

Ismerkedés a PIC18 mikrovezérlőkkel

Főmenü

Nyitólap

Bevezetés a PIC18 assembly programozásába

8 bites előjel nélküli műveletek

Kiterjesztett pontosságú és előjeles műveletek

Mutatók, tömbök, szubrutinok

Assembly programozás haladóknak

A kísérleti áramkör

Az USB használata

I/O portok

Programmegszakítások

Számlálók/időzítők

Analóg perifériák

LCD kijelzők vezérlése

Aszinkron soros I/O

I2C soros I/O

SPI soros I/O

PWM

Szoftver segédlet

PIC18 példaprogramok

Programjainkat az alábbi fejlesztői áramkörök valamelyikén is kipróbálhatjuk

Fejlesztői áramkörök

USB-UART átalakító

Low Pin Count USB Development Kit

PIC18F4550 Proto Board

PICDEM Full Speed USB

8 bites előjel nélküli műveletek

A fejezet tartalma:

- Bitenkénti logikai műveletek
- Bitműveletek
- A STATUS regiszter
- Feltételes programvégrehajtás bitvizsgálattal
- Feltételvizsgálatok a C programnyelvben
- Nulla, nem nulla típusú feltételvizsgálatok
- Egyenlőség és nem egyenlőség vizsgálata
- A switch utasítás megvalósítása assembly nyelven
- Programciklusok
- Léptetés és bitforgatás
- Egyszerű példák a léptetésre
- Aritmetikai kifejezések kiértékelése

Eben a fejezetben a PIC18 mikrovezérlők utasításainak újabb csoportjaival ismerkedünk meg, mégpedig a 8 bites bitenkénti logikai műveletekkel, a bitkezelő (bit vizsgálata, törlése, beállítása) utasításokkal, az összehasonlítás és a vizsgálat eredményétől függő ugró utasításokkal, a programciklust szervező utasításokkal, valamint a bitforgatás és léptetés műveletekkel.

Az alábbi táblázatban felsoroltuk azoknak az aritmetikai és logikai utasításoknak a C nyelvi megfelelőjét, melyekkel ebben a jegyzetben foglalkozunk. Ezeket a műveleteket az adatútra kapcsolódó aritmetikai és logikai egység (ALU) hajtja végre.

Aritmetikai és logikai utasítások

Művelet	Leírás
+, -	(+) összeadás, (-) kivonás
++, --	(++) inkrementálás, (--) dekrementálás
, /	() szorzás, (/) osztás
>>, <<	jobbra léptetés (>>), balra léptetés (<<)
&, , ^	bitenkénti AND (&), IOR (), XOR (^)
~	bitenkénti komplementálás

A fenti C logikai műveletek PIC18 assembly megfelelőit keressük. Az előző fejezetben már megismertedtünk néhány fontos aritmetikai művelettel (összeadás, kivonás, inkrementálás, dekrementálás), s most a bitenkénti logikai ÉS (&), VAGY (|), kizáró vagy (^) és a komplementálás (~) műveletekkel fogunk megismerni, ezek alkotják az ALU logikai utasításkészletét. A szorzással és az osztással majd később foglalkozunk.

Bitenkénti logikai műveletek

A bitenkénti jelző itt azt jelenti, hogy az AND, IOR, XOR műveleteket bitenként, az operandusok azonos helyiértékű bitjei között végezzük. Ezek a műveletek rendkívül hasznosak, ha egy-egy bitszoportot törölni, logikai 1-be állítani vagy ellenkezőjére változtatni akarunk.

Bitenkénti "ÉS" (AND, &) művelet		
Assembly utasítás	Jelentése	C megfelelője
ANDLW k	k & (W) → W	j = j & lit8;
ANDWF f,d,a	(f) & (WREG) →cél	j = j & i;

Mint látható, az aritmetikai utasításokhoz hasonlóan a bitenként műveleteket is végezhetjük egy számkonstans és a **W** munkaregiszter tartalmát felhasználva, vagy egy memória regiszter és a **W** munkaregiszter között. A számkonstans (k, vagy lit8) 0..255 közötti szám lehet.

Az egybites **d** kapcsoló a célhelyet jelöli ki: d=0 esetén az adat a **W** munkaregiszterbe kerül, d=1 esetén pedig az **f** memóriacímre íródik vissza. Az **a** címmódosító azt mondja meg, hogy az **f** cím az Access Bank virtuális memórialapon vagy a **BSR** regiszter által kijelölt memórialapon értendő: a=0 esetén nem vesszük figyelembe **BSR** tartalmát, a=1 esetén pedig a **BSR** által kijelölt memórialapot használjuk.

Bitenkénti MEGENGEDŐ VAGY (IOR,) művelet		
Assembly utasítás	Jelentése	C megfelelője
IORLW k	k (W) → W	j = j lit8;
IORWF f,d,a	(f) (WREG) →cél	j = j i;

Az **AND** műveletnél elmondottak érvényesek a megengedő (**IOR**) és a kizáró vagy (**XOR**) műveletek esetére is.

Bitenkénti KIZÁRÓ VAGY (XOR, ^) művelet		
Assembly utasítás	Jelentése	C megfelelője
XORLW k	$k \wedge (W) \rightarrow W$	$j = j \wedge \text{lit8};$
XORWF f,d,a	$(f) \wedge (WREG) \rightarrow \text{cél}$	$j = j \wedge i;$

A komplementálás (vagy komplement képzés) egyoperandusos művelet, s komplement azt jelenti hogy, hogy az operandus bitjeit bitenként az ellenkezőjére változtatjuk, az $\sim 1 = 0$ és a $\sim 0 = 1$ szabályok szerint. A célhely kiválasztása az **AND** műveletnél leírtak szerint történik itt is.

Bitenkénti KOMPLEMENTÁLÁS (~) művelet		
Assembly utasítás	Jelentése	C megfelelője
COMF f,d,a	$\sim(f) \rightarrow \text{cél}$	$j = \sim j ;$

Mire valók a bitenkénti logikai műveletek? Egy egyszerű példa az ASCII karakterek nagybetűsből kisbetűsbe alakítása, vagy az ellenkező irányú konverzió. Az 'A' betű ASCII kódja például 0x41. Ha ezt a kódot megengedő VAGY kapcsolatba hozzuk a 0x20 számmal, akkor az eredmény 0x61 lesz, ami az 'a' betű ASCII kódja. Hasonló módon bármelyik betű kódját kisbetűssé alakíthatjuk a karakterkód 5. bitjének 1-be állításával (a bitek számozását a legkisebb helyiértéktől, 0 sorszámmal kezdjük). Ha a fordított irányba kell konvertálni (kisbetűt átalakítása nagybetűkké), akkor pedig 0xDF maszkkal kell bitenkénti **ÉS** kapcsolatba hozni a karakter kódjával, a $0 \& x = 0$ valamint az $0 \& x = x$ azonosságokat. Tehát amelyik biten a maszk értéke 0, azon helyiértéken az **ÉS** művelet eredménye a másik operandus értékétől függetlenül nulla lesz.

Természetesen **egyszerre egynél több bitet is törölhetünk, vagy állíthatunk egybe** a bitenkénti logikai műveletekkel. Az alábbi táblázatban egy-egy példán keresztül mutatjuk be a bitszoport törlését **AND** művelettel, egybe állítását **IOR** művelettel, ellenkező állapotra történő állítását **XOR** művelettel, s az összes bit komplementálását a komplementképző utasítással. Az egyszerűség kedvéért 8 bites adatokon végzünk műveleteket. A komplementálás kivételével ezek a műveletek három gépi utasítást igényelnek.

Tegyük fel, hogy az adatmemóriában tárolt 8 bites változók értékei: **i** = 0x2C; **j** = 0xB3; **k** = 0x8A;

C kód	PIC18 assembly kód	Végrehajtás
i = i & 0x0F; bitszoport törlése AND utasítással	movf i,w andlw 0x0F movwf i	i = 0x2C = 0010 1100 &&&& &&&& maszk = 0x0F = 0000 1111 ----- eredmény = 0000 1100 = 0x0C
j = j 0x0E; bitszoport beállítás IOR művelettel	movf j,w iorlw 0x0E movwf j	j = 0xB3 = 1011 0011 maszk = 0x0E = 0000 1110 ----- eredmény = 1011 1111 = 0xBF
k = k ^ 0xC0 bitek komplementálása XOR művelettel	movf k,w xorlw 0xC0 movwf k	k = 0x8A = 1000 1010 AAAA AAAA maszk = 0xC0 = 1100 0000 ----- eredmény = 0100 1010 = 0x4A
k = ~ k ; bitenkénti komplementálás	comf k,F	k = 0x8A = 1000 1010 komplementálás után ----- eredmény = 0111 0101 = 0x75

AND műveletnél: ha a maszk valamelyik bitje null, akkor az eredmény megfelelő bitje is nulla lesz.

IOR műveletnél: ha a maszk valamelyik bitje 1, akkor az eredmény megfelelő bitje is 1 lesz.

XOR műveletnél: ha a maszk valamelyik bitje 1, akkor az eredmény megfelelő bitje az eredeti adat ellenkezője lesz.

Megjegyzés: Ha nagyon megerősítjük magunkat, akkor a fenti három helyett két utasítással is meg tudnánk oldani a feladatot. Például az **i** = **i** & 0x0F; így is írható assembly nyelven:

```
movlw 0x0F
andwf i
```

Bit beállító, bit törlő és bit billegtető utasítások

Valamelyik memóriarekesz vagy speciális funkciójú regiszter egyetlen bitjének törlése, beállítása, vagy ellenkező állapotba billentése olyan gyakran előforduló művelet, hogy külön utasítások formájában is meg vannak valósítva (**bcf**, **bsf**, **btg**). Természetesen az előbb ismertetett bitenkénti logikai műveletekkel is elvégezhetők volnának, de akkor egyenként két-két utasítást igényelnének. Ezen utasítások használatát és működését az alábbi táblázatban foglaltuk össze:

Megnevezés	PIC18 assembly kód	Jelentés
Bit beállítás	bsf f,b,a	1 →f
Bit törlés	bcf f,b,a	0 →f

Bit átbillentés	btg f,b,a	$\sim(f < b) \rightarrow f < b$
-----------------	-----------	---------------------------------

Ezeknél az utasításoknál **f** az operandus címének alsó 8 bitje, **b** a megváltoztatni kívánt bit sorszáma (0..7 közötti szám), **a** pedig az access bit (a=0 esetén az Access Bank-ban, a=1 esetén pedig a **BSR** regiszter által kijelölt memória lapon történik a műveletvégzés).

Nézzünk néhány egyszerű példát ezen utasítások használatára! Legyenek az adatmemóriában tárolt 8 bites változók értékei: i = 0x2C; j = 0xB3; k = 0x8A. Töröljük k 7. bitjét, állítsuk be j 2. bitjét, s komplementáljuk i 5. bitjét! (Az egyszerűség kedvéért feltételezzük, hogy mindhárom változó az Access Bank-ban helyezkedik el!)

C kód	PIC18 assembly kód	Végrehajtás
uint8 i,j,k; k = k & 0x7F; k 7. bitjének törlése	bcf k,7	k = 0x8A = 1000 1010 bcf k,7 ----- eredmény = 0000 1010 = 0x0A
uint8 i,j,k; j = j 0x04; j 2. bitjének beállítása	bsf j,2	j = 0xB3 = 1011 0011 bsf j,2 ----- eredmény = 1011 0111 = 0xB7
uint8 i,j,k; i = i ^ 0x20; i 5. bitjének komplementálása	btg i,5	i = 0x2C = 0010 1100 btg i,5 ----- eredmény = 0000 1100 = 0x0C

Jegyezzük meg, hogy ezen utasításoknak nincs C nyelvi megfelelője. C programokban az egyes bitek törlését, beállítását és komplementálását is a korábban ismertetett logikai műveletekkel és megfelelő maszk érték választásával lehet megvalósítani.

A STATUS regiszter

A státuszregiszter egy nagyon fontos speciális funkciójú regiszter. Mostanáig nem foglalkoztunk vele, de immár elengedhetetlen hogy megismerkedjünk vele, hiszen a feltételes program elágazásokhoz szükségünk lesz rá. A státuszregiszter több olyan jelzőbitet tartalmaz, amelyek a korábban ismertetett bitkezelő utasításokkal is törölhetők vagy beállíthatók, de elsődleges arra szolgálnak, hogy a végrehajtott utasítások mellékhatásaként, ez utasítások eredményétől függően álljanak be 1, vagy 0 értékre. Ezek a státuszbitek jelzik számunkra, hogy a végrehajtott utasítás eredménye nulla volt-e, keletkezett-e átvitel, vagy történt-e túlsordulás, stb.

-	-	-	N	OV	Z	DC	C
---	---	---	---	----	---	----	---

Az egyes bitek jelentése:

Bit	Angol elnevezés	A státuszbit jelentése
C	Carry	Átvitel történt az előző művelet során
DC	Decimal Carry	Tízes átvitel (BCD számbábrázolású műveleteknél)
Z	Zero	Nulla lett az előző művelet eredménye
OV	Overflow	Túlsordulás történt az előző művelet során
N	Negative	Negatív szám lett az előző művelet eredménye
-		Nincs implementálva

A státuszbitek közül **C**, **DC**, **Z**, **OV** és **N** bitműveletekkel is törölhető, illetve beállítható, ezen kívül az aritmetikai és a logikai műveletek mellékhatásaként áll be, az eredménytől függően. A zero (Z) bit akkor áll 1-be, ha egy utasítás nullát eredményez, különben törlődik (0). Az átvitel (C, Carry) bit akkor áll 1-be, ha átvitel történik a legmagasabb helyiértékű bitről. A decimális átvitel (DC, Decimal Carry), a túlsordulás (OV, Overflow) valamint a negatív (N, Negative) állapotjelző bitek szerepével a következő fejezetben fogunk megismerkedni.

Honnan tudjuk, hogy melyik utasítás módosítja mellékhatásként a státuszbiteket, s azok közül melyiket? Ha fellapozzuk a mikrovezérlő adatlapját, akkor az utasításkészletet ismertető összefoglaló táblázat utolsó előtti oszlopában megtaláljuk az adott utasítás által módosított státuszbitek felsorolását. Például:

Utasítás	Szintaxis	Jelentés	Szó	ciklus	Érintett státuszbitek
ADDWF	ADDWF f,d,a	(f) + (WREG) → cél	1	1	C,DC,Z,OV,N
ANDWF	ANDWF f,d,a	(f) & (WREG) → cél	1	1	Z, N
MOVF	MOVF f	(f) → cél	1	1	Z, N
GOTO	GOTO cím	cím-hez ugrik	2	2	egyik sem
SUBWF	SUBWF f,d,a	(f) - (WREG) → cél	1	1	C,DC,Z,OV,N

Mint látjuk, az **ADDWF** utasítás az ALU összes státuszbitjét érinti, a **MOVF** utasítás csak az **N** és **Z** biteket, a **GOTO** utasítás egyiket sem, és így tovább.

A Carry és a Zero bitek

A státuszregiszter 0. bitje Carry (átvitel, C) bit, az 2. bit pedig Zero (nulla, Z) bit néven ismert. Ezeket a BSF/BCF utasításokkal is beállíthatjuk, illetve törölhetjük, de sok utasítás mellékhatásaként is megváltozhat az értékük.

Összeadásnál:

A Zero bit értéke 1 lesz, ha a művelet eredménye nulla.

A Carry bit értéke pedig akkor lesz 1, ha egy művelet elvégzése során átvitel keletkezik a legmagasabb helyiértékű biten, ha az összeg > 255 (0xFF).

Néhány mintapélda:

0xF0 +0x20 ----- 0x10 Z=0, C=1	0x00 +0x00 ----- 0x00 Z=1, C=0	0x01 +0xFF ----- 0x00 Z=1, C=1	0x80 +0x7F ----- 0xFF Z=0, C=0
---	---	---	---

Kivonásnál:

A Zero bit értéke 1 lesz, ha a művelet eredménye nulla.

A Carry bit **törődik**, ha egy művelet elvégzése során áthozatal történik a legmagasabb helyiértékű biten (előjel nélküli alulcsordulás, az eredmény < 0, azaz nagyobb számot vonunk ki kisebb számból).

A Carry bit értéke 1 lesz, ha nem történt áthozatal.

Néhány mintapélda:

0xF0 -0x20 ----- 0xD0 Z=0, C=1	0x00 -0x00 ----- 0x00 Z=1, C=1	0x01 -0xFF ----- 0x02 Z=0, C=0
---	---	---

Megjegyzés: Kivonás eredményeként Z=1, C=0 állapot sohasem jöhet létre, mivel nullát csak akkor kapunk eredményül, ha két egyenlő számot vonunk ki egymásból, ilyenkor viszont nem keletkezik átvitel.

A Carry bit kivonás utáni "viselkedését" a következő megfontolás alapján érthetjük meg: Az A-B kivonási műveletet a mikrovezérlő aritmetikai egysége valójában az $A + (\sim B + 1)$ összeadásra vezeti vissza, s a Carry bit értéke az $A + (\sim B + 1)$ összeadásnak megfelelően áll be. Az $(\sim B + 1)$ mennyiséget egyébként B **kettes komplementésének** nevezzük, s a negatív számok ábrázolásánál még találkozunk vele.

Nézzük meg ezt az 0xF0 - 0x20 kivonás példáján!

0xF0-0x20	$\sim 0x20$	$0xF0 + (\sim 0x20) + 1$
0xF0 -0x20 ----- 0xD0 Z=0, C=1 (nincs áthozat)	0x20 = 0010 0000 $\sim 0x20$ = 1101 1111 = 0xDF	0xF0 +0xDF +0x01 ----- 0xD0 Z=0, C=1 (van átvitel)

Feltételes programvégrehajtás bitvizsgálattal

A Zero és a Carry státuszbitek felhasználásnak egyik fontos területe a programelágazások megvalósításához szükséges feltételvizsgálat. Most a **"bitvizsgálat, ugorj, ha a bit törölt"** (btfs - 'bit test f, skip if clear') és a **"bitvizsgálat, ugorj, ha a bit beállított"** (btfs - 'bit test f, skip if set') utasításokat fogjuk használni feltételes programvégrehajtásra.

btfs f,b,a ;átugorja a következő utasítást, ha az **f** regiszter b. bitje törölt (azaz '0')

btss f,b,a ;átugorja a következő utasítást, ha az **f** regiszter b. bitje beállított (azaz '1')

A PIC18 mikrovezérlők a fentiekén kívül számos további feltételvizsgáló és feltételes programvégrehajtást végző utasítással rendelkeznek, amelyekkel majd később fogunk megismerkedni.

Az alábbi egyszerű program a btfs utasítás használatát mutatja be:

Az **udata_acs** direktíva az Access Bank területét jelöli ki az utána következő változók elhelyezéséhez. A **bemenet** és a **kimenet** nevű változók 1-1 bájtnyi tárhelyet foglalnak el (res = reservation, lefoglalás). A kódot a **CODE 0x000** direktívával most a programtároló memória legelejétől kezdve helyeztük el. Szimulátorban való futtatáshoz így megfelel, de a mikrovezérlőbe ne égezzük be ezt a programot, mert felülírja a bootloadert!

Az első két utasítás szerepe az előkészítés: nullát töltünk a **W** munkaregiszterbe, majd felülírjuk vele a **adat**

nevű változót. Első alkalommal így teljesülni fog a **btfsc adat,0** utasítás feltétele, a program átlép a **goto paratlan** utasításra, s a páros számokat veszi sorra. A számoknak a **kimenet** változóba történő írogatásának természetesen nincs sok értelme, de gondoljunk bele: ugyanígy írogathatnánk a számokat egy periféria kimenő regiszterébe is, amellyel pl. soros porton keresztül kiküldhetnénk a számokat egy másik eszköznek (pl. egy terminál képernyőjére).

A páros számok kiírása után a **goto újra** parancs hatására az **adat** nevű változó tartalmának megnövelésével folytatódik a program, majd visszaugrik a feltételvizsgálathoz. A második lefutáskor a **btfsc adat,0** utasítás feltétele nem teljesül, ezért most a másik ágon folytatódik a program, a páratlan számok sorravételével.

```
#include "pl8f14k50.inc"
    udata_acs      ; Adatterület lefoglalása
    adat    res 1    ; 1 bájtnyi területet foglal
    kimenet  res 1    ; 1 bájtnyi területet foglal

    CODE 0x000      ; Itt kezdődik a program
    movlw 0          ; adat<0>=0 lesz
    movwf adat
ciklus: btfsc adat,0 ; átugorja a következőt, ha adat<0>=0
        goto paratlan
paros:  movlw 0
        movwf kimenet
        movlw 2
        movwf kimenet
        movlw 4
        movwf kimenet
        goto újra
paratlan: movlw 1
        movwf kimenet
        movlw 3
        movwf kimenet
        movlw 5
        movwf kimenet
        incf adat ; megnöveljük adat értékét
        goto ciklus ; végtelen ciklus
    END
```

A változók értékének változását az MPLAB IDE **File registers** ablakában követhetjük nyomon. A program végtelen ciklusban fut, s minden páratlanadik lefutáskor a páros számokat, a következő ciklusban pedig a páratlan számokat veszi sorra (0,2,4 majd 1,3,5). Ennek a programnak természetesen nincs gyakorlati haszna, csupán a szemléltetés kedvéért mutattuk be!



Feltételvizsgálatok a C programnyelvben

Mielőtt a feltételes programvégrehajtással folytatnánk az ismerkedést, tekintsük át a C programnyelvben használatos feltételvizsgálati lehetőségeket, amelyekkel többnyire az if és a ciklusszervező utasításokban találkozunk!

Relációs operátor	Jelentése
==, !=	Egyenlő, Nem egyenlő
>, >=	Nagyobb, Nagyobb,vagy egyenlő
<, <=	Kisebb, Kisebb, vagy egyenlő
&&	Logikai AND (ÉS)
	Logikai OR (VAGY)
!	Negáció (logikai tagadás)

A C programokban ha egy feltételvizsgálat nullától különböző értéket eredményez, akkor a feltételt logikailag igaznak (TRUE) tekintjük.

Vigyázzunk: A logikai tagadás és a bitenkénti komplementképzés különböznek egymástól, eltérő eredményre vezetnek!

!i	nem ugyanaz, mint	~i
i = 0xA0		i = 0xA0
!(i)	 0	~(i)
		 0x5F

A fentiekhez hasonlóan különbözik egymástól a & bitenkénti logikai művelet az && logikai relációtól és az | bitenkénti logikai művelet az || logikai relációtól. A relációs operátorok (!, &&, ||) ugyanis mindig csak abban a tekintetben vizsgálják az operandus(oka)t, hogy nulla, vagy nem nulla értékűek, s a logikai vizsgálat eredménye egyetlen bitnyi adat (0 vagy 1) lesz. A bitenkénti logikai műveletek ezzel szemben bitenként, egymástól függetlenül minden helyiértéken elvégzik a kiértékelést, s az eredmény is 8 vagy 16 bites lesz. Figyeljünk ezekre az apró de annál fontosabb különbségekre, mert ellenkező esetben kellemetlen meglepetésként érhet bennünket, hogy a C programunk nem azt csinálja, amit elvárunk tőle!

Az alábbi példákon C egyenlőség- és egyenlőtlenség-vizsgálatokat mutatunk be (8 bites műveletek)

```

unsigned char a,b,a_lt_b, a_eq_b, a_gt_b, a_ne_b;
a = 5; b = 10;
a_lt_b = (a < b); // eredménye 1
a_eq_b = (a == b); // eredménye 0
a_gt_b = (a > b); // eredménye 0
a_ne_b = (a != b); // eredménye 1

```

Példák C logikai műveletekre (8 bites műveletek)

```

unsigned char a_lor_b, a_bor_b, a_lneg_b, a_bcom_b;
a = 0xF0; b = 0x0F;
a_land_b = (a && b); //logikai ÉS, eredménye 1
a_band_b = (a & b); //bitenkénti ÉS, eredménye 0
a_lor_b = (a || b); //logikai VAGY, eredménye 1
a_bor_b = (a | b); //bitenkénti VAGY, eredménye 0xFF
a_lneg_b = (!b); //logikai negálás, eredménye 0
a_bcom_b = (~b); //bitenkénti negálás, eredménye 0xF0

```

Nulla, nem nulla típusú feltételvizsgálatok

Az előző két listán szereplő utasítások nem mondhatók tipikusnak, a gyakorlatban ritkán fordulnak elő. A feltételvizsgálatokkal leggyakrabban az **if** vagy a ciklusszervező utasításokban találkozhatunk. A C programozási nyelvben használt **if-else** utasítás formáját az alábbi ábrán mutatjuk be. Jegyezzük meg, hogy az **if** ág programtörzse akkor kerül végrehajtásra, ha a feltétel teljesül (a feltételvizsgálat eredménye = 1), az **else** ág törzse pedig akkor kerül végrehajtásra, ha a feltétel nem teljesült (a feltételvizsgálat eredménye = 0).

```

if (feltételvizsgálat) {
    if_törzse <-- akkor hajtódik végre, ha az eredmény nem nulla ("igaz")
} else {
    else_törzs <-- akkor hajtódik végre, ha az eredmény nulla ("hamis")
}

```

Az **if** és az **else** törzse egynél több utasítást is tartalmazhat. Az **else** ág használata opcionális, elhagyható.

Hogyan fogalmazzuk meg a feltételvizsgálatot, ha például az *i* változó nulla, vagy nem nulla értékétől függően akarunk végrehajtani valamilyen utasítást, például a *j = i + j* összeadást? Az alábbi ábrán több lehetőséget is bemutatunk:

Akkor hajtódik végre, ha <i>i</i> = 0	Akkor hajtódik végre, ha <i>i</i> nullától különbözik
<pre> if (!i) { j = i + j; } </pre>	<pre> if (i) { j = i + j; } </pre>

A fenti utasításokat így is írhatjuk:

Akkor hajtódik végre, ha <i>i</i> = 0	Akkor hajtódik végre, ha <i>i</i> nullától különbözik
<pre> if (i == 0) { j = i + j; } </pre>	<pre> if (i != 0) { j = i + j; } </pre>

C programok írásánál gyakori hiba az **egyenlőség-vizsgálat** (==) és az **értékkadás** (=) jelének összekeverése. Az alábbi példákon megmutatjuk, hogy ez milyen meglepetést okozhat:

Hibás kód!	Helyes kód
<pre> if (i = 5) { j = i + j; //Mindig végrehajtódik, mert az i=5 //értékkadás visszatérési értéke 5 lesz! } </pre>	<pre> if (i == 5) { //Az i==5 vizsgálat csak j = i + j; // akkor ad nullától különböző // értéket, amikor i értéke 5 } </pre>

A bitenkénti és a logikai AND művelet különbözősége

Korábban már volt róla szó, hogy a bitenkénti és a logikai ÉS műveletek is könnyen összetéveszthetők. Az alábbi példákban úgy szemléltetjük a kettő különbségét, hogy bemutatjuk a feltételvizsgálat helyes olvasatát, s egy konkrét esetre megmutatjuk, hogy a hasonlóan tűnő feltételvizsgálatok különböző eredményre vezetnek.

```

if (i && j) { // Ha i nem nulla ÉS j sem nulla, akkor...
    /* tedd ezt */ // i = 0xA0, j = 0x0B esetén
} // i && j = 1 (tehát teljesül a feltétel)

```

```

if (i & j) { // Ha i bitenkénti ÉS kapcsolata j-vel nem nulla, akkor...
    /* tedd ezt */ // i = 0xA0, j = 0x0B esetén
} // i & j = 0 (tehát nem teljesül a feltétel)

```

A bitenkénti és a logikai OR művelet különbözősége

Egy kicsit más a helyzet a megengedő VAGY műveleteknél. Habár a logikai művelet (||) itt is csak az operandusok nullától való különbözőségének tényét veszi figyelembe, a bitenkénti művelet (|) pedig minden bitre külön-külön végzi el ugyanezt, a kétféle feltételvizsgálat eredménye csak számszerűségében tér el egymástól, hatása ugyanaz lesz. Az alábbi példában mindkét esetben teljesül a feltétel, tehát az **if** törzs végrehajtásra kerül.

```
if (i || j) {           // Ha i nem nulla VAGY j nem nulla, akkor...
    /* tedd ezt */     // i = 0xA0, j = 0x0B esetén
}                     // i || j = 1 (tehát teljesül a feltétel)
```

```
if (i | j) {           // Ha i bitenkénti VAGY kapcsolata j-vel nem nulla, akkor...
    /* tedd ezt */     // i = 0xA0, j = 0x0B esetén
}                     // i & j = 0xAB (tehát teljesül a feltétel)
```

Nullától való különbözőség vizsgálata

Az alábbi programrészletben egy **if** utasítás szerepel, amely az **i** változó nullától való különbözőségét vizsgálja. Az **if(i)** feltételvizsgálat ekvivalens az **if(!!=0)**-val, nincs előnye egyiknek sem a másikhoz képest. Assembly nyelven a feltételvizsgálatot a **mov i,f** utasítással készíthetjük elő, ami a változó értékét önmagába másolja vissza. A látszólag értelmetlen utasítás azonban mellékhatásként beállítja a **STATUS** regiszter **N** és **Z** bitjeit. Ezek közül most a **Z** bit érdekel bennünket, ami '0' lesz, ha a vizsgált változó értéke nullától különböző. Ezután a **btfs STATUS,Z** utasítás (bitvizsgálat, és ugrás, ha **Z=0**) átugorja a következő utasítást, ha **Z=0**, s így az **if** törzs végrehajtásra kerül. Ha azonban **Z=1**, akkor a **goto end_if** utasítás kerül végrehajtásra, ami átugorja az **if** törzset, s a program az **end_if** címkétől folytatódik.

C nyelven

```
unsigned char i,j;

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

//további utasítások
```

Assembly nyelven

```
mov    i,F           ; i = i
btfs   STATUS,Z      ; ugrás, ha Z=0
goto   end_if        ; Z=1, i=0
if_body:
movf   i,w           ; w = i
addwf  j,F           ; j = j + i
end_if:
; további utasítások
```

Assembly nyelven másképp is megírhatjuk a fenti feltételes elágazást, ha a **btsc** és **goto** utasításpárt egy **bz end_if** utasítással helyettesítjük. A **bz** utasítás jelentése "branch if zero", azaz: **elágazás, ha a Z státuszbit = '1'**. A **bz** utasítás egyike a PIC18 elágaztató utasításainak, amelyek feltételes ugróutasításként működnek, egy vagy több státuszbit beállítottságától függően.

Az egy státuszbit állapotát vizsgáló egyszerű elágaztató utasításokat az alábbi táblázatba foglaltuk össze:

Utasítás	Emlékeztető	A végrehajtott művelet
bz <címke>	branch if zero	A címkehez ugrik, ha Z=1
bnz <címke>	branch if nonzero	A címkehez ugrik, ha Z=0 (nem nulla)
bc <címke>	branch if carry	A címkehez ugrik, ha C=1
bnc <címke>	branch if no carry	A címkehez ugrik, ha C=0 (nincs átvitel)
bn <címke>	branch if negative	A címkehez ugrik, ha N=1
bnn <címke>	branch if not negative	A címkehez ugrik, ha N=0 (nem negatív)
bov <címke>	branch if overflow	A címkehez ugrik, ha V=1
bnov <címke>	branch if no overflow	A címkehez ugrik, ha V=0 (nincs túlcordulás)
bra <címke>	branch unconditional	A címkehez ugrik, feltétel nélkül

A táblázat utolsó sorában szerepel a **bra <címke>** utasítás is, ami a **GOTO** utasításhoz hasonlóan feltétel nélküli ugrást jelent. A feltétel nélküli **bra** és a **GOTO** utasítás azonban különböznek egymástól. A **bra** utasítások egyszavasak, s kódjuk egy legfeljebb 11 bites relatív címet tartalmazhat, ami azt jelenti, hogy az aktuális programszámlálóhoz képest 1023 utasítást ugorhatunk át előre, vagy visszafelé. Ez azt jelenti, hogy a 16 KB memóriával rendelkező PIC18F14K50 mikrovezérlőben **bra** utasítással nem tudunk elugrani a programmemória egyik végétől a másikba. Természetesen a relatív cím kiszámítását a fordítóprogram elvégzi helyettünk. A **GOTO** utasítás ezzel szemben két memóriaszót vesz igénybe, s a teljes (2MB) elvi címtartomány megcímezhető vele.

Nézzük akkor meg, hogy a **bz** utasítással hogy néz ki a módosított programrészlet!

C nyelven

```
unsigned char i,j;
if (i) {
    //if törzs
    j = i + j;
}

//további utasítások
```

Assembly nyelven

```
mov    i,F        ; i = i
bz     end_if      ; ugrás, ha Z=1
movf   i,w         ; w = i
addwf  j,F         ; j = j + i
end_if:
; további utasítások
```

A programunk így nemcsak rövidebb lett (a **GOTO** utasítás elhagyása miatt 2 utasítás-szóval, vagyis 4 bájtal kevesebb a memória-felhasználás), hanem az áttekinthetőség is javult.

Figyeljük meg, hogy az assembly programban használt **bz** utasítás működése ellentétes a C programbeli **if** utasításával: az **if** utasítás végrehajtja, a **bz** utasítás pedig átugorja az **if** törzset, ha teljesül az utána írt feltétel. Így nem meglepő, hogy a helyes működéshez a **bz** utasításnál az **if** feltételének *ellentettjét* (a logikai negáltját) kell írunk. Ha például a fenti programban az **i!=0** feltétel helyett a **i==0** feltétel teljesülését akarnánk előírni, akkor a C programban **if (!i)**, az assembly programban pedig **bnz end_if** feltételt kellene írni.

A feltételes programelágazás általánosabb (if-else) alakja

Az alábbi ábrán láthatjuk a feltételes programelágazás általánosabb formáját C nyelven és a helyenként pszeudo-utasításokkal megtűztelt assembly megfelelőjét. Mint látható, két elágaztató utasítással megoldható az if-else szerkezet leutánczása: a **bz** utasítás feltételes ugrást hajt végre, ha az **if** utasításban előírt feltétel nem teljesül (kihagyja az **if** törzset). A **bra** utasítás feltétel nélküli ugrás, ami az **if** törzs végéről az **else** törzs utáni első utasításra ugrik. Ez biztosítja, hogy az **else** ág törzse ne kerüljön végrehajtásra, ha az **if** utasításban megfogalmazott eredeti feltétel teljesült.

C nyelven

```
unsigned char i,j,k;
if (i) {
    j = i + j;
}
else {
    j = k + j;
}

//további utasítások
```

Assembly nyelven

```
mov    i,F        ; i = i
bz     else_ag     ; ugrás, ha Z=1
movf   i,w         ; w = i
addf   j,F         ; j = j + i
bra    end_if
else_ag:
movf   k,w         ; w = k
addwf  j,F         ; j = j + k
end_if:
; további utasítások
```

Ne feledkezzünk meg róla, hogy a C program **if** utasításban szereplő feltétel ellentettjét kell az Assembly program első elágaztató utasításába írni (most például **bnz** helyett **bz** kellett)!

Egyenlőség és nem egyenlőség vizsgálata

Az alábbi programrészlet bemutatja, hogyan fogalmazhatjuk meg assembly nyelven a **i==j** feltételt vizsgáló **if** utasítást. Az egyenlőség-vizsgálathoz az **i-j** kivonást használjuk fel, amit egy **bnz** utasítás követ. A kivonás művelete ugyanis a kivonás eredményétől függően állítja be a státuszregiszter bitjeit, így a **Z=0** állapot jelzi, ha **i** és **j** nem egyenlők egymással (ekkor kell átugrani az **if** törzset. Itt most nem számít, hogy a **i-j** vagy a **j-i** különbséget képezzük, egyenlőség esetén ugyanúgy nullát kell kapnunk.

C nyelven

```
unsigned char i,j;
if (i==j) {
    //if törzs
    j = i + j;
}

//további utasítások
```

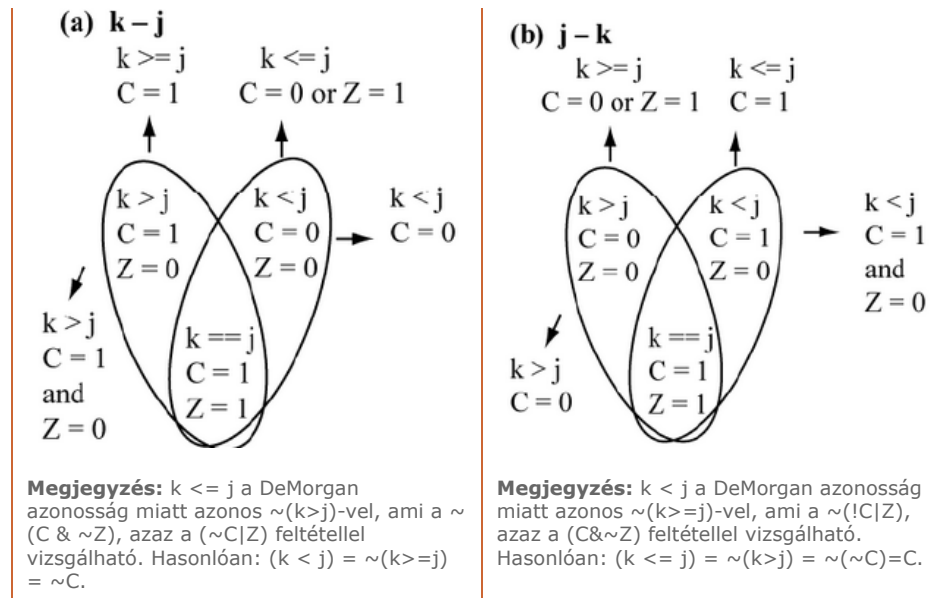
Assembly nyelven

```
movf   j,w         ; w = j
subwf  i,w         ; w = i - j
bnz    end_if      ; ugrás, ha Z=0, i != j
movf   i,w         ; w = i
addwf  j,F         ; j = j + i
end_if:
; további utasítások
```

Megjegyzés: A fentiek alapján könnyen átírhatjuk a programrészletet arra az esetre, ha az **i!=j** feltételt akarjuk vizsgálni, csak a **bnz end_if** utasítást kell kicserélni egy **bz end_if** utasításra.

Egyenlőtlenség vizsgálata kivonással, a Z és C bitek alapján

A kisebb, nagyobb, kisebb vagy egyenlő, nagyobb vagy egyenlő vizsgálatokat a korábban bemutatottakhoz hasonlóan kivonással is vizsgálhatjuk. Két változó (például **j** és **k**) esetén bármelyiket kivonhatjuk a másikból, ennek megfelelően a **Z** és a **C** státuszbitek alakulását nehéz áttekinteni. Az áttekintés segítésére az alábbi ábrákon összefoglaltuk az összes lehetőséget.



Az alábbi példában a $k > j$ feltételt vizsgáljuk, a $k - j$ kivonást használva.

C nyelven

```
unsigned char k,j;
if (k > j) {
    //if törzs
    k = k + j;
}
//további utasítások
```

Assembly nyelven

```
movf j,w      ; w = j
subwf k,w     ; w = k - j
bnc end_if    ; ugrás, ha C=0, k < j
bz end_if     ; ugrás, ha Z=1, k = j
movf j,w      ; w = j
addwf k,F     ; k = k + j
end_if:
; további utasítások
```

A $k > j$ feltétel ellentettje (ami az if törzs kikerüléséhez szükséges) $k \leq j$ alakba írható. Mivel a $k - j$ kivonást választottuk, a $C=0$ és $Z=1$ esetén kell átugrani az if törzset, tehát két bitvizsgáló utasításra van szükség.

Nézzük meg a másik lehetőséget is, amikor a **$k > j$ feltételt a $j - k$ kivonást használva vizsgáljuk!** Az if törzs kikerüléséhez szükséges feltétel $k \leq j$ lesz, melynek feltétele $C=1$. Tehát ebben az esetben csupán egyetlen **bc** feltételvizsgáló utasításra van szükség, tömörebb és áttekinthetőbb a kód.

C nyelven

```
unsigned char k,j;
if (k > j) {
    //if törzs
    k = k + j;
}
//további utasítások
```

Assembly nyelven

```
movf k,w      ; w = k
subwf j,w     ; w = j - k
bc end_if     ; ugrás, ha C=1, k <= j
movf j,w      ; w = j
addwf k,F     ; k = k + j
end_if:
; további utasítások
```

Összehasonlítás, elágazás előjel nélküli feltételvizsgálattal

Az előzőekben példákat láttunk arra, hogy az egyenlő, nem egyenlő, kisebb, stb. feltételvizsgálatokat hogyan valósíthatjuk meg egy kivonás és egy (vagy két) bitvizsgáló utasítással. A kivonás és a státuszbiték nyomonkövetése nem könnyű feladat. Egyes esetekben gondot okozhat, az is, hogy a vizsgálat során az egyik regiszter tartalmát felülírjuk. Kényelmesebb utat kínálnak a PIC18 mikrovezérlők összehasonlító (CPFSEQ, CPFSGT, CPFSLT - Compare f and W, skip if...) utasításai, amelyek az (f) - (W) kivonással összehasonlítják a megírt **f** memóriarekesz és a **W** munkaregiszter tartalmát, s átugorják a következő utasítást a megadott feltétel ($=$, $<$, $>$) teljesülése esetén.

Az alább felsorolt előjel nélküli összehasonlítást végrehajtó utasítások tehát az előző példákhoz hasonlóan kivonással dolgoznak, de nem módosítják egyik operandus tartalmát sem. Szerepük csupán az, hogy a kivonás eredményétől függően a megadott feltételteljesülése esetén átlépjenek egy utasítást.

Szintaxis	Művelet	Ugrás akkor, ha..
CPFSEQ f,a	$(f) - (W)$	$(f) = (W)$
CPFSGT f,a	$(f) - (W)$	$(f) > (W)$
CPFSLT f,a	$(f) - (W)$	$(f) < (W)$

Nézzünk egy egyszerű mintapéldát ezen utasítások használatára!

C nyelven

```

unsigned char j,k;
if (k > j) {
    //if törzs
    k = k + j;
}
//további utasítások

```

Assembly nyelven

```

movf    j,W      ; W = j
cpfsft  k        ; k-j
bra     end_if    ; ugrás, ha k <= j
movf    j,W      ; W = j
addwf   k,F      ; k = k + j
end_if:
; további utasítások

```

A fenti programban az **if** törzs teljesülésének feltétele **k>j**, tehát ha a **j** változó tartalmát betöltjük a **W** munkaregiszterbe, akkor a táblázat szerint a **cpfsft k** utasítást kell használnunk.

A **cpfsft**, **cpfslt** és **cpfseq** utasítások fő előnye az, hogy a kivonást és a feltételvizsgálatot egyetlen utasításban elvégzik, ezért áttekinthetőbb, s néha talán rövidebb kódot eredményeznek. Azonban nagy hátrányuk, hogy ezek az utasítások nem használhatók a több-bájtos mennyiségek összehasonlítására, s hogy a **cpfsft/cpfslt** utasítások nem használhatók az előjeles számok összehasonlítására sem. Az általánosabb módszer tehát a korábban bemutatott kivonás és a megfelelő elágaztató utasítás kombinációja, ahogy ezt majd a következő fejezetben is látni fogjuk.

Számkonstanssal történő előjel nélküli összehasonlítás

A PIC18 mikrovezérlők utasításkészlete csak a **W** munkaregiszter számkonstanssal történő közvetlen összehasonlítását teszi lehetővé, ezért vagy a vizsgálandó regiszter tartalmát, vagy az összehasonlítandó számot be kell tölteni a munkaregiszterbe, majd ezt követően végezzük az összehasonlítást, az előzőekben leírtak szerint. Az alábbiakban mindét esetre mutatunk egy-egy példát:

C nyelven

```

unsigned char i,j;
if (i > 0x40) {
    //if törzs
    i = i + j;
}
//további utasítások

```

Assembly nyelven

```

movlw   0x40      ; W = 0x40
cpfsft  i         ; i > 0x40 ?
bra     end_if    ; ugrás, ha nem
movf    j,W      ; W = j
addwf   i,F      ; i = i + j
end_if:
; további utasítások

```

C nyelven

```

unsigned char i,j;
if (i > 0x40) {
    //if törzs
    i = i + j;
}
//további utasítások

```

Assembly nyelven

```

movf    i,W      ; W = i
sublw   0x40      ; 0x40 - i
bc     end_if    ; ugrás, ha C=1
movf    j,W      ; W = j
addwf   i        ; i = i + j
end_if:
; további utasítások

```

Megjegyzés: Vigyázzunk, a második példában a **sublw 0x40** utasítás a **0x40 - W** kivonást hajtja végre, ezért itt a **C=1** állapot jelzi, hogy nem teljesül a feltétel, s a **bc end_if** utasítással kerülhetjük ki az **if** törzset (lásd az alábbi táblázat 5. esetét, $j=i$ és $k=0x40$ helyettesítéssel).

Az előjel nélküli feltételvizsgálatok összefoglalása

Az alábbi táblázatban összefoglaltuk az előjel nélküli mennyiségekkel dolgozó feltételes vizsgálatokat. Az első oszlopban az előírt feltétel szerepel, C szintaxisban. A második oszlop azt mutatja, hogy milyen művelettel (SUBWF vagy MOVF utasítás) állíthatók be a státuszbitok az adott feltétel vizsgálatához. A harmadik oszlopban azt tüntettük fel, hogy a feltétel teljesülését melyik státuszbit és milyen állapota jelzi. A negyedik és az ötödik oszlop pedig azt mutatja meg, hogy milyen ugróutasítással vizsgálhatjuk meg a feltétel teljesülését ("Igaz" ág), vagy nem teljesülését ("Hamis" ág), és irányíthatjuk a programot a megfelelő ághoz.

Feltétel	Vizsgálat	Ha teljesül...	"Igaz" ághoz ugrik	"Hamis" ághoz ugrik
1. $j == 0$	$j = j$ (movf j)	$Z = 1$	bz	bnz
2. $j != 0$	$j = j$ (movf j)	$Z = 0$	bnz	bz
3. $j == k$	$j - k$ vagy $k - j$	$Z = 1$	bz	bnz
4. $j != k$	$j - k$ vagy $k - j$	$Z = 0$	bnz	bz
5. $j > k$	$k - j$	$C = 0$	bnc	bc
6. $j >= k$	$j - k$	$C = 1$	bc	bnc
7. $j < k$	$j - k$	$C = 0$	bnc	bc
8. $j <= k$	$k - j$	$C = 1$	bc	bnc

A switch utasítás megvalósítása assembly nyelven

A C programokban gyakran előfordulnak láncolt **if-else** struktúrák, amelyekben egy változó értékétől függően kerül valamelyik ág kiválasztásra, s fut le az abban az ághoz előírt tevékenység. Ez a feladatválasztásos

szerkezet annyira alapvető, hogy a C programnyelvbe külön utasításként be is építették **switch** néven, s ezzel tömörebben, áttekinthetőbben fogalmazhatjuk meg, ahogy az alábbi ábrán is láthatjuk.

Láncolt if-else szerkezet	Switch szerkezet
<pre>unsigned char i, j, k; if (i == 1) { k++; } else if (i == 2) { j--; } else if (i == 3) { j = j + k; } else { k = k - j; }</pre>	<pre>unsigned char i, j, k; switch (i) { case 1: k++; break; case 2: j--; break; case 3: j = j + k; break; default: k = k - j; }</pre>

A **switch** utasítás minden **case** blokkja megfelel egy-egy **if** (feltétel) ágának, s mindig a **switch** utasításban megnevezett változó értékét hasonlítja a **case** mellett álló számkonstanshoz. Természetesen a számkonstansoknak nem kell sorbarendezve szerepelniük, s értékük is tetszőleges. A **default** ág akkor kerül végrehajtásra, ha egyik **case** ág feltétele sem teljesült. Fontos szerepe van a **case** ágak végén elhelyezett **break** utasításnak: ez gondoskodik róla, hogy a program ne "csorogjon rá" a következő **case** ágra. A könnyebb megértéshez megadjuk az angol elnevezések magyar jelentését is: switch = kapcsoló, case = eset, default = alapértelmezett, break = megszakítás.

C nyelven

```
unsigned char i, j, k;

switch (i) {

    case 1: k++;
        break;

    case 2: j--;
        break;

    case 3: j = j + k;
        break;

    default: k = k - j;

} // switch vége

//további utasítások
```

Assembly nyelven

```
movlw 1           ; w = 1
subwf i,w         ; i == 1?
bnz case_2        ; nem, akkor tovább
incf k,F          ; k++
bra end_switch    ; break utasítás

case_2:
movlw 2           ; w = 2
subwf i,w         ; i == 2?
bnz case_3        ; nem, akkor tovább
decf j,F          ; j--
bra end_switch    ; break utasítás

case_3:
movlw 3           ; w = 3
subwf i,w         ; i == 3?
bnz default       ; nem, akkor tovább
movf k,w          ; j = j + k
addwf j,F         ; break utasítás
bra end_switch

default:
movf j,w          ; k = k - j
subwf k,F
end_switch:       ; további utasítások
```

Programciklusok

A **while** ciklusszervező utasítás hasonló szerkezetű, mint az **if** utasítás, csupán annyi a különbség, hogy a **while** utasítás törzse mindaddig **ciklikusan ismétlődik**, amíg a **while** utasítás feltételvizsgáló részében a feltétel teljesül.

C nyelven

```
unsigned char i,j;
while (i > j) {
    //if törzs
    j = i + j;
}

//további utasítások
```

Assembly nyelven

```
ciklus_eleje:
movf i,w          ; w = i
subwf j,w         ; w = j - i
bc ciklus_vege    ; ugrás, ha C=1 (i<=j)
movf i,w          ; w = i
addwf j,F         ; j = i + j
bra ciklus_eleje

ciklus_vege:
; további utasítások
```

A feltételvizsgálat ugyanúgy történik, mint az **if** utasításnál. Ha a feltétel nem teljesül, akkor a program a **while** törzs végére ugrik. Ha pedig a feltétel teljesül, akkor végrehajtásra kerül a **while** törzs, majd egy feltétel nélküli ugrással a program visszatér a feltételvizsgálathoz. A fenti példában a feltétel: **i > j**, a nemteljesülés feltétele tehát **i <= j**, vagy megfordítva: **j >= i**.

Ha összehasonlítjuk a **while** és az **if** utasítás assembly nyelven írt kódját, akkor észrevehetjük, hogy a programban az egyetlen különbséget a **while** törzs végén elhelyezett feltétel nélküli ugrás jelenti, ami a ciklikus ismétlést valósítja meg.

Megjegyzés: ha nem gondoskodunk róla, hogy a **while** utasítás feltételét jelentő reláció logikai értéke a **while** törzsben valahány ciklus végrehajtása során hamisra ne változzon, akkor a program végtelen ciklusba kerül!

Az előtesztelő **while** utasításhoz hasonlóan működik a hátultesztelő **do {} while** utasítás. A különbség annyi, hogy **do {} while** utasítás esetén a feltételvizsgálat a **do-while** törzs végrehajtása után történik. Ez azt is

jelenti, hogy legalább egyszer mindenképpen lefutnak a **do-while** törzs utasításai, akkor is, ha lehetetlen feltételt adunk meg. Ez a legfontosabb tulajdonsága, s ez dönti el, hogy egy programrészben while vagy do-while utasítást kell használnunk.

C nyelven

```
unsigned char i,j;
do {
    //if törzs
    j = i + j;
} while (i > j)

//további utasítások
```

Assembly nyelven

```
ciklus_eleje:
    movf i,w      ; w = i
    addwf j,F     ; j = i + j
    movf i,w      ; w = i
    subwf j,w     ; w = j - i
    bnc ciklus_vege ; ugrás, ha C=0 (i>j)
; további utasítások
```

Amint a fenti egyszerű példában láthatjuk, a do-while ciklus szerkezete assembly nyelven egyszerűbb és áttekinthetőbb, hiszen csak egyetlen ugrást tartalmaz.

A C programnyelvben gyakran használják a **for** ciklusszervező utasítást is. Az alábbi ábrán bemutatott összehasonlításból láthatjuk, hogy a **for** utasítás csupán egy kompaktabb felírási módja a **while** ciklusszervező utasításnak. A **for** utasításban a feltételvizsgálaton kívül egy kezdőérték beállítása és egy "léptető" utasítás szerepel. A **while** utasítás esetén a kezdőérték beállítását a **while** utasítás előtt kell elhelyezni, a ciklusváltozó léptetését pedig a ciklus törzsében kell elvégezni.

FOR ciklus

```
#include <p18cxxx.h>
#include <stdio.h>
unsigned int i,j;
void main(void) {
    j=0;
    for(i=1; i<=100; i++) {
        j=j+i;
    }
    printf("sum(1..100) = %u\n",j);
}
```

WHILE ciklus

```
#include <p18cxxx.h>
#include <stdio.h>
unsigned int i,j;
void main(void) {
    i=1; j=0;
    while(i<=100) {
        j=j+i; i++;
    }
    printf("sum(1..100) = %u\n",j);
}
```

A kimenet: sum(1..100) = 5050

A fenti programok az MPLAB szimulátorában kipróbálhatók. Az "stdio.h" állomány becsatolására a printf függvény használata miatt van szükség. A program összeadja a természetes számokat 1-től 100-ig, s az eredményt a standard outputra írja ki (UART1). A szimulátorban engedélyezni kell a képernyőre történő kiírást az Uart1 kimenetről, s akkor a program lefutása után az output ablakban az UART1 fülre kattintva a fenti kimenetet kell kapnunk.

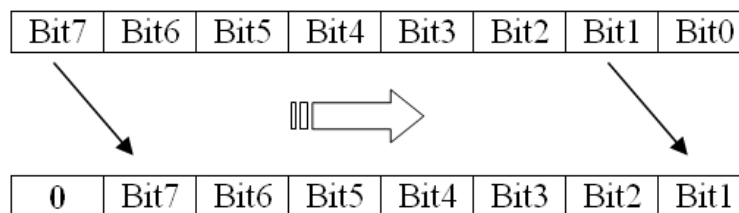
Ellenőrizhetjük azt is, hogy ha az i változó kezdőértékét kellően nagyra választjuk (pl. i=101), hogy az i<=100 feltétel ne teljesüljön, akkor a ciklus egyszer sem fut le. A for ciklus tehát az előltesztelő while utasításnak felel meg.

Léptetés és bitforgatás

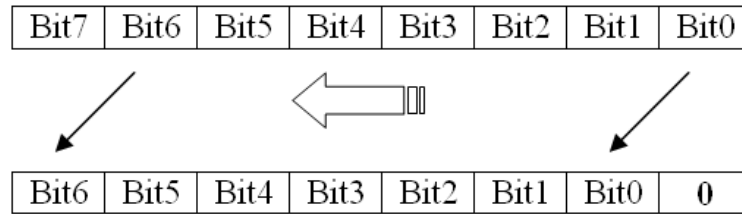
A hardverközelinek számító C nyelv az aritmetikai és logikai műveleteken kívül olyan műveleteket is ismer, mint például egy változó biteinek balra vagy jobbra léptetése. Mivel a balra léptetés azt jelenti, hogy a binárisan ábrázolt szám minden bitje eggyel magasabb helyiértékre kerül, ezért gyakorlatilag kettővel való szorzást jelent (ugyanúgy, ahogyan tízes számrendszerben is tízzel való szorzást jelent, ha egy szám után egy nullát írunk, amivel minden számjegy eggyel magasabb helyiértékre kerül). Hasonló okok miatt a jobbra léptetés kettővel való osztásnak felel meg.

Mit jelent a C nyelvű programban az $i \gg 1$ kifejezés? Ha az i változó előjel nélküli egész típusú, vagy előjeles egész típusú, de nemnegatív értékű, akkor az $i \gg 1$ kifejezés a logikai jobbra léptetésnek felel meg (lásd az alábbi ábrán!). Ha viszont i előjeles egész típusú és nemnegatív értékű, akkor az $i \gg 1$ kifejezés értelmezése implementációfüggő, vagyis az adott fordítóprogramtól függ, hogy hogyan kezeli ezt az esetet. Az **MPLAB C18** fordítója elég "tisztességtelen" ebből a szempontból: **nem őrzi meg a szám előjelét!**

Térjünk vissza az egyszerűbb esethez: $i \gg 1$ előjel nélküli számokkal, logikai jobbra léptetéssel! Amint az ábrán is látható, az eredeti érték minden bitje jobbra lép egy helyet. A legalacsonyabb helyiértékű bit elvész, a legmagasabb helyiértékre pedig 0 kerül.



A jobbra léptetéshez hasonlóan működik a C programokban az $i \ll 1$ balra léptetés is, csak ellenkező irányban: minden bit eggyel magasabb helyiértékre kerül. A legmagasabb helyiértékű bit elvész, a legalacsonyabb helyiértékre pedig 0 kerül.



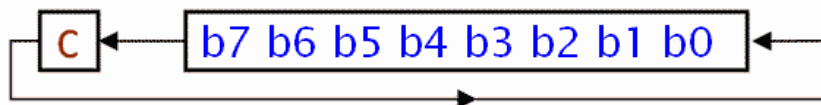
A PIC18 mikrovezérlő család bitforgató és léptető utasításainak készlete az alábbi utasításokkal rendelkezik:

Assembly utasítás	Az utasítás funkciója	Gépi kód	Módosul
RLCF f ,d,a	cél = (f) balra forgatással a Carry biten keresztül	0011 01da ffff ffff	C, N, Z
RRCF f,d,a	cél = (f) jobbra forgatással a Carry biten keresztül	0011 00da ffff ffff	C, N, Z
RLNCF f ,d,a	cél = (f) jobbra forgatással	0100 01da ffff ffff	N, Z
RRNCF f ,d,a	cél = (f) jobbra forgatással	0100 00da ffff ffff	N, Z

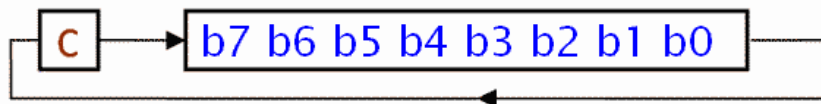
Megjegyzés: Az utasítás **d** és **a** bitjének szerepe megegyezik a korábban tanultakkal, d=0 esetén a cél helye a W munkaregiszter, d=1 esetén pedig a megcímzett regiszterbe íródik vissza az eredmény. Ha az **a** bit értéke nulla, akkor az Access bank-ból, a=1 esetén pedig a **BSR** regiszter által megcímzett memórialapból vesszük az **f** regisztert. A táblázatban megadtuk az utasítások gépi kódját is, és felsoroltuk a STATUS regiszter azon bitjeit, amelyek tartalma módosulhat az utasítás mellékhatásaként.

Mint látni fogjuk, a **PIC18 bitléptető és bitforgató utasításainak nincs közvetlen C nyelvi megfelelője**, ezért csak kiegészítő utasításokkal használhatjuk a C nyelvből ismert aritmetikai vagy logikai eltolás megvalósítására. Ha a forgatás a Carry biten keresztül történik, akkor a megürülő bitre a Carry bit tartalma másolódik be, a kicsorduló bit pedig a Carry bit új tartalma lesz. Az utasítás a Carry biten kívül az **STATUS** regiszter **N** és **Z** bitjeire is hatással van. Az alábbi ábra felül a balra forgató **RLCF**, alatt pedig a jobbra forgató **RRCF** utasítás működését szemlélteti.

Bitforgatás balra, a Carry biten keresztül



Bitforgatás jobbra, a Carry biten keresztül

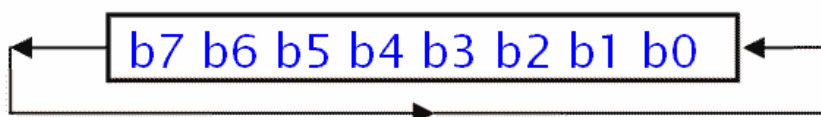


Megjegyzés: A fenti műveletekből akkor lesz a C nyelvi $i < 1$ vagy $i > 1$ kifejezéseknek megfelelő logikai eltolás, ha a bitforgatás előtt a Carry bitet kinullázzuk egy **bcf STATUS,C** utasítással.

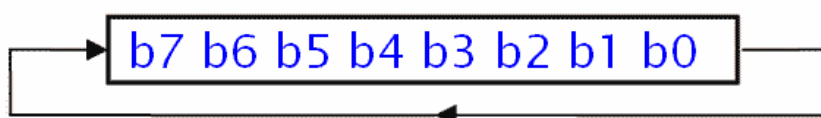
Bitforgatás a Carry bit kihagyásával

A bitforgatás végezhető a Carry bit kiiktatásával is, ekkor a megürülő bitre az éppen kicsorduló bit másolódik be. Ezeknél az utasításoknál a Carry bit értéke nem módosul, az utasítás csupán a **STATUS** regiszter **N** és **Z** bitjeire van hatással. Az alábbi ábra felül a balra forgató **RLNCF**, alatt pedig a jobbra forgató **RRNCF** utasítás működését szemlélteti.

Bitforgatás balra, Carry bit nélkül



Bitforgatás jobbra, Carry bit nélkül



Egyszerű példák a léptetésre

Az alábbi programrészletben láthatjuk, hogy a C programnyelvben használt több-helyiértékes léptetéseket az egy helyiértékkel történő léptetés ismételt végrehajtásával valósíthatjuk meg.

C nyelven

```
unsigned char i,j,k;

i = i >> 2;
j = k << 5;

// további utasítások
```

Assembly nyelven

```
bcf STATUS,C ; C = 0
rrcf i ; i = i >> 1
bcf STATUS,C ; C = 0
rrcf i ; i = i >> 1
movff k,j ; j = k
movlw 5 ; W = ciklusszám
cikl: bcf STATUS,C ; C = 0
rlcf j ; j = j << 1
decf WREG ; W = W-1
bnz cikl ; ugrás, ha W>0
; további utasítások
```

Figyeljük meg, hogy az **i** előjel nélküli 8 bites változókét helyiértékkel történő jobbra léptetését két jobbra történő bitforgatással oldottuk meg. Természetesen mindkettő előtt törölni kell a Carry bitet.

A második esetben elsőször átmásoljuk a **k** változó tartalmát a **j** változóba, majd egy hátultesztlő ciklust szervezünk, melynek törzsében kinullázzuk a **STATUS** regiszter **C** bitjét és egy helyiértékkel balra léptetjük a **j** változó bitjeit. A ciklusokat a **W** munkaregiszterben számláljuk, s akkor fejeződik be a ciklus, amikor a **W** regiszter tartalma nulla lesz a soron következő dekrementálás után.

Aritmetikai kifejezések kiértékelése

Használjuk eddigi ismereteinket egy valamivel összetettebb feladatra, egy kifejezés helyettesítési értékének kiszámítására! Legyen az egyszerűség kedvéért minden változónk előjel nélküli 8 bites egész, s írjunk át assembly utasításokra az **n = j + (i<<3) - k** értékadást!

Célszerű a kiértékelést $i < 3$ kiszámításával kezdeni.

A kifejezés kiértékelésének lépései:

1. Töltsük be az **i** változó értékét a **W** regiszterbe!
2. Léptessük balra 3 helyiértékkel **W** tartalmát!
(háromszor ismételjük:
 - a. töröljük a **Carry** bitet
 - b. balra forgatjuk **WREG** tartalmát
3. Tároljuk el az **n** változóba **W** tartalmát!
4. Töltsük be a **W** munkaregiszterbe **j** tartalmát!
5. Adjuk hozzá **n**-hez a **W** regiszter tartalmát!
6. Töltsük be a **W** munkaregiszterbe **k** tartalmát!
7. Vonjuk ki **n** tartalmából a **W** regiszter tartalmát!

```
;--- n=j+(i<<3)-k;
movf i,W ; W = i
bcf STATUS,0 ; C = 0
rlcf WREG ; W = i<<1
bcf STATUS,0 ; C = 0
rlcf WREG ; W = i<<2
bcf STATUS,0 ; C = 0
rlcf WREG ; W = i<<3
movwf n ; n = W = i<<3
movf j,W ; W = j
addwf n ; n = n+W = j+i<<3
movf k,W ; W = k
subwf n ; n = n-k = j+(i<<3)-k
```

Megjegyzések:

1. Figyeljünk rá, hogy ha memóriarekeszként hivatkozunk a **W** munkaregiszterre, akkor nem **W**-t, hanem **WREG**-et kell írni!
2. A bemutatott programrészlet korántsem optimális, de még nem ismerkedtünk meg a szorzással, ami lerövidíthetné az $i < 3$ kiszámítását.

Mintaprogram: A fenti programrészletet könnyen kiegészíthetjük, hogy egy lefordítható és az MPLAB szimulátorában kipróbálható programot kapjunk. Helyet kell foglalnunk a változóknak (az alábbi példában az Access Bank-ban helyeztük el a változókat). A változók kezdeti értékének beállításával nem foglalkoztunk, ezért a program indításakor kézzel kell beállítani a kívánt adatokat a File Registers ablakban (például $i=3$; $j=100$; $k=30$, mindegyik tízes számrendszerben értendő). De nézhetjük a változókat a Watch ablakban is, ahogy az alábbi ábrán látható.

A megfigyelhető mennyiségek: **W** (ezt **WREG** néven találjuk meg az SFR ablakban), és az **i,j,k,n** változók, a File Registers ablakban. A fentebb említett $i=3$; $j=100$; $k=30$; kezdőértékekkel a végeredmény $0x5E = 94$ lesz.

```
=====
; FIGYELEM! ezt a programot csak a szimulátorban futtassuk,
; mert a CODE 0x000 direktíva miatt felülírná a bootloadert!
=====
#include "p18f14k50.inc"
; Figyelem! A projekthez csatolni kell a PIC18f14k50.1kr állományt is!
    udata_acs ; Adatterület lefoglalása
i    res 1
j    res 1
k    res 1
n    res 1

    CODE 0x000
;=== n=j+(i<<3)-k;
;--- i<<3 kiszámítása
    movf i,W ; W = i
    bcf STATUS,0 ; C = 0
    rlc WREG ; W = i<<1
    bcf STATUS,0 ; C = 0
    rlc WREG ; W = i<<2
    bcf STATUS,0 ; C = 0
    rlc WREG ; W = i<<3
    movwf n ; n = W = i<<3
```

```

;--- j hozzáadása
    movf j,W      ; W = j
    addwf n       ; n = n+W = j+i<<3
;--- k kivonása
    movf k,W      ; W = k
    subwf n       ; n = n-k = j+(i<<3)-k
    goto $
END

```

Az alábbi ábrán a program nyomkövetése látható, ahol éppen befejeződött az $i \ll 3$ kiszámolása és eltárolása.

