Kiterjesztett pontosságú műveletek

A fejezet tartalma:

Az előző fejezetekben 8 bites műveletekkel foglalkoztunk. Természetesen előfordulhatnak olyan esetek, amikor ennél nagyobb bitszámú adatokkal kell műveleteket végeznünk. Minél nagyobb számokkal dolgozunk, vagy minél pontosabb számábrázolásra van szükség, annál több számjegyre, annál több bitre van szükségünk a tároláshoz. **Általánosságban N biten 0-tól 2^N-1-ig terjedő számokat tudunk ábrázolni**, ha előjel nélküli számábrázolásról van szó.

Emlékeztetőül az alábbi táblázatban felsoroltuk azokat a változótípusokat amelyekkel ebben a jegyzetben találkozunk. Mint látjuk, 8 bites bájtokkal, valamint 16 és 32 bites szavakkal lesz dolgunk.

Méret	Bájtok	Ábrázolási tartomány (előjel nélkül)	C típus (PIC18 fordító)	C típus (e jegyzetben)
8-bit	1	0 - 2^8-1 = 0 - 255	unsigned char	uint8
16-bit	2	0 - 2^{16}-1 = 0 - 65 535	unsigned int	uint16
32-bit	4	0 - 2^{32}-1 = 0 - 4 294 967 295	unsigned long	uint32

A 16 vagy 32 bites szavak legalacsonyabb helyiértékű bájtját **LSB**-vel jelöljük (a Least Significant Byte elnevezés alapján), a legmagasabb helyiértékű bájtjukat pedig **MSB**-vel (Most Significant Byte).

A PIC18 mikrovezérlők felépítéséből következően a több-bájtos regisztereknél (pl. **PCL**, **FSRx**, **PROD**, **TBLPTR**) tárolási módja "little endian", vagyis mindig az alacsonyabb memória címen található az alacsonyabb helyiértékű bájt/szó. A C18 fordító is ezt a sorrendet használja a több-bájtos változók memóriában történő elhelyezésénél, így kézenfekvő a választás, hogy az assembly programoknál is ezt a konvenciót kövessük.

Több-bájtos változók inicializálása

Az alábbi programban bemutatjuk, hogy a több-bájtos változók esetén hogyan végezhetjük el a kezdeti érték beállítását (inicializálás). A helyfoglalásnál az **i res 2** direktíva két, egymást követő bájtot foglal le, a **k res 4** pedig négy bájtot.

```
#include "p18f14k50.inc"
; FIGYELEM! A projekthez csatolni kell a 18f14k50.lkr állományt is!
   udata_acs                    ; C nyelven:
i  res 2                        ; uint16 i;
k  res 4                        ; uint32 k;

   CODE
;--- C nyelven: i = 0xC428;
   movlw 0x28
   movwf i         ; i LSB = 0x28
   movlw 0xC4      ;
   movwf i+1       ; i MSB = 0xC4
;--- C nyelven: k = 0xAF459BC0;
   movlw 0xC0
   movwf k         ; k LSB    = 0xC0
   movlw 0x9B
   movwf k+1       ; k 2.bájt = 0x9B
   movlw 0x45
   movwf k+2       ; k 3.bájt = 0x45
   movlw 0xAF
   movwf k+3       ; k MSB    = 0xAF
```

```
    goto $          ; végtelen ciklus
    END
```

Az **i** szimbolikus név a kisebb helyiértékű bájt memóriabeli címére mutat, a **movlw 0x28** és **movwf i** utasítások az **i** változó alacsony helyiértékű bájtját inicializálják. Az i+1 kifejezés az **i** változó magasabb helyiértékű bájtjára mutató címet jelent.

A C nyelvű **k = 0xAF459BC0;** utasítás négy movlw/movwf utasításpárt igényel, egy-egy pár a k változó minden egyes bájtjának beállításához. Az, hogy milyen sorrendben töltjük fel a bájtokat, teljesen közömbös, csak az a fontos, hogy mindegyik adat a megfelelő bájtba kerüljön.

Megjegyzés: Linker állomány nélküli projektnél a több-bájtos helyfoglalást az alábbiak szerint végezhetjük el:

```
#include "p18f14k50.inc"
    CBLOCK 0x000                ; C nyelven:
    i:2                         ; uint16 i;
    k:4                         ; uint32 k;
    ENDC

    ORG 0x000
;--- C nyelven: i = 0xC428;
    movlw 0x28
    movwf i         ; i LSB = 0x28
    ... stb.
```

Több-bájtos bitenkénti logikai műveletek

Mivel a bitenkénti logikai műveletek szigorúan csak az azonos helyiértékű bitek között történnek, s nem keletkezik átvitel, így a több-bájtos változók alacsonyabb és magasabb helyiértékű bájtjain egyenként is elvégezhetjük a műveletet, s a sorrend is közömbös. Az alábbi példán a bitenkénti AND művelet elvégzését mutatjuk be.

C nyelven

```
Uint16 i,j;

  i = i&j;

// további utasítások
```

Assembly nyelven

```
    Udata_acs
i   res 2
j   res 2
    CODE
    movf  j,w
    andwf i,F       ; i = i & j (LSB)
    movf  j+1,w
    andwf i+1,F     ; i = i&j (MSB)
; további utasítások
```

16 bites összeadás és kivonás

Összeadásnál vagy kivonásnál először a változók alacsonyabb helyiértékű felén végezzük el a műveletet, s a keletkező átvitelt vagy áthozatot figyelembe vesszük a változók magasabb helyiértékű felén végzendő műveletnél.

Összeadás

```
   0x 34  F0
+  0x 22  40
Carry:    1
-----------
   0x 57  30
```

Kivonás

```
   0x 34  10
-  0x 22  40
Borrow:  -1
-----------
   0x 11  D0
```

A fenti műveletek assembly programozásához két új utasítást is felhasználunk:
ADDWFC f hatása: f = f + WREG + C, vagyis az f változóhoz WREG és Carry bit tartalmát is hozzáadja.
SUBWFB f hatása: f = f - WREG - B, vagyis az f változóból kivonja WREG és a Borrow bit tartalmát.
A Borrow bit egyébként a Carry bit negáltja: B = ~C.

C nyelven

```
Uint16 i,j,p,q;

  i = i + j;

  p = p - q;

// további utasítások
```

Assembly nyelven

```
    movf   j,w
    addwf  i,f     ; i = i + j (LSB)
    movf   j+1,w
    addwfc i+1,f   ; i = i + j (MSB)

    movf   q,w
    subwf  p,f     ; p = p - q (LSB)
    movf   q+1,w
    subwfb p+1,f   ; p = p - q (MSB)
; további utasítások
```

A programban előbb betöltjük a WREG regiszterbe a **j** változó alacsonyabb helyiértékű felét, majd hozzáadjuk az **i** változó alacsonyabb helyiértékű feléhez. A keletkezett átvitelt figyelembe kell vennünk a változók magasabb helyiértékű felének az összeadásánál, ezért a magasabb helyiértékű bájtok összeadásához nem az **ADDWF**, hanem az **ADDWFC** utasítást használjuk. **Az addwf és az addwfc utasítások közötti adatmozgatás (movf) nem változtatja meg a Carry bitet!**

A kivonásnál is hasonlóan járunk el. Előbb az alacsonyabb helyiértékű adatszavakon végezzük el a

kivonást, majd a keletkezett áthozatalt a SUBWFB utasítással alkalmazásával vesszük figyelembe. Emlékeztetőül: a SUBWF p utasítás a p = p - W műveletnek felel meg.

32 bites összeadás és kivonás

A 16 bites összeadás vagy kivonás könnyen kiterjeszthető tetszőleges bájtszámú változókra. Az alábbi példában 32 bites változók összeadását mutatjuk be. A program megjegyzéseiben a legkisebb helyiértékű adatbájtot LSB-nek, a magasabb helyiértékűeket pedig rendre 2. bájtnak, 3. bájtnak, és MSB-nek neveztük.

C nyelven

```
Uint32 k,j;

k = k + j;
```

A legalacsonyabb helyiértékű bájtok összeadása után minden további helyiértéken az ADDWFC utasítást használjuk!

Assembly nyelven

```
k:  res  4      ; 4 bájtot, azaz
j:  res  4      ; 32 bitet foglal

  movf   j,w
  addwf  k      ; LSB-k összeadása

  movf   j+1,w
  addwfc k+1    ; 2. bájtok és C összeadása

  movf   j+2,w
  addwfc k+2    ; 3. bájtok és C összeadása

  movf   j+3,w
  addwfc k+3    ; MSB-k és C összeadása
```

Megjegyzés: Könnyen belátható, hogy a **k = k - j** 32 bites kivonás is a fentihez hasonlóan végezhető, csak az **addwf** utasítás helyére **subwf**, az **addwfc** utasítás helyére pedig **subwfb** utasítást kell írnunk.

16/32 bites inkrementálás és dekrementálás

A 16 bites változók inkrementálását vagy dekrementálását úgy végezhetjük el, hogy előbb megnöveljük (vagy csökkentjük) a változó alacsonyabb helyiértékű felét, s amennyiben keletkezett átvitel vagy áthozat, akkor megnöveljük/csökkentjük a változó magasabb helyiértékű felét is. Van azonban egy apró probléma: nincs olyan utasítás a PIC18 mikrovezérlők utasítás készletében, ami a Carry bitet figyelembe vételével végezné az inkrementálást/dekrementálást. Emiatt trükköznünk kell: nullát fogunk hozzáadni, vagy kivonni, a Carry bit figyelembevételével, ami lényegében a Carry bit hozzáadásának (illetve kivonásnál a Borrow bit kivonásának) felel meg. A másik lehetőség a trükközésre: az "inkrementálás/dekrementálás és ugrás, ha az eredmény nem nulla" típusú speciális utasítások használata.

Az alábbi táblázatban összefoglaltuk a PIC18 mikrovezérlő család inkrementálással és dekrementálással kapcsolatos utasításait:

Assembly utasítás	Az utasítás funkciója	Gépi kód	Módosul
DECF f ,d,a	cél = (f) - 1	0000 01da ffff ffff	C,DC,Z,OV,N
DECFSZ f,d,a	cél = (f) - 1; ugrás, ha az eredmény nulla	0010 11da ffff ffff	egyik sem
DCFSNZ f ,d,a	cél = (f) - 1; ugrás, ha az eredmény nem nulla	0100 11da ffff ffff	egyik sem
INCF f ,d,a	cél = (f) + 1	0010 10da ffff ffff	C,DC,Z,OV,N
INCFSZ f,d,a	cél = (f) + 1; ugrás, ha az eredmény nulla	0011 11da ffff ffff	egyik sem
INFSNZ f ,d,a	cél = (f) + 1; ugrás, ha az eredmény nem nulla	0100 10da ffff ffff	egyik sem

Az alábbi ábrán kétféle módszert is mutatunk a 16 bites változók inkrementálására és egyet a dekrementálására. Az első módszernél az infsnz és utasítást használjuk, amelyek eggyel növeli a megcímzett változó értékét, és átlépi a következő utasítást, ha az eredmény nem nulla. Így a második utasításra, a magasabb helyiértékű bájt növelésére csak akkor kerül sor, ha az alacsony helyiértékű bájt növelésekor túlcsordulás történt (nulla lett az alacsony helyiértékű bájt értéke).

INKREMENTÁLÁS

```
uint16 k;
k++;

infsnz  k
incf    k+1
```

INKREMENTÁLÁS

```
uint16 k;
k++;

movlw   0x0
incf    k
addwfc  k+1
```

DEKREMENTÁLÁS

```
uint16 k;
k--;

movlw   0x0
decf    k
subwfb  k+1
```

A második módszernél nullát töltünk a W munkaregiszterbe, s az alacsony helyiértékű bájt inkrementálása (vagy dekrementálása) után a nullát tartalmazó W regisztert és a Carry bit előző művelettel beállított értékét adjuk hozzá (vagy vonjuk ki) a magasabb helyiértékű bájthoz. Ez a módszer egy utasítással hosszabb, mint az első módszer, viszont könnyen kiterjeszthető 32 bites változókra is, ahogy az az alábbi ábrán láthatjuk:

32 BITES INKREMENTÁLÁS

```
uint32 k;
k++;

movlw  0x0
incf   k
addwfc k+1
addwfc k+2
addwfc k+3
```

32 BITES DEKREMENTÁLÁS

```
uint32 k;
k--;

movlw  0x0
decf   k
subwfb k+1
subwfb k+2
subwfb k+3
```

Az érdekesség kedvéért bemutatjuk az első változat (kicsit nyögvenyelős) 32 bites kiterjesztését is:

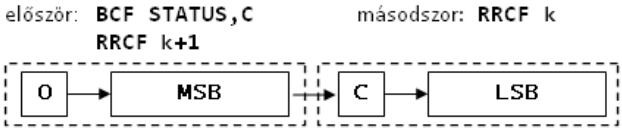
```
#include "p18f14k50.inc"
   udata_acs      ; Adatterület lefoglalása
k  res 4          ; k 32 bites változó

   CODE
;--- k++;
   incfsz  k      ; k LSB inkrementálása
   bra vege       ; vege, ha az eredmény nem nulla
   incfsz  k+1    ; k 2.bájt inkrementálása
   bra vege       ; vege, ha az eredmény nem nulla
   incfsz  k+2    ; k 3.bájt inkrementálása
   bra vege       ; vege, ha az eredmény nem nulla
   incf    k+3    ; k MSB inkrementálása
vege:
   goto $         ; végtelen ciklus
   END
```

Az elv az, hogy az adott helyiértékű bájt inkrementálása után kiugrunk, ha az inkrementált bájt nem nulla lett.

16/32 bites logikai eltolás

A több-bájtos változóknál a jobbra léptetést a magasabb helyiértékű bitekkel kell kezdeni (előtte természetesen törölnünk kell a Carry bitet). Amikor a változó alacsonyabb helyiértékű bitjeit léptetjük jobbra, akkor figyelembe kell vennünk az előző lépésben a Carry bitbe kiléptetett bitet. Az alacsonyabb helyiértékű bitek jobbra léptetéséhez tehát az **RRCF** (Rotate Right through Carry) utasítást a Carry bit törlése nélkül használjuk. A 16 bites jobbra léptetés sémája és a hozzá tartozó utasítások az alábbi ábrán láthatók.



C nyelven

```
Uint16 k;
k = k >> 1;
```

Assembly nyelven

```
bcf  STATUS,C  ;Carry törlése
rrcf k+1       ;k.MSB >> 1
rrcf k         ;k.LSB >> 1
```

A fenti eljárás könnyen kiterjeszthető 32 bites változókra is:

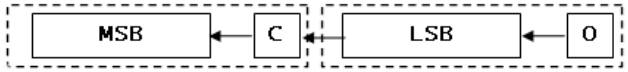
```
; Példaprogram: 32 bites logikai eltolás jobbra
;--- uint32 k
     udata_acs      ; Adatterület lefoglalása
k    res 4          ; j: 32 bites változó

     CODE
;--- k >> 1
     bcf STATUS.C   ; töröljük a Carry bitet
     rrcf k+3       ; k MSB léptetése
     rrcf k+2       ; k 3. bájt léptetése
     rrcf k+1       ; k 2. bájt léptetése
     rrcf k         ; k LSB léptetése
```

Megjegyzés: ha több helyiértékű léptetést kell végezni (mint pl. k = k >> 3), akkor azt a teljes fenti utasítássorozat ciklikus ismétlésével oldhatjuk meg.

A balra léptetést fordítva kell csinálnunk: először a változó alacsonyabb helyiértékű bitjeit léptetjük balra, majd a magasabb helyiértékű biteket forgatjuk egy **RLCF** utasítással, ami figyelembe veszi a Carry bitbe előzőleg kiléptetett bitet. A 16 bites balra léptetés sémája és a hozzá tartozó utasítások az alábbi ábrán láthatók.

másodszor: **RLCF k+1** először: **BCF STATUS,C**
RLCF k



C nyelven

```
Uint16 k;
k = k << 1;
```

Assembly nyelven

```
bcf  STATUS,C  ;Carry törlése
rlcf k         ;k.LSB << 1
rlcf k+1       ;k.LSB << 1
```

A fenti eljárás könnyen **kiterjeszthető 32 bites változókra** is:

```
; Példaprogram: 32 bites logikai eltolás balra
;--- uint32 k
     udata_acs      ; Adatterület lefoglalása
k    res 4          ; k: 32 bites változó

     CODE
;--- k << 1
     bcf STATUS.C   ; töröljük a Carry bitet
     rlcf k         ; k LSB léptetése
     rlcf k+1       ; k 2. bájt léptetése
     rlcf k+2       ; k 3. bájt léptetése
     rlcf k+3       ; k MSB léptetése
```

Kiterjesztett pontosságú feltételvizsgálatok

Nullától való különbözőség vizsgálata

Az alábbi ábrán egy 16 bites nulla/nem nulla típusú feltételvizsgálatot mutatunk be. Az assembly nyelvű kód tervezésénél kihasználhatjuk azt a tulajdonságot, hogy az **i** változó alacsonyabb és magasabb helyiértékű fele közötti bitenkénti "megengedő VAGY" (IORWF) művelet eredmény csak akkor lesz nulla, ha **i** 16 bitje közül mindegyik nulla. Ez az eljárás könnyen általánosítható 32 bites változókra is.

C nyelven

```
Uint16 i,j;

if (i) {
  j = j + i;
}

//további utasítások
```

Assembly nyelven

```
    movf   i,w       ;W = i.LSB
    iorwf  i+1,w     ;W = i.MSB | i.LSB
    bz     end_if    ;ugrik, ha Z=1 (i==0)
    movf   i,w
    addwf  j
    movf   i+1,w
    addwfc j+1
end_if:
    ; további utasítások
```

Megjegyzés: talán még emlékszünk rá, hogy a MOVf f típusú adatmozgatás beállítja az N és Z státuszbiteket. Mégsem írhatjuk a fenti programot egyszerűen így:

```
movf k,F
movf k+1,F
bz end_if   <-- Itt Z csak k.MSB állapotát tükrözi!
```

A MOV f típusú utasítás ugyanis felülírja a Z bitet, így az mindig csak az utolsó művelet eredményének megfelelő állapotot mutatja.

Egy további példa 16 bites feltételvizsgálatra

Az alábbi példában a i || !j feltétel tulajdonképpen a (i!=0) || (j==0) összetett feltételt jelenti, tehát bármelyik részfeltétel teljesül, akkor az if törzse végrehajtandó. A i és j változók nullától való különbözőségét az előző példánál már jól bevált módon, a vizsgált változó alacsonyabb és magasabb helyiértékű fele közötti IORWF művelettel vizsgáljuk.

C nyelven

```
Uint16 i,j;

if (i || !j) {
  j = j + i;
}

//további utasítások
```

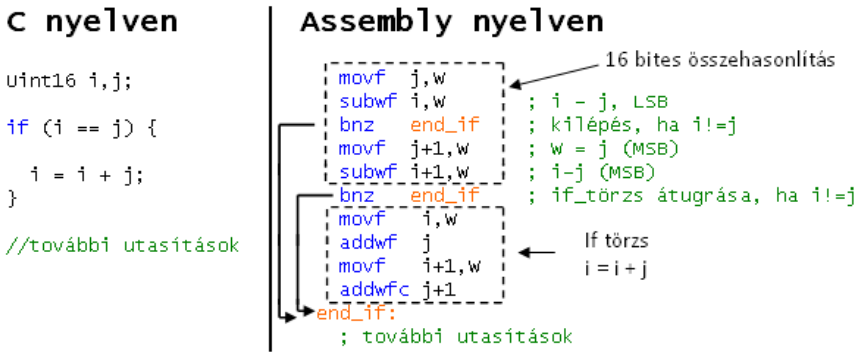
Assembly nyelven

```
    movf   i,w       ;W = i.LSB
    iorwf  i+1,w     ;W = i.MSB | i.LSB
    bnz    if_body   ;teljesült a feltétel ha i!=0
    movf   j,w0      ;W = j.LSB
    iorwf  j+1,w     ;W = j.MSB | j.LSB
    bnz    end_if    ;ugrik, ha j != 0
if_body:
    movf   i,w
    addwf  j
    movf   i+1,w     <-- If törzs
    addwfc j+1           j = j+i
end_if:
    ; további utasítások
```

Megjegyzés: Annak, hogy a nullától való különbözőséget az IORWF utasítással vizsgáljuk, semmi köze ahhoz a tényhez, hogy az if utasítás feltételében most történetesen logikai vagy kapcsolat (||) állt. Ha az if utasításban (i && !j) állna, akkor is IOR utasítással vizsgálnánk a nullától való különbözőséget.

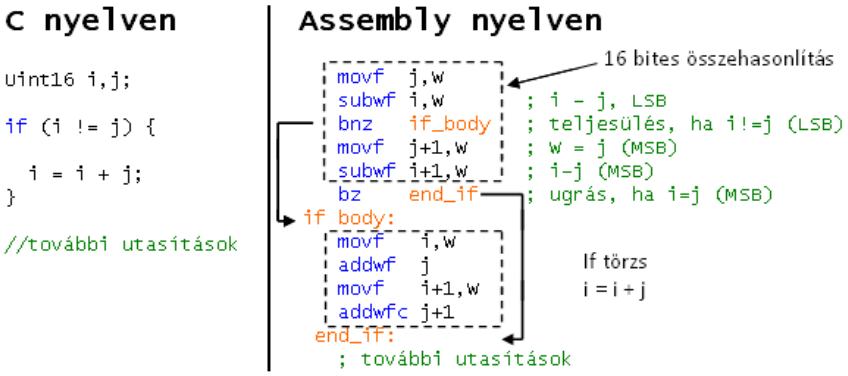
Egyenlőség vizsgálata

Az alábbi példában 16 bites változók egyenlőségét vizsgáljuk. Az if utasítás törzsét akkor kell végrehajtani, ha az i == j feltétel teljesül. Az assembly nyelvű programban az összehasonlítást bájtonként végezzük. Előbb az alacsonyabb helyiértékű bájtokat hasonlítjuk kivonással, és azonnal átlépjük az if törzsét, ha az egyenlőség ezen a bájton nem teljesül. Ugyanígy hasonlítjuk össze a magasabb helyiértékű bájtokat is. Az összehasonlítás sorrendje egyébként itt tetszőleges. A vizsgálatok végén a Z=1 állapot jelzi, ha az i == j feltétel teljesült az adott helyiértéken. Az if törzs kikerüléséhez szükséges ugrás feltétele pedig ennek ellentettje, amihez a BNZ utasítást használhatjuk.



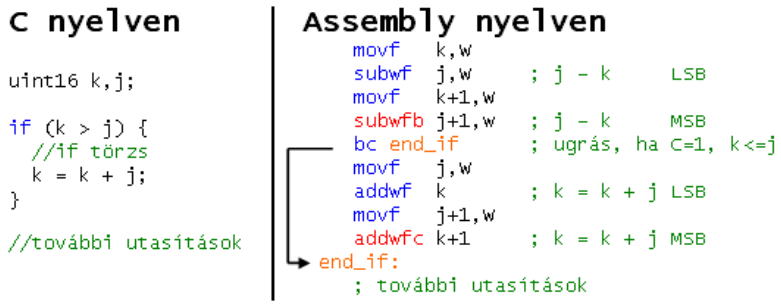
Nem egyenlőség vizsgálata

Hasonlóan végezhetjük a 16 bites változók nem egyenlőségének vizsgálatát, csupán az ugrások feltétele lesz más. Az alábbi példában az if utasítás törzsét akkor kell végrehajtani, ha az i != j feltétel teljesül. Az assembly nyelvű programban az összehasonlítást bájtonként végezzük. Előbb az alacsonyabb helyiértékű bájtokat hasonlítjuk össze (kivonással és a Z jelzőbit vizsgálatával) és azonnal az if törzsre ugorhatunk, ha az egyenlőség ezen a bájton nem teljesül. Csak az alacsony helyiértékű bájtok egyenlősége esetén szükséges összehasonlítanunk a magasabb helyiértékű bájtokat is. A második vizsgálat végén akkor kell kikerülnünk az if törzset, ha Z=1 állapotot találunk, ez ugyanis i és j egyenlőségét jelzi, ami az if feltételének az ellentettje.



A "nagyobb, mint..." feltétel vizsgálata

Az alábbi példában a k > j feltételt vizsgáljuk. Az egyetlen különbség az előző fejezetben bemutatott 8 bites összehasonlításhoz képest az, hogy a kivonást és az összeadást 16 bites változókon végezzük. A **k > j** feltételt most is a **j - k** kivonást használva vizsgáljuk! Az **if** törzs kikerüléséhez szükséges feltétel **k<=j** lesz, melynek feltétele **C=1**. Így csupán egyetlen **bc** feltételvizsgáló utasításra van szükség.



Megjegyzés: A magasabb helyiértékű bájtok kivonásánál/összeadásánál ne felejtsük el figyelembe venni az áthozatot/átvitelt! A fenti ábrán emlékeztetésül pirossal kiemeltük az áthozatot/átvitelt kezelő utasításokat.

A fenti összehasonlítást könnyen kiterjeszthetjük 32 bites változókra is:

```
#include "p18f14k50.inc"
; FIGYELEM! A projekthez csatolni kell a 18f14k50.lkr állományt is!
   udata_acs                    ; C nyelven:
k  res 4                        ; uint32 k;
j  res 4                        ; uint32 j;

   CODE
```

```
;--- if(k > j)
   movf   k,W
   subwf  j,W     ; j - k     LSB
   movf   k+1,W
   subwfb j+1,w   ; j - k     2. bájt
   movf   k+2,W
   subwfb j+2,w   ; j - k     3. bájt
   movf   k+3,W
   subwfb j+3,w   ; j - k     MSB
   bc end_if      ; ugrás, ha C=1, k<=j
;--- if törzs: k = k + j
   movf   j,W
   addwf  k       ; k = k + j LSB
   movf   j+1,W
   addwfc k+1     ; k = k + j 2. bájt
   movf   j+2,W
   addwfc k+2     ; k = k + j 3. bájt
   movf   j+3,W
   addwfc k+3     ; k = k + j MSB
end_if:
;--- további utasítások
   END
```

Előjeles egész számok ábrázolása

Minden eddigi példában előjel nélküli számokkal, változókkal volt dolgunk. Természetesen, szeretnénk majd előjeles mennyiségekkel is dolgozni, műveletet végezni olyan számokkal, mint például -100 vagy +27. De ahhoz, hogy ezekre sor kerülhessen, előbb találnunk kell egy olyan bináris számábrázolási módszert, amellyel az előjeles számok is kezelhetők. Az előjeles számok ábrázolására több lehetőség is kínálkozik, melyek közül a három legelterjedtebbet tekintjük át röviden. Ezeknél a számábrázolási módok két tulajdonságukban megegyeznek: az egyik tulajdonságuk az, hogy a pozitív számokat ugyanúgy ábrázolják, mint az előjel nélküli számábrázolásnál, a másik pedig az, hogy negatív számok esetében a legmagasabb helyiértékű bit "1" állapotú.

Előjel-abszolútértékes ábrázolás

Az előjel-abszolútértékes (signed magnitude) ábrázolás onnan kapta a nevét, hogy az előjeles számokat egy előjelbittel és azt azt követő abszolútértékkel ábrázoljuk. A legmagasabb helyiértékű bitet használjuk az előjel tárolására (pozitív számoknál az előjelbit nulla, a negatív számoknál pedig 1), a maradék bitekbe pedig a szám abszolútértékét írjuk. A negatív számok ábrázolásának ez a legegyszerűbb módja, hiszen egyszerűen csak egy 1-est írunk az előjel helyére.

Például: 8 biten ábrázolva, -12 = 0x8C = 0b1000 1100

Ennek a számábrázolásnak az az előnye, hogy az előjel és az abszolútérték közvetlenül rendelkezésre áll (pl. a szám negálásához csak az előjel bitet kell invertálni). Hátránya azonban, hogy kétféle ábrázolás is van a nullának, s nem használható ugyanaz a hardver az előjeles számokkal végzett műveletekhez, mint amelyik az előjel nélküli számokhoz való. Ezt a számábrázolási módot elsősorban BCD aritmetikához, illetve a lebegőpontos számábrázolásban használják.

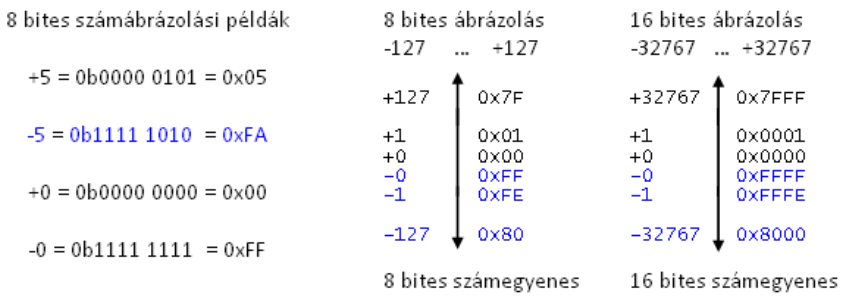
Inverz kódú ábrázolás

Ez az ábrázolásmód is nagyon egyszerű. Az előjel helyére 1-et írunk a többi bitet pedig invertáljuk.

Egyes komplemens: - N = ~(+N)

Az előző példára alkalmazva: -12 = 0xF3 = 0b1111 0011

Az inverz kódú (vagy más néven egyes komplemens) ábrázolást egyes lyukszalag vezérlésű szerszámgépekben alkalmazzák, vagy némelyik hardver gyorsítós grafikus vezérlőben. Az előjel nélküli egész számokhoz való bináris összeadó áramköröket és logikát használhatjuk az inverz kódú ábrázolással kezelt előjeles számokra is, amennyiben tolerálható, hogy egyes esetekben az eredmény eggyel eltér a helyes értéktől, ami akkor fordul elő, ha két negatív, vagy egy pozitív és egy negatív számot adunk össze.



Ha megnézzük például a 8 bites számegyenes mentén elhelyezkedő számokat, akkor rájövünk, hogy miből adódik az eltérés azoknál az összeadásoknál,amelyek átlépik a nullát. Induljunk ki -1-ből, melyhez a 0xFE hexadecimális érték tartozik, s adjunk hozzá egyet! Az eredmény 0xFF, ami -0-nak felel meg. Adjunk hozzá még egyet! Ekkor 0xFF-ből 0x00 lesz, ami +0-nak felel meg, ami eggyel kevesebb a várt eredménynél. Ha mégegyet hozzáadunk, akkor 0x01-et,azaz +1-et kapunk eredményül, ami megint eggyel kevesebb a vártnál.

Tehát azt kaptuk, hogy: -1 + 1 = -0; -1 + 2 = +0; -1 + 3 = +1

Mindez azért van, mert a számegyenes kétfelé szétcsúszott, -0 és +0 között van egy egységnyi hézag! Ha a

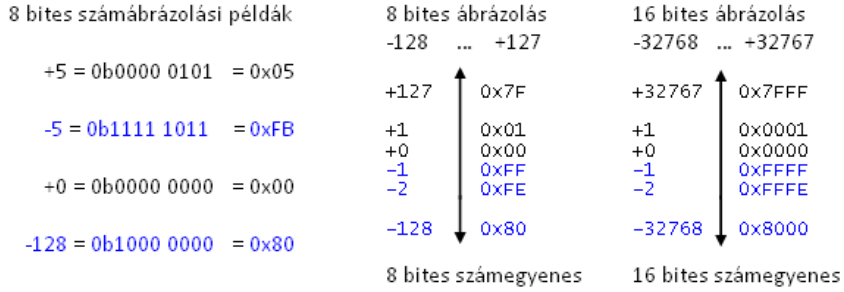
negatív számok félegyenesét egy egységgel feljebb tolnánk, vagyis a negatív számokat úgy képeznénk, hogy az egyes komplemens képzése (bitenkénti invertálás) után a számhoz egyet még hozzáadnánk, akkor kiküszöbölhetnénk a nullát átlépő összeadások problémáját. Most jön a meglepetés, pont ez lesz a beígért harmadik módszer, amelyet a mikroprocesszorok és mikrovezérlők elterjedten használnak!

Kettes komplemens kódú ábrázolás

Mind az előjel-abszolútértékes mind az inverz kódú ábrázolás nagy hibája, hogy problematikus az összeadás és a kivonás műveletének elvégzése illetve az eredmény értelmezése. További gondot jelent, hogy mindkét ábrázolásmódban pozitív és negatív számként is ábrázolhatjuk a zérust. Ez sajnos azt jelenti, hogy nincs kölcsönösen egyértelmű megfeleltetés a szimbólumsorozatok és az általuk ábrázolt szám között.

Ezeket a problémákat küszöböli ki a kettes komplemens kódú ábrázolás. Egy bináris szám kettes komplemensét úgy képezhetjük, hogy a szám inverzéhez hozzáadunk 1-et.

Kettes komplemens: **-N = ~(+N) + 1**
A korábbi példára alkalmazva: -12 = 0xF4 = 0b1111 0100



Előjeles decimális szám kettes komplemense

Ha van egy előjeles decimális (tízes számrendszerben felírt) N számunk, akkor hogyan képezzük annak a kettes komplemensét? Például a **-N = ~(+N) + 1** képlet használatával az alábbi lépeseket hajtsuk végre:

Ha N pozitív	konvertáljuk N-et hexadecimálisra!	**+60 = 0x3C**
Ha N negatív	1. az előjelet elhagyva konvertáljuk N-et hexadecimálisra! 2. A hexadecimális számot írjuk át binárisra! 3. A bináris számot bitenként invertáljuk! 4. Az invertált számot írjuk vissza hexadecimális alakra, és adjunk hozzá egyet! Az így kapott szám N kettes komplemense, ez lesz a -N szám ábrázolása.	60 = 0x3C 0011 1100 1100 0011 0xC3 +0x01 -60 = 0xC4

Hexadecimális számok konvertálása előjeles decimális számmá

Ha adott egy hexadecimális szám, a gyanútlan olvasóban felmerülhet a kérdés: honnan tudjuk, hogy az adott szám egyes vagy kettes komplemens ábrázolású előjeles szám, vagy pedig előjel nélküli szám? A válasz az, hogy magából a hexadecimális számból ez nem derül ki, hiszen pl. a legnagyobb helyiértékű bitre sincs ráírva, hogy ez most helyiérték vagy előjelbit. Ahhoz tehát, hogy egy adott számot helyesen értelmezzünk, többlet-tudásra, kiegészítő információra van szükségünk. Tudnunk kell, hogy milyen módszerrel kódolt, milyen konvenciók alapján kezelt adatról van szó, hogy mi is aszerint értelmezzük azt.

Nézzünk erre egy egyszerű példát! Ha egy regiszterből például 0xFE értéket olvasunk ki, akkor azt hogy értelmezzük?

0xFE ha előjel nélküli 8 bites egésznek vesszük = 254
0xFE ha kettes komplemens ábrázolású 8 bites előjeles egész = -2

Ha tudjuk, hogy az adott hexadecimális szám kettes komplemens ábrázolású előjeles egész, akkor az alábbi lépésekben konvertálhatjuk tízes számrendszerbe:

Első módszer:

1. Megállapítjuk a szám előjelét. Ha a legmagasabb helyiértékű bit = 1 (a legmagasabb helyiértékű hexadecimális számjegy > 7), akkor negatív számról van szó *(s folytassuk a 2. pontnál!)*, ellenkező esetben pedig pozitív számról *(akkor folytassuk az 5. pontnál!)*.
2. A hexadecimális számot írjuk át binárisra!
3. A bináris számot bitenként invertáljuk!
4. Az invertált számot írjuk vissza hexadecimális alakra, és adjunk hozzá egyet!
5. A kapott számot konvertáljuk 10-es számrendszerbe, és hozzáírjuk az előjelet!

Példa: értelmezzük az 0xF0 számot 8 bites, kettes komplemens ábrázolású számként!

1. Az első számjegy vizsgálata	F > 7, 0xF0 negatív szám
2. Binárissá alakítjuk	0xF0 = 0b1111 0000
3. Bitenként invertáljuk	0x0F = 0b0000 1111

4. Visszaírjuk hexadecimális alakba, és megnöveljük	0x0F + 1 = 0x10
5. Tízes számrendszerbe konvertáljuk, s elé írjuk az előjelet	0x10 = 16 → **-16**

Tehát a 0xF0 szám -16-ot jelent!

Második módszer:

Talán soknak tűnik ez a sok számrendszer váltás, de a bitenkénti invertálás bináris számrendszerben a legegyszerűbb, s a binárisból hexába vagy a hexából binárisba váltás roppant egyszerű, hiszen a hexadecimális rendszer számjegyei egy-egy négyjegyű számcsoportot alkotnak binárisba átírva. De ha a kivonás kényelmesebb, akkor a fenti leírás 2., 3. és 4. lépése helyett az átalakítandó N számot egyszerűen vonjuk ki 0-ból (nem törődve az átvitellel), kihasználva a 0 - (-N) = N azonosságot.

A fenti példa szerinti 0xF0-át értelmezve:

az előjel = - (mert F>7),
az abszolútérték = 0 - 0xF0 = 0x10 = 16

Tehát a szám **-16**!

Túlcsordulás a kettes komplemens ábrázolású aritmetikában

Tekintsük például a 8 bites kettes komplemens ábrázolást, melynek felhasználásával -128 és +127 közötti számokat tudunk ábrázolni. Ha a (+1) + (+127) = (+128) összeadással próbálkozunk, akkor mi történik, hiszen a +128 kívül esik az ábrázolható számtartományon.

```
 +127  =  0x7F
+  +1  =  0x01
-------------
  128  != 0x80 (ez valójában -128-nak felel meg a kettes komplemensű számábrázolásban)
```

Miből derül ki, hogy a fenti 8 bites összeadásban túlcsordulás történt? Abból, hogy két pozitív számot összeadtunk, és negatív lett az eredmény.

A fenti esethez hasonlóan túlcsordulás történhet akkor is, ha két negatív számot adunk össze, melyek összege -128-nál kisebb számot eredményezne (pl. (-128) + (-1) = -129). Ilyen esetben - a túlcsordulás miatt - az eredmény pozitív lesz.

Az előjel nélküli számok összeadásánál a Carry bit jelezte a túlcsordulást. Megmutatjuk, hogy a kettes komplemens ábrázolású számok összeadásánál a Carry bit hasznavehetetlen, nem alkalmas a túlcsordulás jelzésére. Vegyük például a (+1) + (-1) = 0 összeadást, ami a 8 bites kettes komplemens ábrázolásban 0x01 + 0xFF alakba írható. Ez az összeg 0x00-t ad, tehát az eredmény helyes, ugyanakkor a legmagasabb helyiértéken keletkezett átvitel, tehát C=1. Tehát a Carry bit nem azt mutatja, hogy helyes-e az összeadásunk eredménye.

Az előjeles számokkal végzett műveletek esetén az STATUS regiszter OV státuszbitje jelzi, hogy történt-e túlcsordulás, ezt kell használnunk. Az OV státuszbitet a hardver a

$$V = C_{MSB} \wedge C_{MSB-1}$$

képlet alapján állítja elő, ahol C_{MSB} a legmagasabb helyiértéken keletkezett átvitel (vagyis ugyanaz, mint a Carry bit), C_{MSB-1} az eggyel kisebb helyiértékű biten keletkezett átvitel, a '^' jel pedig a kizáró vagy (XOR) logikai művelet jele.

Az alábbi egyszerű példákban bemutatjuk mind a négy lehetséges esetet a C és OV státuszbitek beállására.

Összeadás	Előjel nélkül	Előjelesen	Összeadás	Előjel nélkül	Előjelesen
0x01 +0xFF 0x00	1 +255 0	1 + -1 0	0x80 +0xFF 0x7F	128 +255 127	-128 + -1 +127
C=1, Z=1, OV=0, N=0			C=1, Z=0, OV=1, N=0		

Összeadás	Előjel nélkül	Előjelesen	Összeadás	Előjel nélkül	Előjelesen
0x01 +0x7F 0x80	1 +127 128	1 +127 -128	0x80 +0x20 0xA0	128 + 32 160	-128 + +32 -96
C=0, Z=0, OV=1, N=1			C=0, Z=0, OV=0, N=1		

Figyeljük meg, hogy a Carry bit akkor nulla, amikor az előjel nélküli számokkal végzett összeadás eredménye helyes. Az OV (Overflow) bit pedig akkor nulla, amikor az előjeles (kettes komplemens ábrázolású) számok összeadás helyes.

Műveletek előjeles változókkal

A kettes komplemens ábrázolás egyik fő előnye, hogy az összeadás és kivonás az előjeles számokkal is ugyanúgy végezhető, mint az előjel nélküli számokkal (csak a túlcsordulást jelző bit lett más). Vannak azonban

olyan műveletek, amelyeket az előjeles változókon más utasítással kell végrehajtani. Ezek áttekintését az alábbi összefoglaló táblázat segíti.

Műveletek előjeles és előjel nélküli számokhoz	Ugyanaz?
&, |, ^, ~ (bitenkénti logikai műveletek)	igen
+, –	igen
= =, !=	igen
<< (balra léptetés)	igen
>, >=, <, <= (összehasonlítás)	**nem**
>> (jobbra léptetés)	**nem**
*, /	**nem**

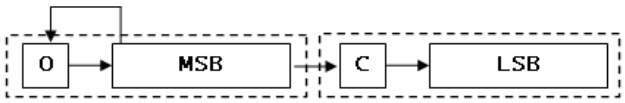
Mint láthatjuk, a műveletek közül az összehasonlítás, a jobbra léptetés és a szorzás/osztás igényel "különleges elbánást" az előjeles számok esetében. Ezek közül ebben a fejezetben a jobbra léptetéssel és az előjeles feltételvizsgálatokkal foglalkozunk, a szorzást és az osztást majd egy későbbi fejezetben tárgyaljuk.

Aritmetikai eltolás jobbra

Az előzőekben már foglalkoztunk a **logikai jobbra léptetéssel**, ami az előjel nélküli változóknál a kettővel történő osztásnak felel meg. Az előjeles változóknál azonban nem erre, hanem az aritmetikai léptetésre van szükségünk, amelynél a legmagasabb helyiértékű bit visszamásolódik a megürülő helyre (ezáltal megkettőződik). Ez biztosítja, hogy a kettes komplemens ábrázolásnál a jobbra léptetés megőrizze a szám előjelét.

Példa: 0x80 = -128 aritmetikai jobbra léptetése
0x80 = 0b1000 0000 → 0b1100 0000 = 0xC0 = -64

Mivel a PIC18 mikrovezérlők nem rendelkeznek aritmetikai léptetést végző utasítással, így a jobbra léptetés előtt előkészítő utasításokat kell beiktatni a legmagasabb helyiértékű bitnek a Carry bitbe történő másolására. Az alábbi programrészlet a 16 bites **k** változó aritmetikai jobbra léptetését végzi el.



C nyelven

```
signed int k;

k = k >> 1;
```

Assembly nyelven

```
bcf   STATUS,C  ;Carry bit törlése
btfsc k+1,7     ;előjel bit = 0?
bsf   STATUS,C  ;C<-1, ha előjel=1
rrcf  k+1       ;k.MSB >> 1
rrcf  k         ;k.LSB >> 1
```

Először nullázzuk a **STATUS** regiszter **C** bitjét (Carry bit), majd megvizsgáljuk a **k** változó legmagasabb helyiértékű bitjét egy **btfsc** (bit test, skip if clear) utasítással (a magasabb helyiértékű bájt címe **k+1**, a legmagasabb helyiértékű bitjének sorszáma pedig 7). Ez az utasítás átlépi a következő utasítást, ha a vizsgált bit értéke nulla. Tehát ha a vizsgált bit értéke nulla, akkor a Carry bit értéke is nulla, ha pedig a vizsgált bit értéke 1, akkor a program ráfut a bsf STATUS,C utasításra, ami '1'-be állítja a Carry bitet. Végeredményben tehát, ha körülményesen is, de átmásoltuk a **k** változó előjel bitjét a Carry állapotjelző bitbe.

A továbbiak úgy zajlanak, mint a logikai léptetésnél: előbb a magas helyiértékű bájtot léptetjük jobbra, majd az alacsonyabb helyiértékűt. A bitforgatás a Carry biten keresztül történik, ez gondoskodik róla, hogy a magasabb helyiértékű bájtból kicsorduló bit belépjen az alacsonyabb helyiértékű bájt legfelső bitjébe: az **rrcf k+1** utasítás az MSB 0. bitjét belépteti a Carry bitbe, az **rrcf k** utasítás pedig C értékét belépteti az LSB 7. bitjébe.

Az eljárás könnyen általánosítható 32 bites változókra is:

32 bites aritmetikai eltolás jobbra

```
bcf   STATUS,C  ;Carry bit törlése
btfsc k+3,7     ;előjel bit = 0?
bsf   STATUS,C  ;C<-1, ha előjel=1
rrcf  k+3       ;k >> 1 (MSB)
rrcf  k+2       ;k >> 1 (3. bájt)
rrcf  k+1       ;k >> 1 (2. bájt)
rrcf  k         ;k >> 1 (LSB)
```

Megjegyzés: Az MPLAB C18 fordítója előjeles változók esetén is logikai jobbra léptetésként értelmezi a >> operátort, ezért vigyázzunk, előjeles változóknál ne használjuk a jobbra léptetést!

Előjeles feltételvizsgálat

A PIC18 mikrovezérlők két olyan állapotjelző bittel rendelkeznek, amelyek hasznosak az előjeles számokon végezett feltételvizsgálatoknál:

OV a kettes komplemens ábrázolású számokkal végzett műveleteknél a túlcsordulást jelzi
N a negatív eredményt jelzi (N = 1 lesz, ha az MSB = 1).

Fentieken kívül természetesen használható a **Z** státuszbit is, ami ugyanúgy, mint az előjel nélküli számokkal

végzett műveleteknél, itt is azt jelzi, hogy nulla volt-e az eredmény.

Sajnos, a PIC18 mikrovezérlők nem rendelkeznek előjeles feltételvizsgáló és elágaztató utasításokkal, amelyek egyetlen lépésben elvégeznék a státuszbitek kiértékelését, így azt csak több lépésben, bonyolult elágazásokkal tudjuk a programmal elvégeztetni. A feltételes elágaztató utasítások előtt a szokásos módon egy kivonást (**SUBWF**/**SUBWFB**) kell végeznünk.

Az i > j feltétel előjeles vizsgálata

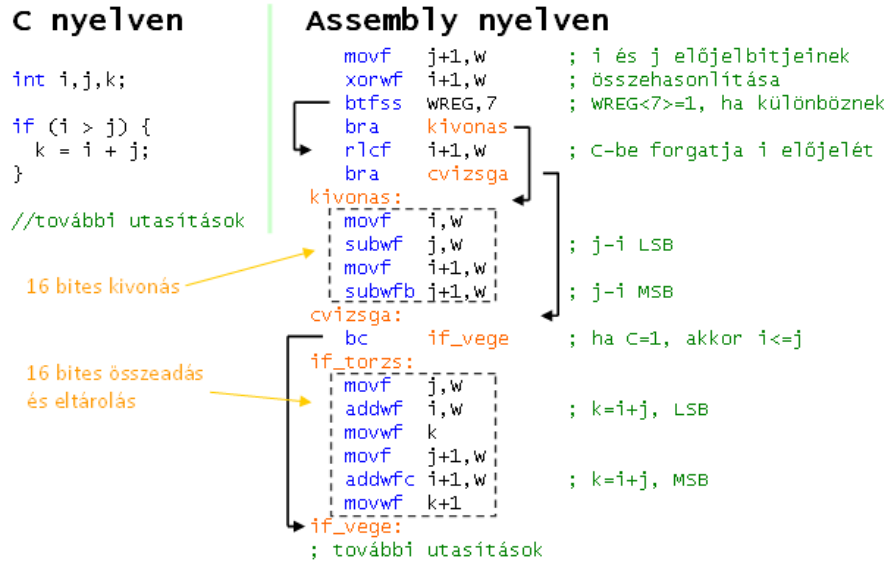
Az alábbi táblázatban az **i>j** feltétel vizsgálatához elvégzett **j - i** kivonás különböző eseteit mutatjuk be. Figyeljük meg az **N**, **C** és **OV** státuszbitek beállását!

Kivonás	Előjel nélkül	Előjelesen	Kivonás	Előjel nélkül	Előjelesen
0x01 - 0x7F 0x82	1 - 127 130	1 - 127 -126	0xFF -0x80 0x7F	255 - 128 127	-1 - -128 +127
N=1, OV=0, C=0 **i > j teljesül**			N=0, OV=0, C=1 **i>j nem teljesül**		

Kivonás	Előjel nélkül	Előjelesen	Kivonás	Előjel nélkül	Előjelesen
0x7F - 0x80 0xFF	127 - 128 255	127 - -128 -1	0xFF -0x01 0xFE	255 - 1 254	-1 - 1 -2
N=1, OV=1, C=0 **i > j nem teljesül**			N=1, OV=0, C=1 **i > j teljesül**		

Figyeljük meg, hogy **ha a kisebbítendő és a kivonandó előjele megegyezik** (felső sor), akkor az előjel nélküli változóknál használt, a Carry bit vizsgálatán alapuló összehasonlítás is helyes eredményre vezet (ha C=0, akkor i>j igaz). **Ha a kisebbítendő és a kivonandó előjele különbözik,** akkor elegendő megnézni, hogy i és j közül melyik a pozitív. (Ugyanis bármelyik nemnegatív szám nagyobb, mint bármelyik negatív szám...)

Az alábbi ábrán megmutatjuk, hogy az MPLAB C18 fordítója hogyan alkalmazza a fenti algoritmust (az MPLAB fejlesztői környezetben a Disassembly Listing nevű ablakban tekinthetjük meg a C fordító által generált assembly kódot).



Mivel a példában **i** és **j** 16 bites előjeles változók, így az **i+1** és **j+1** címen elhelyezkedő bájtok 7. bitjét kell összehasonlítani, az az előjelbit. Ezt úgy tehetjük meg egyszerűen, hogy a **W** munkaregiszterben előállítjuk (i+1) és (j+1) bitenkénti kizáró vagy (XOR) kapcsolatát. Ha **WREG**<7> = 1 lesz, akkor a kér változó előjele különböző. Ha pedig **WREG** 7. bitje nulla, akkor azonos előjelűek.

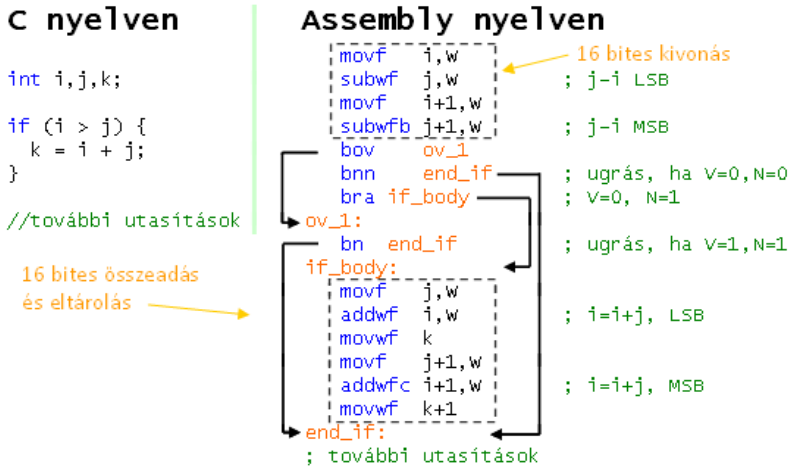
Ha **WREG** 7. bitje '1' volt, akkor **i** és **j** ellentétes előjelűek. Ekkor csak azt kell megnézni, hogy **i** előjele nem negatív-e. Ha nem negatív akkor az i>j feltétel teljesül, mert a különbözőségből következik, hogy akkor **j** a negatív szám. Ehhez a vizsgálathoz az előjel bitet a Carry bitbe forgatjuk az **rlcf i+1,W** utasítással, majd elugrunk a **C** tartalmát vizsgáló programrészhez. (**Megjegyzés:** a **W** megadásának itt az a célja, hogy az eredmény a munkaregiszterbe kerüljön, az **i** változó eredeti tartalmát ne írjük át.)

A másik ágon haladva (amikor **i** és **j** előjele megegyezik, és a **btfss WREG,7** utasítás nem ugrik) el kell végeznünk a **j - i** 16 bites kivonást, s az előjel nélküli összehasonlításnál tanult módon C=1 jelzi az if törzs kikerülésének feltételét (és megfordítva: **C=0** jelenti az **i>j** feltétel teljesülését).

Másik módszer az i > j feltétel előjeles vizsgálata

A másik vizsgálati módszer az **OV** és az **N** bitek vizsgálatán alapul. Az **OV** bit 0 értéke azt jelzi, hogy az előjel figyelembevételével elvégzett művelet eredménye helyes (nem történt túlcsordulás). Mivel **i>j** esetén az

előjelhelyesen elvégzett **j-i** kivonása negatív kell, hogy legyen, így az **i>j** feltétel teljesülését **OV=0** esetén **N=1** jelzi (N=1 jelzi, ha negatív az eredmény). Túlcsordulás (OV=1) esetén azonban megfordul a helyzet: **N=1** estén i<=j és **N=0** jelzi, hogy **i>j** teljesül. Az alábbi egyszerű példában ezt az utóbbi módszert használjuk a 16 bites előjeles változók összehasonlítására, az **i > j** feltétel teljesülését vizsgálva.



A programrészlet egy 16 bites kivonással kezdődik, melyet a (~OV & N) | (OV & ~N) feltételvizsgálat követ. A sok elágaztatás a PIC18 mikrovezérlő esetében elkerülhetetlen. A fejlettebb mikrovezérlők (pl. a PIC24 is) és mikroprocesszorok azonban rendelkeznek ún. előjeles elágaztató utasításokkal, amelyek közvetlenül vizsgálják az (OV^N) feltételt (ahol a ^ műveleti jel a kizáró vagy kapcsolatot jelenti).

Az előjeles feltételvizsgálatok összefoglalása

A feltétel	Művelet	A teljesülés feltétele
if (i > j) {}	j - i	OV=0, N=1 vagy OV=1, N=0
if (i <= j) {}	i - j	OV=0, N=0 vagy OV=1 , N=1

Előjel kiterjesztése

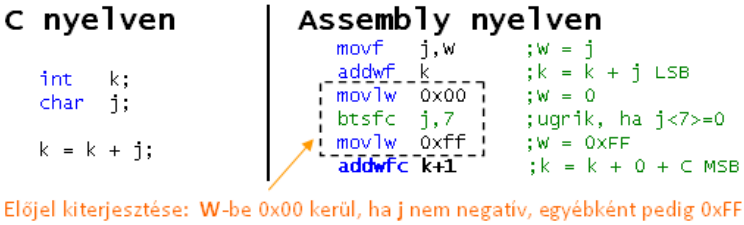
Még az olyan műveleteknél is, mint az összeadás és a kivonás, amelyek egyformán használhatók előjeles és előjel nélküli számokra, óvatosságra és körültekintésre van szükség, ha különböző bitszámú operandusokkal dolgozunk. A kisebb bitszámú változót ki kellett egészíteni a másik változóval egyező méretre, mielőtt az összeadást vagy kivonást elvégeztük volna. Az előjel nélküli változóknál egyszerűen nullával kell feltölteni a hiányzó magasabb helyiértékű biteket. Az előjeles változóknál mása helyzet, ezeknél a kisebb bitszámú változó előjelbitjével kell feltölteni a hiányzó magas helyiértékű biteket.

Az alábbi két ábrán 8 és 16 bites számok előjel nélküli és előjeles összeadására mutatunk egy-egy példát. Előjel nélküli esetben az alacsony helyiértékű bájtok összeadása után nullát töltünk a munkaregiszterbe, s a kétbájtos változó magasabb helyiértékű feléhez ezt a nullát adjuk hozzá, a Carry bit tartalmával együtt (k.MSB = k.MSB + 0 + C, ahol C az alacsony helyiértékű bájtok összeadásánál keletkezett átvitel).

Az előjeles változóknál csak annyi a különbség, hogy az egybájtos operandus 7. bitjétől függően 0xFF vagy 0x00 tartalommal töltjük fel a W munkaregisztert, s ezt adjuk a második operandus magasabb helyiértékű bájtjához. Ez a előjeles 8 bites szám előjelhelyes kiterjesztésének felel meg (8-bit → 16-bit típuskonverzió).

a.) Vegyes művelet, 8/16 bites előjel nélküli változókkal

b.) Vegyes művelet, 8/16 bites előjeles változókkal



A 16-bit → 32-bit típuskonverzió sem bonyolultabb, arra is mutatunk egy példát:

c.) Vegyes művelet, 16/32 bites előjeles változókkal

C nyelven

```
int32  k;
int16  j;

k = k + j;
```

Assembly nyelven

```
movf   j,w
addwf  k          ;k = k+j  LSB
movf   j+1,w
addwfc k+1        ;k = k+j  2.bájt
movlw  0x00       ;w = 0
btsfc  j+1,7      ;ugrik, ha j >= 0
movlw  0xff       ;w = 0xFF
addwfc k+2        ;k = k + 0 + C 3. bájt
addwfc k+3        ;k = k + 0 + C MSB
```

Előjel kiterjesztése: **W**-be 0x00 kerül, ha **j** nem negatív, egyébként pedig 0xFF

Megjegyzés: Ügyeljünk rá, hogy a több-bájtos értékek összeadásánál/kivonásánál a magasabb helyiértékű bájtoknál a Carry bitet is figyelembevevő addwfc/subwfb utasítást használjuk!