_EFEF  =  2#1010_1001_1110_1011_1110_1111_1110_1111
```

Zgled:

```
L   MD 20              // MD 0 v AKU1
OD  DW#16#A0A0_0FF0    // maska
T   QD 0               // rezultat na QD 0
```

1.1.6 XOD

(exclusive or double) pri neposrednem naslavljanju poveže soležne bite akumulatorja AKU1 in akumulatorja AKU2 po logični funkciji EKSKLUZIVNI ALI:

```
...     // prej        AKU1: 16#89AB_CDEF  =  2#1000_1001_1010_1011_1100_1101_1110_1111
                       AKU2: 16#0123_6543  =  2#0000_0001_0010_0011_0110_0101_0100_0011
XOD
...     // potem       AKU1: 16#8888_A8AC  =  2#1000_1000_1000_1000_1010_1000_1010_1100
                       AKU2: 16#0123_6543  =  2#0000_0001_0010_0011_0110_0101_0100_0011
```

Drugi operand je lahko zapisan tudi kot konstanta v ukazu:

```
...     // prej        AKU1: 16#89AB_CDEF  =  2#1000_1001_1010_1011_1100_1101_1110_1111

XOD DW#16#A0C3_678F  // 16#A0C3_678F  =  2#1010_0000_1100_0011_0110_0111_1000_1111

...     // potem       AKU1: 16#2968_AA60  =  2#0010_1001_0110_1000_1010_1010_0110_0000
```

Zgled:

```
L   MD 0    // MD 0 v AKU1
L   ID 0    // ID 0 v AKU1, MD 0 gre v AKU2!
XOD         //
T   MD 96   // rezultat na MD 96
```

1.2 POMIČNE OPERACIJE (SHIFT)

```
SLW  konstanta
```

(Shift Left Word) pomakne bite iz spodnje besede AKU1 za toliko bitov v levo, kolikor znaša konstanta za ukazom. Rezultat dobimo v spodnji besedi AKU1. Izpraznjeni biti v AKU1 zavzamejo vrednost 0. Zgornja beseda AKU1 ostane nespremenjena, prav tako se ne spremeni vsebina AKU2. Pogojni kodi se postavita tako, da CC0 zavzame vrednost 0, CC1 pa vrednost zadnjega pomaknjenega bita.

```
SLW
```

Če število bitov za ukazom ni navedeno, se upošteva številska vrednost s spodnje besede iz AKU2.

```
...     // prej       AKU1: 16#89AB_0F55  =  2#1000_1001_1010_1011_0000_1111_0101_0101

SLW 6   //

...     // potem      AKU1: 16#89AB_D540  =  2#1000_1001_1010_1011_1101_0101_0100_0000
```

```
SLD  konstanta
```

(Shift Left Doubleword) pomakne bite iz AKU1 za toliko bitov v levo, kolikor znaša konstanta za ukazom. Rezultat dobimo v AKU1. Izpraznjeni biti v AKU1 zavzamejo vrednost 0. Vsebina AKU2 se ne spremeni. Pogojni kodi se postavita tako, da CC0 zavzame vrednost 0, CC1 pa vrednost zadnjega pomaknjenega bita.

```
SLD
```

Če število bitov za ukazom ni navedeno, se upošteva številska vrednost s spodnje besede iz AKU2.

```
...     // prej       AKU1: 16#89AB_0F55  =  2#1000_1001_1010_1011_0000_1111_0101_0101

SLD 6   //

...     // potem      AKU1: 16#6AC3_D540  =  2#0110_1010_1100_0011_1101_0101_0100_0000
```

```
SRW  konstanta
```

(Shift Right Word) pomakne bite iz spodnje besede AKU1 za toliko bitov v desno, kolikor znaša konstanta za ukazom. Rezultat dobimo v spodnji besedi AKU1. Izpraznjeni biti v AKU1 zavzamejo vrednost 0. Zgornja beseda AKU1 ostane nespremenjena, prav tako se ne spremeni vsebina AKU2. Pogojni kodi se postavita tako, da CC0 zavzame vrednost 0, CC1 pa vrednost zadnjega pomaknjenega bita.

```
SRW
```

Če število bitov za ukazom ni navedeno, se upošteva številska vrednost s spodnje besede iz AKU2.

```
...     // prej       AKU1: 16#FF55_AAFF  =  2#1111_1111_0101_0101_1010_1010_1111_1111

SRW 3   //

...     // potem      AKU1: 16#FF55_155F  =  2#1111_1111_0101_0101_0001_0101_0101_1111
```

```
SRD  konstanta
```

(Shift Right Doubleword) pomakne bite iz AKU1 za toliko bitov v levo, kolikor znaša konstanta za ukazom. Rezultat dobimo v AKU1. Izpraznjeni biti v AKU1 zavzamejo vrednost 0. Vsebina AKU2 se ne spremeni. Pogojni kodi se postavita tako, da CC0 zavzame vrednost 0, CC1 pa vrednost zadnjega pomaknjenega bita.

```
SRD
```

Če število bitov za ukazom ni navedeno, se upošteva številska vrednost s spodnje besede iz AKU2.

```
...     // prej       AKU1: 16#FF55_AAFF  =  2#1111_1111_0101_0101_1010_1010_1111_1111

SRD 3   //

...     // potem      AKU1: 16#1FEA_B55F  =  2#0001_1111_1110_1010_1011_0101_0101_1111
```

```
SSI  konstanta
```

(Shift Signed Integer) pomakne bite iz spodnje besede AKU1 za toliko bitov v desno, kolikor znaša konstanta za ukazom. Rezultat dobimo v spodnji besedi AKU1. Izpraznjeni biti v AKU1 zavzamejo vrednost 15. bita (bita z najvišjo težo iz spodnje

besede AKU1 – predznaka 16-bitnega predznačenega celega števila). Zgornja beseda AKU1 ostane nespremenjena, prav tako se ne spremeni vsebina AKU2. Pogojni kodi se postavita tako, da CC0 zavzame vrednost 0, CC1 pa vrednost zadnjega pomaknjenega bita.

```
SSI
```

Če število bitov za ukazom ni navedeno, se upošteva številska vrednost s spodnje besede iz AKU2.

```
...     // prej       AKU1: 16#FF55_AF0A  =  2#1111_1111_0101_0101_1010_1111_0000_1010

SSI 4   //

...     // potem      AKU1: 16#FF55_FAF0  =  2#1111_1111_0101_0101_1111_1010_1111_0000
```

```
SSD  konstanta
```

(Shift Signed Doubleword) pomakne bite iz AKU1 za toliko bitov v levo, kolikor znaša konstanta za ukazom. Rezultat dobimo v AKU1. Izpraznjeni biti v AKU1 zavzamejo vrednost 31. bita (bita z najvišjo težo iz AKU1 – predznaka 32-bitnega predznačenega celega števila). Vsebina AKU2 se ne spremeni. Pogojni kodi se postavita tako, da CC0 zavzame vrednost 0, CC1 pa vrednost zadnjega pomaknjenega bita.

```
SSD
```

Če število bitov za ukazom ni navedeno, se upošteva številska vrednost s spodnje besede iz AKU2.

```
...     // prej       AKU1: 16#FF55_AF0A  =  2#1111_1111_0101_0101_1010_1111_0000_1010

SSD 4   //

...     // potem      AKU1: 16#FFF5_5AF0  =  2#1111_1111_1111_0101_0101_1010_1111_0000
```

```
...     // prej       AKU1: 16#7F55_AF0A  =  2#0111_1111_0101_0101_1010_1111_0000_1010

SSD 4   //

...     // potem      AKU1: 16#0FF5_5AF0  =  2#0000_1111_1111_0101_0101_1010_1111_0000
```

1.3 ROTACIJSKE OPERACIJE (ROTATE)

`RLD  konstanta`

(Rotate Left Doubleword) zasuče bite iz AKU1 za toliko bitov v levo, kolikor znaša konstanta za ukazom. Rezultat dobimo v AKU1. Vsebina AKU2 se ne spremeni. Pogojni kodi se postavita tako, da CC0 zavzame vrednost 0, CC1 pa vrednost zadnjega pomaknjenega bita.

`RLD`

Če število bitov za ukazom ni navedeno, se upošteva številska vrednost s spodnje besede iz AKU2.

```
...     // prej       AKU1: 16#89AB_0F55  =  2#1000_1001_1010_1011_0000_1111_0101_0101
RLD 6   //
...     // potem      AKU1: 16#6AC3_D562  =  2#0110_1010_1100_0011_1101_0101_0110_0010
```

`RRD  konstanta`

(Rotate Right Doubleword) zasuče bite iz AKU1 za toliko bitov v desno, kolikor znaša konstanta za ukazom. Rezultat dobimo v AKU1. Vsebina AKU2 se ne spremeni. Pogojni kodi se postavita tako, da CC0 zavzame vrednost 0, CC1 pa vrednost zadnjega pomaknjenega bita.

`RRD`

Če število bitov za ukazom ni navedeno, se upošteva številska vrednost s spodnje besede iz AKU2.

```
...     // prej       AKU1: 16#FF55_AAFF  =  2#1111_1111_0101_0101_1010_1010_1111_1111
RRD 3   //
...     // potem      AKU1: 16#FFEA_B55F  =  2#1111_1111_1110_1010_1011_0101_0101_1111
```

RLDA

Ukaz zasuče bite iz AKU1 za en bit v levo. 0. bit iz AKU1 se postavi na vrednost CC1,. Rezultat dobimo v AKU1. Vsebina AKU2 se ne spremeni. Pogojni kodi se postavita tako, da CC0 zavzame vrednost 0, CC1 pa vrednost zadnjega pomaknjenega bita (31. bita iz AKU1).

```
...     // prej       AKU1: 16#89AB_0F55  =  2#1000_1001_1010_1011_0000_1111_0101_0101

RLDA    //

...     // potem      AKU1: 16#1356_1EAX  =  2#0001_0011_0101_0110_0001_1110_1010_101x
```

RRDA

Ukaz zasuče bite iz AKU1 za en bit v desno. 31. bit iz AKU1 se postavi na vrednost CC1,. Rezultat dobimo v AKU1. Vsebina AKU2 se ne spremeni. Pogojni kodi se postavita tako, da CC0 zavzame vrednost 0, CC1 pa vrednost zadnjega pomaknjenega bita (0. bita iz AKU1).

```
...     // prej       AKU1: 16#89AB_0F55  =  2#1000_1001_1010_1011_0000_1111_0101_0101

RRDA    //

...     // potem      AKU1: 16#X4D5_87AA  =  2#x100_0100_1101_0101_1000_0111_1010_1010
```

1.4 PRIMERJALNE OPERACIJE

1.4.1 Primerjava 16-bitnih predznačenih celih števil

```
==I
<>I
>I
>=I
<I
<=I
```

Izračun razlike AKU2 – AKU1, kjer se kot operanda upoštevata vsebini spodnjih besed akumulatorjev in to kot 16-bitni predznačeni celi števili. Izračunana razlika se NE zapiše NIKAMOR, se pa glede na predznak razlike postavita CC0 in CC1. Glede na uporabljeno primerjalno operacijo (ki jo je treba razumeti med AKU2 in AKU1) in ugotovljeno razliko se ustrezno postavi RLO, ki ga lahko uporabimo v nadaljnjih ukazih. Vsebini AKU1 in AKU2 ostaneta nespremenjeni.

CC0	CC1	Pomen (splošno)
0	0	Rezultat operacije je enak nič (= 0)
0	1	Rezultat operacije je pozitiven (> 0)
1	0	Rezultat operacije je negativen (< 0)
1	1	Rezultat operacije je neveljaven

Tabela 7.8: Pomen pogojnih kod CC0 in CC1

```
L   MW 10
L   IW 10
==I
= Q 4.0
>I
= Q 4.1
<I
= Q 4.2
```

V programu iz zgornjega zgleda bo tako lučka na izhodu `Q 4.2` gorela samo v primeru, če bo vsebina `MW 10` manjša od vsebine `IW 10`. Obe vsebini moramo seveda interpretirati kot 16-bitni predznačeni celi števili (integer).

1.4.2 Primerjava 32-bitnih predznačenih celih števil

```
==D

<>D

>D

>=D

<D

<=D
```

Izračun razlike AKU2 – AKU1, kjer se kot operanda upoštevata vsebini akumulatorjev in to kot 32-bitni predznačeni celi števili. Izračunana razlika se NE zapiše NIKAMOR, se pa glede na predznak razlike postavita CC0 in CC1. Glede na uporabljeno primerjalno operacijo (ki jo je treba razumeti med AKU2 in AKU1) in ugotovljeno razliko se ustrezno postavi RLO, ki ga lahko uporabimo v nadaljnjih ukazih. Vsebini AKU1 in AKU2 ostaneta nespremenjeni.

1.4.3 Primerjava realnih števil

```
==R

<>R

>R

>=R

<R

<=R
```

Izračun razlike AKU2 – AKU1, kjer se kot operanda upoštevata vsebini akumulatorjev in to kot realni števili, zapisani s plavajočo vejico na 32 bitih. Izračunana razlika se NE zapiše NIKAMOR, se pa glede na predznak razlike postavita CC0 in CC1. Glede na uporabljeno primerjalno operacijo (ki jo je treba razumeti med AKU2 in AKU1) in ugotovljeno razliko se ustrezno postavi RLO, ki ga lahko uporabimo v nadaljnjih ukazih. Vsebini AKU1 in AKU2 ostaneta nespremenjeni.

1.5 ARITMETIČNE OPERACIJE

1.5.1 Operacije nad 16-bitnimi predznačenimi celimi števili

Operanda sta vsebini spodnjih besed akumulatorjev in to kot 16-bitni predznačeni celi števili. Kot prvi operand nastopa AKU2, kot drugi pa AKU1. Rezultat se zapiše v AKU1 (praviloma kot 16-bitno celo število na spodnjo besedo, pri čemer ostane zgornja beseda nespremenjena), glede na njegov predznak se postavita CC0 in CC1. Vsebina AKU2 ostane nespremenjena.

`+I`

AKU2 + AKU1

`-I`

AKU2 – AKU1

`*I`

AKU2 * AKU1; rezultat je 32-bitno število v AKU1

`/I`

AKU2/AKU1; rezultat je kvocient kot 16-bitno število v spodnji besedi AKU1, ostanek pa je kot 16-bitno celo število v zgornji besedi AKU1

1.5.2 Operacije nad 32-bitnimi predznačenimi celimi števili

Operanda sta vsebini akumulatorjev in to kot 32-bitni predznačeni celi števili. Kot prvi operand nastopa AKU2, kot drugi pa AKU1. Rezultat se kot 32-bitno celo število zapiše v AKU1, glede na njegov predznak se postavita CC0 in CC1. Vsebina AKU2 ostane nespremenjena.

`+D`

AKU2 + AKU1

`-D`

AKU2 – AKU1

`*D`

AKU2 * AKU1; rezultat je 32-bitno celo število v AKU1

`/D`

AKU2/AKU1; rezultat je kvocient kot 32-bitno celo število v AKU1

`MOD`

AKU2/AKU1; rezultat je ostanek deljenja kot 32-bitno celo število v AKU1

1.5.3 Operacije nad realnimi števili

Operanda sta vsebini akumulatorjev in to kot 32-bitni realni števili s plavajočo vejico. Kot prvi operand nastopa AKU2, kot drugi pa AKU1. Rezultat se kot 32-bitno realno število zapiše v AKU1, glede na njegov predznak se postavita CC0 in CC1. Vsebina AKU2 ostane nespremenjena.

`+R`

AKU2 + AKU1

`-R`

AKU2 – AKU1

`*R`

AKU2 * AKU1; rezultat je 32-bitno realno število v AKU1

`/R`

AKU2/AKU1; rezultat je 32-bitno realno število v AKU1

1.5.4 Operacija prištevanja konstante

Če želimo prišteti konstantno 16-bitno predznačeno celo število k AKU1 (k njegovi spodnji besedi, interpretirani kot 16-bitno predznačeno celo število), lahko uporabimo ukaz:

```
+   konstanta
```

npr.:

```
+   15
```

ali za odštevanje

```
+   -1
```

Rezultat se kot 16-bitno celo število zapiše na spodnjo besedo AKU1, pri čemer se zgornja beseda AKU1 ne spremeni.

Za prištevanje konstantnega 32-bitnega predznačenega celega število k AKU1 (kjer njegovo vsebino obravnavamo kot 32-bitno predznačeno celo število), pa pri konstanti za ukazom to specificiramo takole:

```
+   L#15
```

ali za odštevanje

```
+   L#-1
```

Rezultat se kot 32-bitno celo število zapiše na AKU1.

Tovrste operacije prištevanja ne vplivajo niti na AKU2 niti na bite v statusni besedi.

1.5.5 Aritmetični operaciji INC in DEC

Večino aritmetičnih operacij bomo obravnavali v drugi knjigi. Tukaj omenimo le dve, ki ju je STEP 7 podedoval od STEP 5. Prav lahko bi shajali tudi brez njiju, saj so ju nadomestile zmogljivejše operacije, a ker sta zelo zabavni, si ju podrobneje oglejmo.

Operaciji INC in DEC sta namenjeni povečevanju oziroma zmanjševanju vrednosti akumulatorja AKU1 za neko konstantno vrednost. Pomembno je vedeti, da delujeta le na najnižji zlog akumulatorja, pri čemer ni prenosa (ali sposodka) na višje bite akumulatorja AKU1. Ti ostanejo nespremenjeni, prav tako pa operaciji ne vplivata na AKU2. Operaciji nista odvisni od statusnih bitov in nanje tudi ne vplivata.

Ukaza imata naslednjo obliko:

```
INC  konstanta
```

in

```
DEC  konstanta
```

Pri tem je konstanta poljubno desetiško število od 0 do 255. Za vrednost te konstante želimo povečati ali zmanjšati vsebino najnižjega zloga AKU1.

Za lažje razumevanje si oglejmo nekaj primerov.

```
L   +2208      // AKU1: 16#0000_08A0  =  B#(  0,  0,  8,160)  =   2208
INC   92       //              16#5C  =  B#(  0,  0,  0, 92)  =     92
T   MW 20      // AKU1: 16#0000_08FC  =  B#(  0,  0,  8,252)  =   2300
```

Na začetku v AKU1 naložimo (16-bitno) desetiško število 2208. Po ukazu povečanja (inkrementiranja) za 92 pa rezultat prenesemo na MW 20. In kakšen je rezultat? Takšen, kakršnega smo tudi pričakovali: 2300 vendar!

A do prave zabave še nismo prišli. Kaj pa, če bi želeli desetiško število iz prejšnjega primera povečati ne za 92, temveč za 114? Nič lažjega:

```
L   +2208      // AKU1: 16#0000_08A0  =  B#(  0,  0,  8,160)  =   2208
INC 114        //              16#72  =  B#(  0,  0,  0,114)  =    114
T   MW 20      // AKU1: 16#0000_0812  =  B#(  0,  0,  8, 18)  =   2066
```

Kako zdaj to? Številu smo nekaj pozitivnega prišteli, dobili pa smo celo manj, kot smo imeli na začetku! Mar so imeli prste vmes naši vrli bančniki in borzniki? Na srečo ne, le spomniti se moramo uvodoma izrečenih opozoril in omejitev: prenosa z najnižjega zloga na višje bite pri tej operaciji ni! "Pričakovani" rezultat 2322 je od dobljenega v tem primeru večji ravno za en preneseni bit, torej za 256.
Pri zadnjem zgledu nas zato ne bo več zaneslo: od desetiške konstante 555 odštejmo (dekrementirajmo) 55.

```
L   +555       // AKU1: 16#0000_022B  =  B#(  0,  0,  2, 43)  =    555
DEC   55       //              16#72  =  B#(  0,  0,  0, 55)  =     55
T   QW 0       // AKU1: 16#0000_02F4  =  B#(  0,  0,  2,244)  =    756
```

Je še kdo pričakoval rezultat 500?

1.6 MATEMATIČNE FUNKCIJE

Matematične funkcije se izvajajo kot absolutne operacije neodvisno od kakršnihkoli pogojev. Njihov operand se nahaja v AKU1 kot 32-bitno realno število, rezultat pa se tudi v takšnem formatu zapiše v AKU1, pri čemer se postavijo še ustrezni biti v statusni besedi.

1.6.1 Trigonometrične funkcije

`SIN`

izračuna sinus kota, podanega v radianih

`COS`

izračuna kosinus kota, podanega v radianih

`TAN`

izračuna tangens kota, podanega v radianih

`ASIN`

izračuna arkus sinus in poda vrednost kota v radianih

`COS`

izračuna arkus kosinus in poda vrednost kota v radianih

`ATAN`

izračuna arkus tangens in poda vrednost kota v radianih

1.6.2 Ostale matematične funkcije

`SQR`

izračuna kvadrat

`SQRT`

izračuna kvadratni koren

`EXP`

izračuna eksponentno funkcijo e^{AKU1}.
Za poljubno potenciranje je uporaben izraz $a^b = e^{b \cdot \ln(a)}$

`LN`

izračuna naravni logaritem ln(*AKU1*).
Za poljubno logaritemsko osnovo se poslužimo izraza $\log_b a = \ln(a)/\ln(b)$

1.7 OPERACIJE PRETVARJANJA

1.7.1 BTI

```
BTI
```

(BCD To Integer) Operacija vzame za operand vsebino spodnje besede AKU1, ki jo obravnava kot BCD število, podano z vrednostjo na spodnjih 12 bitih (3 nibbli) in s predznakom (o tem govorijo biti od 15 – 12; če so vsi enaki "1", je število negativno, če pa so enaki "0", je število pozitivno). Rezultat se zapiše v spodnjo besedo AKU1 kot 16-bitno predznačeno celo število. Zgornja beseda AKU1 se ne spremeni.
Ukaz ne vpliva na statusno besedo.
Če operand ni veljavno BCD število, operacijski sistem krmilnika poskuša klicati organizacijski blok OB 121. Če tega bloka ni, se obdelava uporabniškega krmilnega programa ustavi.

```
...     // prej       AKU1: 16#89AB_0015  =  2#1000_1001_1010_1011_0000_0000_0001_0101

BTI    //

...     // potem      AKU1: 16#89AB_000F  =  2#1000_1001_1010_1011_0000_0000_0000_1111
```

1.7.2 BTD

```
BTD
```

(BCD To Double) Operacija vzame za operand vsebino AKU1, ki ga obravnava kot BCD število, podano z vrednostjo na spodnjih 28 bitih (7 nibblov) in s predznakom na zgornjih štirih bitih. Rezultat se zapiše v AKU1 kot 32-bitno predznačeno celo število.
Ukaz ne vpliva na statusno besedo.
Če operand ni veljavno BCD število, operacijski sistem krmilnika poskuša klicati organizacijski blok OB 121. Če tega bloka ni, se obdelava uporabniškega krmilnega programa ustavi.

1.7.3 ITB

```
ITB
```

(Integer To BCD) Operacija vzame za operand vsebino spodnje besede AKU1, ki jo obravnava kot 16-bitno predznačeno celo število. Rezultat se zapiše v spodnjo besedo AKU1 kot BCD število s števkami na spodnjih treh tetradah in predznakom na bitih 15 – 12. Zgornja beseda AKU1 se ne spremeni.

Če je operand po absolutni vrednosti večji od 999, se postavita statusna bita OV in OS, pretvorba pa se ne izvrši.

```
...     // prej        AKU1: 16#89AB_FE63  =  2#1000_1001_1010_1011_1111_1110_0110_0011

ITB    //

...     // potem       AKU1: 16#89AB_F413  =  2#1000_1001_1010_1011_1111_0100_0001_0011
```

1.7.4 DTB

```
DTB
```

(Double To BCD) Operacija vzame za operand vsebino AKU1, ki jo obravnava kot 32-bitno predznačeno celo število. Rezultat se zapiše v AKU1 kot BCD število s števkami na spodnjih sedmih tetradah in predznakom na bitih 31 – 28. Zgornja beseda AKU1 se ne spremeni.
Če je operand po absolutni vrednosti večji od 9.999.999, se postavita statusna bita OV in OS, pretvorba pa se ne izvrši.

1.7.5 ITD

```
ITD
```

(Integer To Double) Operacija vzame za operand vsebino spodnje besede AKU1, ki jo obravnava kot 16-bitno predznačeno celo število. Rezultat se zapiše v AKU1 kot 32-bitno predznačeno celo število, kar pomeni, da se predznak operanda z bita 15 prenese še v bite 16-31.
Statusna beseda se pri tem ukazu ne spremeni.

1.7.6 DTR

```
DTR
```

(Double To Real) Operacija vzame za operand vsebino AKU1, ki jo obravnava kot 32-bitno predznačeno celo število. Rezultat se zapiše v AKU1 kot 32-bitno realno število.
Statusna beseda se pri tem ukazu ne spremeni.

1.7.7 RND+

`RND+`

Operacija vzame za operand vsebino AKU1, ki jo obravnava kot 32-bitno realno število. Rezultat se zapiše v AKU1 kot navzgor zaokroženo 32-bitno predznačeno celo število.
Če bi rezultat presegal obseg 32-bitnih predzančenih celih števil ali če operand ni veljavno realno število, se postavita bita OV in OS v statusni besedi, pretvorba pa se ne izvrši.

1.7.8 RND–

`RND-`

Operacija vzame za operand vsebino AKU1, ki jo obravnava kot 32-bitno realno število. Rezultat se zapiše v AKU1 kot navzdol zaokroženo 32-bitno predznačeno celo število.
Če bi rezultat presegal obseg 32-bitnih predzančenih celih števil ali če operand ni veljavno realno število, se postavita bita OV in OS v statusni besedi, pretvorba pa se ne izvrši.

1.7.9 RND

`RND`

Operacija vzame za operand vsebino AKU1, ki jo obravnava kot 32-bitno realno število. Rezultat se zapiše v AKU1 kot najbližje zaokroženo 32-bitno predznačeno celo število. Če bi rezultat bil natančno med sodim in lihim številom, prevlada zaokrožitev proti sodemu številu.
Če bi rezultat presegal obseg 32-bitnih predzančenih celih števil ali če operand ni veljavno realno število, se postavita bita OV in OS v statusni besedi pretvorba pa se ne izvrši.

1.7.10 TRUNC

`TRUNC`

Operacija vzame za operand vsebino AKU1, ki jo obravnava kot 32-bitno realno število. Rezultat se zapiše v AKU1 kot 32-bitno predznačeno celo število brez upoštevanja decimalnega dela. Če bi rezultat bil natančno med sodim in lihim številom, prevlada zaokrožitev proti sodemu številu.
Če bi rezultat presegal obseg 32-bitnih predzančenih celih števil ali če operand ni veljavno realno število, se postavita bita OV in OS v statusni besedi, pretvorba pa se ne izvrši.

1.7.11 INVI

```
INVI
```

Operacija vzame za operand vsebino spodnje besede AKU1, nad katero napravi eniški komplement (ničle postanejo enice, enice pa ničle). Rezultat se zapiše v spodnjo besedo AKU1, zgornja beseda AKU1 pa se ne spremeni.
Operacija nima vpliva na statusno besedo.

```
...     // prej       AKU1: 16#89AB_0015  =  2#1000_1001_1010_1011_0000_0000_0001_0101

INVI    //

...     // potem      AKU1: 16#89AB_FFEA  =  2#1000_1001_1010_1011_1111_1111_1110_1010
```

1.7.12 INVD

```
INVD
```

Operacija vzame za operand vsebino AKU1, nad katero napravi eniški komplement (ničle postanejo enice, enice pa ničle). Rezultat se zapiše v AKU1.
Operacija nima vpliva na statusno besedo.

1.7.13 NEGI

```
NEGI
```

Operacija vzame za operand vsebino spodnje besede AKU1, nad katero napravi dvojiški komplement. Rezultat se zapiše v spodnjo besedo AKU1, zgornja beseda AKU1 pa se ne spremeni. Če operand in rezultat obravnavamo kot 16-bitni predznačeni celi števili, pomeni ta operacija negacijo oz. spremembo predznaka.
Operacija s svojim rezultatom vpliva na bite CC0, CC1, OV in OS v statusni besedi.

```
...     // prej       AKU1: 16#89AB_0001  =  2#1000_1001_1010_1011_0000_0000_0000_0001

NEGI    //

...     // potem      AKU1: 16#89AB_FFFF  =  2#1000_1001_1010_1011_1111_1111_1111_1111
```

1.7.14 NEGD

`NEGD`

Operacija vzame za operand vsebino AKU1, nad katero napravi dvojiški komplement. Rezultat se zapiše v AKU1. Če operand in rezultat obravnavamo kot 32-bitni predznačeni celi števili, pomeni ta operacija negacijo oz. spremembo predznaka.
Operacija s svojim rezultatom vpliva na bite CC0, CC1, OV in OS v statusni besedi.

1.7.15 NEGR

`NEGR`

Operacija vzame za operand vsebino AKU1 kot 32-bitno realno število, ki mu spremeni predznak. Rezultat se kot 32-bitno realno število zapiše v AKU1.
Operacija nima vpliva na statusno besedo.

1.7.16 ABS

`ABS`

Operacija vzame za operand vsebino AKU1 kot 32-bitno realno število, ki mu postavi predznak mantise na "0" in na ta način izračuna absolutno vrednost. Rezultat se kot 32-bitno realno število zapiše v AKU1. Operacija nima vpliva na statusno besedo.

1.8 POSREDNO NASLAVLJANJE

V dosedanjih zgledih in obravnavi (poglavje 7.4) smo srečali takojšnje naslavljanje (*immediate addressing*), kjer je operand kot neka konkretna vrednost vključen pri samem ukazu, in neposredno naslavljanje (*direct addressing*), pri katerem nastopa operand kot vsebina, zapisana na določeni lokaciji. Pri neposrednem naslavljanju lahko vsebino uporabljenega naslova sicer spreminjamo, vendar ostaja naslov vseskozi nespremenjen, torej takšen, kakršnega smo določili pri pisanju programa: sestavljen iz identifikatorja naslova (področja: I, Q, PI, M, DB), podatkovne širine (brez posebne oznake – bit, B – zlog, W – beseda, D – dvojna beseda) in lokacije identifikatorja, npr.: `I 1.6`, `QB 4`, `MD 20`... Ne smemo pa pozabiti, da smo neposredno naslavljali tudi časovnike, števce, funkcijske bloke (npr. `T 4`, `C 7`, `FB 12`)..., pri čemer je bila podatkovna širina odvisna od ukaza (povpraševanje po elementu ali nalaganje njegove vsebine v akumulator).

Včasih pa si želimo, da bi kakšen naslov operanda lahko določili šele med izvajanjem programa na krmilniku in da bi ga morebiti tudi spreminjali. Tovrsten pristop je možen s pomočjo posrednega naslavljanja (*indirect addressing*). V tem poglavju se bomo na kratko dotaknili samo **pomnilniškega posrednega naslavljanja**; podrobnosti o delu z naslovnima registroma AR1 in AR2 bomo prihranili za drugo knjigo.

Pri pomnilniškem posrednem naslavljanju imamo v pomnilniku izbrano lokacijo, katere vsebino razumemo kot naslov. To vsebino lahko izračunamo oziroma določimo med samim delovanjem programa. Za začetek si oglejmo, kako bi posredno naslovili nek števec ali časovnik. Vsak izmed njih je natančno določen s svojo naslovno številko, ki jo lahko zapišemo kot celo število, npr. `C 41`. Za zapis takšne naslovne številke pri števcih, časovnikih, funkcijskih blokih, funkcijah ali podatkovnih blokih potrebujemo 16-bitno pomnilniško lokacijo, torej eno besedo. Odločimo se torej, da bomo naslovno številko števca nekako zapisali v pomožno pomnilniško besedo `MW 20`. Če želimo nasloviti števec `C 41`, moramo v `MW 20` naložiti vrednost 41, tako da bo vsebina naslednja:

```
MW 20:   2#0000_0000_0010_1001  =  16#0029  =  41
```

V programu lahko potem poskrbimo za štetje navzdol z naslednjim zapisom:

```
...
CD  C[MW 20]    // štetje navzdol števca, katerega naslov...
                // ...je definiran z vsebino MW 20
...
```

Vsebino lokacije, zapisane v oglatem oklepaju, razumemo kot naslov, v konkretnem primeru kot naslov števca. Na enak način lahko uporabimo tudi druge ukaze, ki se nanašajo na števec. S spremembo vsebine `MW 20` bi dosegli, da bi se v programu štetje navzdol izvedlo na nekem drugem števcu!

Podobno bi lahko zapisali npr. ukaz za sprožitev časovniške funkcije podaljšanje pulza na izbranem časovniku:

```
A   I 0.1
SE  T[MW 40]    // podaljšanje pulza na časovniku z naslovom,
...             // ... definiranim z vsebino MW 40
```

Če bi bila vsebina `MW 40` naslednja:

```
MW 40:   2#0000_0000_0000_1010  =  16#000A  =  10
```

bi se ob ustrezni spremembi RLO-ja sprožila izbrana časovna funkcija na časovniku `T 10`.

Podobno bi izvedli pogojni klic izbranega funkcijskega bloka:

```
A   M 14.1
CC  FB[MW 8]    // klic funkcijskega bloka z naslovom,
...             // ... definiranim z vsebino MW 8
```

Ustrezna vsebina `MW 8`

```
MW 8:   2#0000_0000_0001_0000  =  16#0010  =  16
```

bi poskrbela, da bi ob setiranem bitu `M 14.1` klicali funkcijski blok `FB 16`.

Zdaj tudi razumemo, zakaj takšnemu načinu naslavljanja pravimo posredno naslavljanje. Operand namreč **ni** vsebina pomnilniške lokacije, ki je navedena v oglatem oklepaju! Ta vsebina pomnilniške lokacije namreč pomeni **naslov**, in šele vsebino s tega naslova bomo uporabili kot operand!

Z vpisom naslovne številke na neko 16-bitno pomnilniško lokacijo lahko posredno naslavljamo samo tiste elemente, kjer je naslov določen s to naslovno številko: števce, časovnike, funkcijske bloke, funkcije in podatkovne bloke.

Posredno naslavljanje vhodov (I), izhodov (Q), pomožnih pomnilnikov (M), perifernih vhodov in izhodov ter podatkov v podatkovnih blokih pa je nekoliko bolj zapleteno. Pri tovrstnih elementih lahko namreč poljubno naslovimo en sam bit, zlog, besedo ali dvojno besedo.

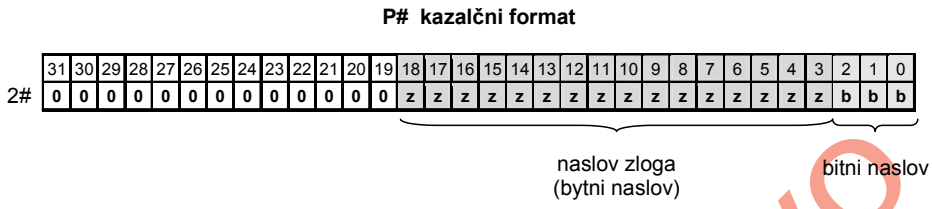
Naslov posameznega bita za identifikatorjem naslova smo v splošnem označili kot

```
byte.bit
```

kjer je `byte` naslov zloga (byta) in `bit` naslov bita v tem zlogu. Bitni naslov zavzema vrednosti od 0 do 7, bytni naslov pa je lahko od 0 do 65535. Navedimo nekaj že znanih primerov:

```
I 1.6          Q 124.2          M 16.0
```

Da na neki lokaciji zapišemo naslov takšnega elementa, potrebujemo 32 bitov. Tovrstnemu zapisu naslova pravimo **kazalec** (pointer), ki ima naslednji format (slika 7.16):



Slika 7.16: Format 32-bitnega kazalca (pointerja)

Na spodnjih treh bitih (biti od 0 do 2) je v obliki dvojiškega števila podan bitni naslov, naslednjih šestnajst bitov (biti od 3 do 18) pa služi zapisu bytnega naslova. Le-ta je prav tako podan v obliki dvojiškega števila, pri čemer kot 16-bitno besedo opazujemo samo vsebino bitov od 3 do 18!

Kazalce za naslove v zgornjih treh primerih (`I 1.6`, `Q 124.2`, `M 16.0`) preprosto zapišemo kot:

```
P#1.6    =  16#0000_0000_0000_000E  =  2#0000_0000_0000_0000_0000_0000_0000_1110
P#124.2  =  16#0000_0000_0000_03E2  =  2#0000_0000_0000_0000_0000_0011_1110_0010
P#16.0   =  16#0000_0000_0000_0080  =  2#0000_0000_0000_0000_0000_0000_1000_0000
```

Takšne vsebine bi zapisali na neke 32-bitne lokacije, npr. na dvojne pomnilniške besede `MD 40`, `MD 44` in `MD 48`. Posredno naslavljanje bi potem opravili takole:

```
I [MD 40]          Q [MD 44]          M [MD 48]
```

Napišimo program, ki bo izvajal logično povezavo IN med dvema vhodnima bitoma. Kateri vhod bo prvi operand, bomo določili v `MD 16`, kateri pa drugi, bomo zapisali v `MD 20`. Rezultat opazujmo na izhodnem bitu, ki je definiran z vsebino `MD 24`!

```
A    I[MD 16]     // povpraševanje po prvem vhodu
A    I[MD 20]     // povpraševanje po drugem vhodu
=    Q[MD 24]     // rezultat na izhod
```

Kaj moramo storiti, da se bo logična povezava IN izvajala nad vhodoma `I 0.3` in `I 1.7`, rezultat pa naj bo na `Q 4.4`? Na nek način moramo na navedene pomnilniške dvojne besede naložiti (ali izračunati) naslednje vsebine:

```
MD 16:    P#0.3    =  16#0000_0000_0000_0003  =  2#0000_0000_0000_0000_0000_0000_0000_0011
MD 20:    P#1.7    =  16#0000_0000_0000_000F  =  2#0000_0000_0000_0000_0000_0000_0000_1111
MD 24:    P#4.4    =  16#0000_0000_0000_0024  =  2#0000_0000_0000_0000_0000_0000_0010_0100
```

To bi lahko izvedli preko posebnih nadzornih funkcij s pomočjo programirne naprave (Monitor/Modify Variables). V programu pa bi bila možna rešitev takšna:

```
L    P#0.3        // priprava kazalca za prvi vhod
T    MD 16
L    P#1.7        // priprava kazalca za drugi vhod
T    MD 20
L    P#4.4        // priprava kazalca za izhod
T    MD 24
A    I[MD 16]     // povpraševanje po prvem vhodu
A    I[MD 20]     // povpraševanje po drugem vhodu
=    Q[MD 24]     // rezultat na izhod
```

Z navedenimi vsebinami `MD 16`, `MD 20` in `MD 24` bi program izvajal povsem enako nalogo kot naslednji ukazi:

```
A    I 0.3        // povpraševanje po prvem vhodu
A    I 1.7        // povpraševanje po drugem vhodu
=    Q 4.4        // rezultat na izhod
```

A ponovimo še enkrat, da ne bo pomote: prednost posrednega naslavljanja je ravno možnost izbiranja oziroma določanja operandov med obratovanjem! To bomo lepo ilustrirali z zgledom, kjer bomo srečali tudi posredno naslavljanje lokacij z večjo podatkovno širino.

Naloga zahteva, da se ob pritisku na tipko `I 0.0` na vse zloge pomožnega pomnilnika od `MB 0` do vključno `MB 10` zapiše vsebina, ki se nahaja na vhodnem zlogu `IB 1`.

Ker se mora (in sme) nekaj zgoditi samo ob pritisku na tipko, bo treba uporabiti dinamični člen. Prav nam bo prišla tudi zanka, ki se bo ponovila 11-krat, saj je toliko zlogov pomožnega pomnilnika od `MB 0` do `MB 10`. Prav naslov zloga pomožnega pomnilnika pa je v našem primeru tisto, kar bomo morali spreminjati. Zato rezervirajmo dvojno besedo `MD 20` kot lokacijo, kjer bo zapisan naslov zloga (kazalec), kamor bo v danem trenutku treba zapisati vsebino z vhodnega zloga `IB 1`. Potrebujemo tudi števec za ponovitve, ki naj bo na `MW 30`.

Po dinamičnem členu na začetku programa je treba poskrbeti, da bo na `MD 20` zapisan kazalec oziroma naslov pomnilniškega zloga `MB 0`. Pripadajoči kazalec je `P#0.0`. Pri posrednem naslavljanju širših podatkovnih enot (zlog, beseda, dvojna beseda) **moramo obvezno** vpisati za bitni naslov pri kazalcu vrednost 0. Bytni naslov pa je naslovna številka tistega zloga podatkovne enote, ki ima najvišjo težo. Pri naslavljanju zlogov je tako bytni naslov kar naslov dotičnega zloga, če želimo posredno nasloviti besedo ali

dvojno besedo, pa vzamemo za bytni naslov številko najvišjega zloga; kot že dobro vemo, je ta številka po vrednosti znotraj besede ali dvojne besede najmanjša!

Po začetni vrednosti kazalca moramo določiti še, kolikokrat se bo zanka ponovila. V zanki pa v AKU1 naložimo vsebino vhodnega zloga `IB 0` ter jo prenesemo na pomnilniški zlog, katerega naslov je določen z vsebino `MD 20`, torej na `MB[MD 20]`. Zatem je treba naslov pomnilniškega zloga povečati za 1. Ker zaenkrat ne poznamo drugih aritmetičnih operacij, uporabimo ukaz `INC`. Zanj veljajo nekatere omejitve (poglavje 7.6), ki jih ne smemo prezreti; no v našem primeru zaradi majhnega obsega ne pričakujemo težav.
Zakaj pa smo zapisali `INC 8` in ne `INC 1`, kot bi pričakovali na prvi pogled? Prav zato, ker moramo povečati naslov zloga, ne pa bitnega naslova! In povečanje vsebine akumulatorja AKU1 za 8 bo zaradi načina zapisa kazalca rezultiralo s povečanjem bytnega naslova za 1. Tako izračunani kazalec seveda shranimo na svoje mesto (`MD 20`).
Na koncu v AKU1 naložimo še števec ponovitev (`MW 30`); tega v skladu z našimi zahtevami obdela ukaz `LOOP`. Tako nam je z nekaj spretnosti uspelo realizirati posredno naslavljanje, ki bi mu v konkretnem primeru lahko rekli tudi indeksno naslavljanje.

```
      A    I 0.0       // povprašaj po tipki na vhodu I 0.0
      FP   M 40.0      // dinamični člen; POZOR: prir. tabela!
      JC   SKOK        // če je pritisk, skoči na akcijo
      BEU              // sicer končaj

SKOK: L    P#0.0       // začetni kazalec za MB 0
      T    MD 20

      L    +11         // število ponovitev zanke
ZANK: T    MW 30       // shrani (novo) stanje števca ponovitev

      L    IB 1        // naloži vsebino vhodnega zloga IB 0
      T    MB[MD 20]   // in jo prenesi na trenutno izbrani MB

      L    MD 20       // naloži kazalec...
      INC  8           // ga ustrezno povečaj !!!
      T    MD 20       // in pripravi za naslednji

      L    MW 30       // naloži števec ponovitev
      LOOP ZANK        // zmanjšaj št. ponovitev in ponovi
      BE
```

Ponovimo opozorilo, da moramo pri posrednem naslavljanju zloga, besede ali dvojne besede bitni naslov v kazalcu postaviti na 0. To pa nas ne ovira, da z istim kazalcem ne bi mogli nasloviti tudi bita z bitnim naslovom 0; seveda imamo v mislih bit 0 v zlogu, definiranem z bytnim naslovom.

Zavedati se moramo, da lahko pri posrednem naslavljanju napravimo cel kup napak. Zgodi se nam, da izračunamo ali vpišemo neveljaven naslov, spet drugič je naslov veljaven, vendar takšne enote na krmilniku ni, tretjič pa lahko z izračunanim naslovom "povozimo" kakšno lokacijo, ki jo sicer uporabljamo za neke druge namene.

Po svoje je problematično tudi spreminjanje naslova med obratovanjem. Če npr. posredno naslovljenemu pomožnemu pomnilniku priredimo (in ne setiramo!) rezultat neke logične funkcije, ki je v danem trenutku "1", potem pa s spremenjeno vsebino iste lokacije posredno naslovimo nek drug pomožni pomnilnik, se zgodi, da prej naslovljeni pomožni pomnilnik ostane na vrednosti "1", dasiravno ga nismo setirali v pravem pomenu besede! Zato moramo biti pri posrednem naslavljanju in delu s kazalci zelo previdni in v krmilju vedno sprogramirati možnost reševanja takšnih ("nepredvidenih") situacij.

Za konec si oglejmo še nekaj primerov posrednega naslavljanja:

```
L    P#124.0      // prvi kazalec
T    MD 40        // lokacija s kazalcem
L    P#4.0        // drugi kazalec
T    MD 44        // lokacija s kazalcem
L    P#64.0       // tretji kazalec
T    MD 48        // lokacija s kazalcem
...
...
L    IW[MD 40]    // naloži vhodno besedo IW 124
AW   W#16#F0F0    // operacija AW
T    QW[MD 44]    // rezultat prenesi na QW 4
...
...
L    ID 0         // naloži dvojno vhodno besedo ID 0
T    MD[MD48]     // rezultat prenesi na MD 64
...
```

1.9 PODATKOVNI BLOKI

Pri številnih krmiljenih procesih potrebujemo poleg krmilnega programa še različne podatke, pa naj bodo to časi trajanja posameznih faz postopka (npr. odprtja vsake izmed smeri prometnih tokov v semaforiziranem križišču), količine različnih snovi pri mešalnih procesih ali kakšni drugi parametri, ki so odvisni od tipa izdelka, kakršen je trenutno v proizvodnji. Če bi te podatke zapisovali neposredno v krmilnem programu skupaj z ukazi, bi njihovo spreminjanje ob uvedbi nekega novega izdelka bilo dokaj zahtevno in nepregledno tudi za veščega programerja. Ker pa npr. pri posameznih receptih pričakujemo pogoste spremembe, je seveda elegantnejša možnost, da v okviru pristojnosti prepustimo izbiro recepta in njegovo morebitno spreminjanje kar ustrezno usposobljenemu operaterju. Zato lahko tovrstne podatke zapisujemo v posebne namenske bloke – t.i. podatkovne bloke. Še več – tudi v primerih, ko želimo zabeležiti določena dosežena stanja procesa v izbranih trenutkih, lahko to prav tako s pomočjo krmilnega programa zapišemo v pripadajoče podatkovne bloke.

Na krmilnikih srečamo dve različici podatkovnih blokov: globalne podatkovne bloke, do katerih lahko dostopamo iz vseh uporabniških funkcijskih blokov in funkcij, ter instančne podatkovne bloke, ki so namenjeni prenašanju parametrov pri klicu izbranega funkcijskega bloka. Več podrobnosti o uporabi instančnih podatkovnih blokov je na voljo v poglavju o parametriranih funkcijskih blokih.

Podatki v podatkovnih blokih so organizirani po bitih, bytih (zlogih), 16-bitnih besedah (W) in 32-bitnih dvojnih besedah (D), podobno, kot je bilo obrazloženo pri naslavljanju pomožnih pomnilnikov, vhodov in izhodov. Število razpoložljivih podatkovnih blokov pri centralni procesni enoti CPU315-DP je 255 in jih lahko naslavljamo kot globalne podatkovne bloke od DB 1 do DB 255. Za obravnavanje podatkovnih blokov ima CPU dva posebna registra: DB register, namenjen pretežno za delo z globalnimi podatkovnimi bloki, in DI register, ki je praviloma posvečen opravilom z instančnimi podatkovnimi bloki. Funkcionalno pa med registroma ni nikakršne razlike; tako lahko (če v posameznih ukazih ne uporabljamo polnega naslova podatkov iz konkretnega podatkovnega bloka) sočasno dostopamo do dveh podatkovnih blokov: do enega prek DB registra, do drugega pa prek DI registra.

V odvisnosti od uporabljenega registra (DB ali DI) za dostop do podatkovnega bloka pri identifikatorju naslova srečamo naslednje oznake:

Podatkovna širina	**DB register**	**DI register**
bit	`DBX 20.3`	`DIX 20.3`
byte (zlog)	`DBB 20`	`DIB 20`
16-bitna beseda (word)	`DBW 20`	`DIW 20`
32-bitna beseda (doubleword)	`DBD 20`	`DID 20`

Podobno kot pri ostalih identifikatorjih naslova v zapisu sledi naslov zloga (bytni naslov), v konkretnem primeru iz zgornje tabele je to 20; pri naslavljanju posameznih bitov pa za piko še bitni naslov.

Nekatere sistemske funkcije SFC, s katerimi razpolaga CPU, omogočajo kreiranje, brisanje, kopiranje in druge operacije nad podatkovnimi bloki. Pogosto pa pri delu podatkovne bloke ustrezno pripravimo kar z urejevalnikom na programirni napravi (osebnem računalniku), jih inicializiramo in potem naložimo na krmilnik – podobno kot npr. funkcijske bloke.

1.9.1 Ukazi za delo s podatkovnimi bloki

Če ne uporabljamo polnega naslova za dostop do podatkov v podatkovnem bloku, moramo podatkovni blok najprej "odpreti" prek enega izmed registrov DB ali DI. Vsi nadaljnji ukazi v programu se potem nanašajo na odprti podatkovni blok, seveda do ukazov, ki sledijo odprtju kakšnega drugega podatkovnega bloka.

Odpiranje podatkovnega bloka prek DB registra poteka z ukazom:

```
OPN  DB 15   // odpre podatkovni blok DB 15
```

Podatkovni blok DI 20 pa odpremo prek DI registra z ukazom:

```
OPN  DI 20   // odpre podatkovni blok DI 20
```

Vsebino z izbranega naslova iz podatkovnega bloka lahko naložimo v AKU1 z ukazom `L`, vsebino iz AKU1 pa na izbrano lokacijo v podatkovnem bloku prenesemo z ukazom `T`.

```
OPN  DB 10   // odpre podatkovni blok DB 10
L    DBW 8   // naloži v AKU1 vsebino 8. besede iz DB 10
+    8       // vsebino poveča za 8
OPN  DB 12   // odpre podatkovni blok DB 12
T    DBW 14  // prenese vsebino AKU1 na 14. besedo iz DB 12
```

Povsem enak učinek bi dosegli z uporabo polnih naslovov pri ukazih:

```
L    DB10.DBW 8   // naloži vsebino 8. besede iz DB 10
+    8            // vsebino poveča za 8
T    DB12.DBW 14  // prenese vsebino na 14. besedo iz DB 12
```

Seveda veljajo pri teh operacijah vse značilnosti glede podatkovne širine, ki smo jih spoznali pri delu z akumulatorji.

Binarne operacije pa nam omogočajo tudi dostop do posameznih bitov v podatkovnem bloku:

```
OPN   DB 10    // odpre podatkovni blok DB 10
A     I 0.0    // če je vhod I 0.0
A     DBX 3.4  // in če je bit 3.4 iz DB 10
OPN   DB 12    // odpre podatkovni blok DB 12
=     DBX 0.7  // priredi RLO bitu 0.7 iz DB 12
```

Z zapisom polnih naslovov bi se program glasil takole:

```
A     I 0.0          // če je vhod I 0.0
A     DB10.DBX 3.4   // in če je bit 3.4 iz DB 10
=     DB12.DBX 0.7   // priredi RLO bitu 0.7 iz DB 12
```

Na voljo pa so tudi ukazi, s katerimi v AKU1 naložimo ključne informacije o odprtem podatkovnem bloku (ločeno za odprtje prek registra DB in DI):

```
L     DBLG     // naloži v AKU1 dolžino podatkovnega bloka,
               // odprtega prek registra DB (v bytih)


L     DILG     // naloži v AKU1 dolžino podatkovnega bloka,
               // odprtega prek registra DI (v bytih)


L     DBNO     // naloži v AKU1 naslov podatkovnega bloka,
               // odprtega prek registra DB


L     DINO     // naloži v AKU1 naslov podatkovnega bloka,
               // odprtega prek registra DI
```