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

Progammegszakítások

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

Bevezetés a PIC18 mikrovezérlők assembly programozásába

A fejezet tartalma:

A mikrovezérlők egyetlen, nagysűrűségű integrált (VLSI) áramkörben megvalósított kompakt "számítógépek", szemben a mikroprocesszorokkal, amelyek működéséhez további áramköri elemeket: külső memóriát, periféria áramköröket kell hozzájuk kapcsolni. A mikrovezérlők már tartalmazzák a beépített, nem felejtő programtároló memóriát, az adattároláshoz szükséges RAM memóriát, s számos funkciót el tudnak látni a beépített periféria áramkörökkel (soros kommunikáció, óra, számlálók, ADC, impuzusszélesség modulációs vezérlő, stb.).

A mikrovezérlők általában szerényebb teljesítményűek, mint az általános célú processzorok, s nem tartalmaznak virtuális memória támogatást sem. De a mikrovezérlők és mikroprocesszorok közötti határ az előbbiek rohamos fejlődése miatt elmosódni látszik.

Sok más mikrovezérlőhöz hasonlóan a PIC18 mikrovezérlő család tagjai is Harvard felépítésűek, azaz a programtár és az adattár elkülönül, külön buszon kapcsolódnak a központi egységhez, ami hatékony memóriahasználatot tesz lehetővé. A programtároló memória 16 bites szószélességű, az adatút és az adatregiszterek pedig 8 bites szélességűek. A PICCOLO projektben használt **PIC18F14K50** mikrovezérlő esetében a programtár mérete 16 kilobájt, azaz 8192 utasítás fér bele. Az adattároló statikus RAM memória mérete 768 bájt, melynek utolsó harmada (0x200 - 0x2FF címtartomány) kettős hozzáférésű, az USB periféria pufferterületeként is használható. A ritkábban módosított adatok tárolására további 256 bájt EEPROM memória áll rendelkezésre. Ez az EEPROM memória gyakorlatilag inkább perifériának tekintendő, mivel írásnál/olvasásnál a címzése és a hozzáférés regisztereken keresztül történik.

A központi egység (CPU) órajelének frekvenciája 0-48 MHz lehet. Egy utasítás végrehajtása négy órajelcikust igényel, így a **PIC18F14K50** mikrovezérlő a maximális órajelfrekvencián 12 millió utasítás végrehajtására képes (12 MIPS = 12 Million Instruction Per Second).

Ugyanilyen sebességű a **PIC18F4550** mikrovezérlő is, de a programtároló memóriája kétszer nagyobb (32 kilobájt, azaz 16384 utasítás fér bele), az adattároló memóriája pedig 2048 bájt, melynek második fele (0x400 - 0x7FF címtartomány) kettős hozzáférésű, az USB periféria pufferterületeként is használható.

A PIC18 mikrovezérlő család tagjai gazdag beépített periféria készlettel rendelkeznek. A **PIC18F14K50** mikrovezérlőben például van USB, UART, I2C és SPI soros port, 10 bites felbontású Analóg-Digitális átalakító, 2 db analóg komparátor, egy 8 bites és 3 db 16 bites számláló, input capture és output compare modul, impulzusszélesség moduláció (PWM). Természetesen nem használható egyidejűleg minden periféria, az alacsony lábszám (15 db ki-/bevezetés) korlátozza az egyidejű hozáférhetőséget. Hasonló perifériakészlettel rendelkezik a **PIC18F4550** mikrovezérlő is, de nagyobb lábszámú, összesen 35 ki-/bemenettel rendelkezik.

A programtároló memória szervezése

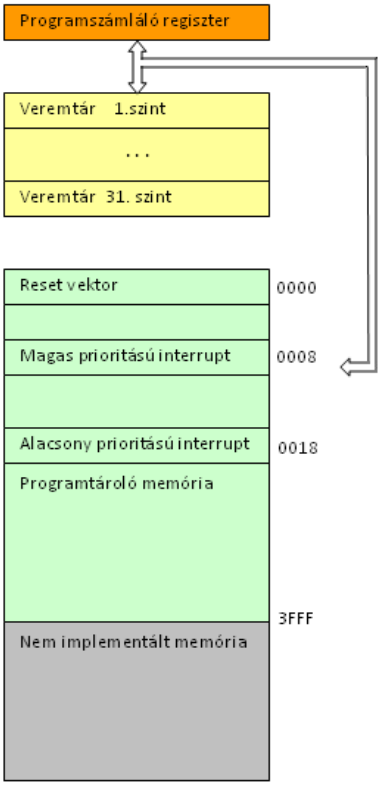
A PIC18 mikrovezérlők a futtatandó programot 16 bites szószélességű, nem felejtő, de elektromosan törölhető és többször újraírható (ún. flash) memóriában tárolják. A memória szervezése azonban olyan, hogy a címezhető egységek 8 bites szavak legyenek, az adatút szélességéhez igazodva, s "little endian" típusú, vagyis az utasításszó alacsonyabb helyiértékű bitjei kerülnek alacsonyabb címre.

A PIC18 mikrovezérlők módosított Harvard felépítésűek: habár elkülönül bennük a programmemória és az

adatmemória, a futó programok mégis tudják olvasni, írni és törölni a programtároló memóriát - a teljes működési tápfeszültség-tartományban. A programmemória olvasása bájtonként történik. Az írása viszont 32 bájtos egységekben, a törlése pedig 64 bájtos blokkokban végezhető. Ömlesztett törlés (bulk erase) programból nem indítható.

A **PIC18F14K50** mikrovezérlő programtároló memóriájának térképe az alábbi ábrán látható (**PIC18F4550** esetén annyi a különbség, hogy az implementált memória címtartományának felső határa 7FFF).

1. ábra: A PIC18F14K50 mikrovezérlő programtároló memóriájának térképe



Az utasításszámláló (PC) az utasítás alacsonyabb helyiértékű felének memória címére mutat. Az utasításszámláló 21 bites, így elvileg 2^21 = 2097152 bájt = 2 megabájt megcímzésére van lehetőség. Tárolása három bájtot igényel, közülük csak a legalacsonyabb helyiértékű bájt (a **PCL** regiszter) írható/olvasható közvetlenül. A magasabb helyiértékű bájt a **PCLATH** regiszteren, a legfelső bitek pedig a **PCLATU** regiszteren keresztül írhatók/olvashatók. A konzisztens műveletek érdekében **PCLATH** és **PCLATU** értéke akkor másolódik át a PC utasításszámláló megfelelő bitjeibe, amikor a **PCL** regiszterbe írunk. Olvasásnál hasonlóképpen történik: A PC magasabb helyiértékű bitjei átmásolódnak a **PCLATH** és **PCLATU** regiszterekbe, amikor a **PCL** regiszter tartalmát olvassuk. A **CALL, RCALL** és **GOTO** utasítások kivételt képeznek: ezek közvetlenül módosítják a programszámláló értékét, ezért ezeknél az utasításoknál nem másolódik át **PCLATH** és **PCLATU** értéke.

Az adattároló memória szervezése

A Harvard felépítésnek megfelelően a PIC18 mikrovezérlők különálló adattároló memóriával (statikus RAM) rendelkeznek, melynek adatútja 8 bites, a címzése pedig 12 bites szélességű. A 12 bites címzés révén a megcímezhető adatterület maximális mérete 4096 bájt lehet. A 4096 bájtos címtartomány 256 bájtos lapokra (bank) van felosztva. Ennek a felosztásnak az a kézenfekvő oka, hogy az egyszavas utasításokban 8 bit áll rendelkezésre az operandus megcímzésére, így tehát 256 bájtnyi adatterület címezhető közvetlenül.

Az adatmemória címtartományának egy része a **Speciális Funkciójú Regiszterek** (SFR) számára van fenntartva. Ide vannak leképezve a mikrovezérlő központi egysége és a perifériák működését vezérlő regiszterek, a CPU és a perifériák állapotát jelző státuszregiszterek és az adatátviteli csatornák adatregiszterei.

A fizikailag létező adatmemória rekeszeit Microchip terminológiával általános célú regisztereknek (General Purpose Registers) nevezik. Ennek az elnevezésnek az az oka, hogy az adatmemória bármely rekeszével közvetlenül végezhetünk logikai, aritmetikai vagy bitműveleteket, ugyanúgy, mintha a CPU regiszterei volnának. Az általános célú regiszterek memóriaterületének mérete a **PIC18F14K50** mikrovezérlő esetében 768 bájt, melynek utolsó harmada (0x200 - 0x2FF címtartomány) kettős hozzáférésű, az USB periféria pufferterületeként is használható. A **PIC18F4550** mikrovezérlő adattároló memóriája 2048 bájt, melynek második fele (0x400 - 0x7FF címtartomány) használható az USB periféria pufferterületeként. A nem implementált memóriaterület olvasása esetén 0 értéket kapunk.

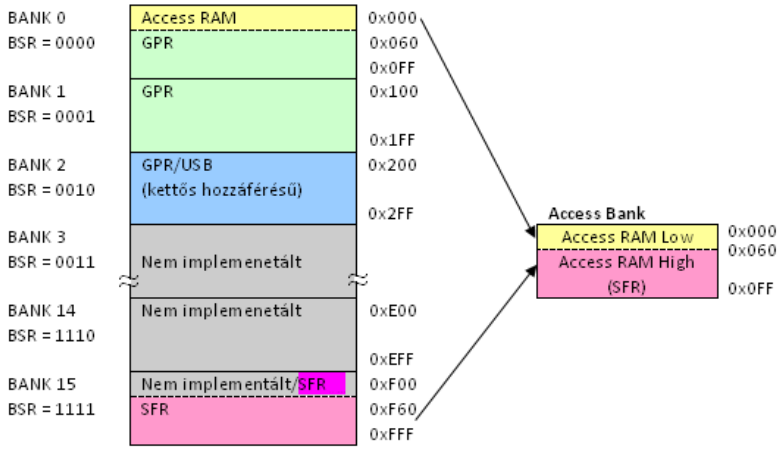
A memórialapok címzése a **BSR** (Bank Select Register) alsó 4 bitjével történik, ez a négy bit egészíti ki az egyszavas utasítások 8 címbitjét az adatmemória megcímzéséhez szükséges 12 bitre. A gyakran használt regiszterek könnyebb elérésének biztosítására szolgál az úgynevezett Access Bank, ami egy olyan memórialapot jelent, amely mindig elérhető egyetlen utasításciklusban, a **BSR** regiszter tartalmától függetlenül. Erre a lapra vannak leképezve a speciális funkciójú regiszterek (0xF60 - 0xFFF) és az általános célú regiszterek első 96 bájtja (0x000 - 0x05F).

Megjegyzések:

1. A **PIC18F14K50** mikrovezérlőnek az a néhány speciális funkciójú regisztere, ami az 0xF53 - 0xF5F címtartományba esik, nincs leképezve az Access Bank nevű virtuális memórialapra.

2. A **PIC18F4550** mikrovezérlő esetében a kettős hozzáférésű memóriatartomány 0x400-tól 0x7FF-ig terjed, s nincs olyan Speciális Funkciójú regiszter, ami ne volna leképezve az Access Bank virtuális memórialapra.

2. ábra: a PIC18F14K50 mikrovezérlő adatmemória-térképe



EEPROM adatmemória

A PIC18 mikrovezérlők a flash programmemória és a statikus RAM adatmemória mellett többnyire EEPROM adatmemóriával is rendelkeznek. Az EEPROM memória nem felejtő, s programból írható/olvasható a teljes működési tápfeszültség tartományban. Az EEPROM memória alapvetően abban különbözik a programtároló flash memóriától, hogy nem blokkonként, hanem bájtonként írható, s újraírhatósági szám (endurance) többnyire egy nagyságrenddel nagyobb (tehát legalább 100 000-1 000 000-szor újraírható). Az EEPROM memória azonban nincs közvetlenül leképezve az adatmemória címtartományába, így csak közvetett módon, a speciális funkciójú regisztereken keresztül írható/olvasható. Az EEPROM memória kezelésével egy későbbi fejezetben fogunk foglalkozni.

A CPU regiszterei

A PIC18 mikrovezérlők számos regisztert tartalmaznak, amelyek a Speciális Funkciójú Regiszterek címtartományában érhetők el. Ezek jelentős hányada a perifériák működésével kapcsolatos, azonban a 0xFD8 és 0xFFF közötti címeken található regiszterek a központi egységhez (CPU) tartoznak. Az alábbi táblázatban röviden összefoglaltuk ezen regiszterek címét, nevét és funkciójuk rövid megnezevését. A legfontosabb regisztereket kiemeléssel megjelöltük.

1. táblázat: A PIC18 központi egységének regiszterei

Cím	Név	Leírás
0xFFF	TOSU	Top of stack upper (Verem teteje: felső bitek)
0xFFE	TOSH	Top of stack high (Verem teteje: magas bájt)
0xFFD	TOSL	Top of stack low (Verem tetetje: alsó bájt)
0xFFC	**STKPTR**	**Stack pointer (Veremmutató)**
0xFFB	PCLATU	Upper program counter latch (Programszámláló:felső bitek)
0xFFA	PCLATH	High program counter latch (Programszámláló:magas bájt)
0xFF9	PCL	Program counter low byte (Programszámláló:alsó bájt)
0xFF8	TBLPTRU	Table pointer upper (Táblamutató: felső bitek)
0xFF7	TBLPTRH	Table pointer high (Táblamutató: magas bájt)
0xFF6	TBLPTRL	Table pointer low (Táblamutató: alsó bájt)
0xFF5	TABLAT	Table latch (Tábla adatretesz)
0xFF4	PRODH	High product register (Szorzat: magas bájt)
0xFF3	PRODL	Low product register (Szorzat: alsó bájt)
0xFF2	INTCON	Interrupt control register (Programmegszakítás vezérlő regiszter)
0xFF1	INTCON2	Interrupt control register 2 (Programmegszakítás vezérlő regiszter)
0xFF0	INTCON3	Interrupt control register 3 (Programmegszakítás vezérlő regiszter)
0xFEF	INDF0(1)	Indirect file register pointer 0 (Indirekt címzés)
0xFEE	POSTINC0(1)	Post increment pointer 0 (Indirekt címzés utóinkrementálással)
0xFED	POSTDEC0(1)	Post decrement pointer 0 (Indirekt címzés utódekrementálással)
0xFEC	PREINC0(1)	Preincrement pointer 0 (Indirekt címzés előinkrementálással)
0xFEB	PLUSW0(1)	Add WREG to FSR0 (Indirekt címzés: FSR0+WREG)
0xFEA	FSR0H	File select register 0 high byte (FSR0 címmutató magas bájtja)

0xFE9	FSR0L	File select register 0 low byte (FSR0 címmutató alacsony bájtja)
0xFE8	**WREG**	**Working register (munkaregiszter)**
0xFE7	INDF1(1)	Indirect file register pointer 1 (Indirekt címzés)
0xFE6	POSTINC1(1)	Post increment pointer 1 (Indirekt címzés utóinkrementálással)
0xFE5	POSTDEC1(1)	Post decrement pointer 1 (Indirekt címzés utódekrementálással)
0xFE4	PREINC1(1)	Preincrement pointer 1 (Indirekt címzés előinkrementálással)
0xFE3	PLUSW1(1)	Add WREG to FSR1 (Indirekt címzés: FSR1+WREG)
0xFE2	FSR1H	File select register 1 high (FSR1 címmutató magas bájtja)
0xFE1	FSR1L	File select register 1 low (FSR1 címmutató alacsony bájtja)
0xFE0	**BSR**	**Bank select register (Memórialap-választó regiszter)**
0xFDF	INDF2(1)	Indirect file register pointer 2 (Indirekt címzés)
0xFDE	POSTINC2(1)	Post increment pointer 2 (indirekt címzés utóinkrementálással)
0xFDD	POSTDEC2(1)	Post decrement pointer 2 (indirekt címzés utódekrementálással)
0xFDC	PREINC2(1)	Preincrement pointer 2 (indirekt címzés előinkrementálással)
0xFDB	PLUSW2(1)	Add WREG to FSR2 (Indirekt címzés: FSR1+WREG)
0xFDA	FSR2H	File select register 2 high byte (FSR2 címmutató magas bájtja)
0xFD9	FSR2L	File select register 2 low byte (FSR2 címmutató magas bájtja)
0xFD8	**STATUS**	**Status register (Aritmetikai és Logikai Egység állapotjelzője)**

(1) Nem fizikai regiszter

A **PCL, PCLATH, PCLATU** regiszerekről már volt szó a fentiekben, ezeken keresztül írható/olvasható a 21 bites programszámláló. **STKPTR** a hardveres veremtár mutatója, **TABLAT** és a **TBLPTRx** regiszterek a programmemória elérésére szolgálnak. **PRODL** és **PRODH** a hardveres szorzóegység eredményét tároló regiszterek. Az **INTCONx** regiszterek bitjei a perifériák programmegszakítási kérelmeit engedélyezik vagy tiltják. Az **FSR0, FSR1** és **FSR2** mutatók az adatmemória indirekt címzésére szolgálnak. Az **INDF, POSTINC, POSTDEC, PREINC, PLUSW** címek viszot nem fizikai regisztereket takarnak, hanem ezen címek használatakor a hozzájuk tartozó FSRx mutatóval megcímzett adatbájtokkal történik a műveletvégzés. Ennek részleteiről majd az **Indirekt címzés** alfejezetben lesz szó részletesebben. **WREG** a munkaregiszter, **BSR** a memórialap-választó regiszter (a 12 bites adatmemória cím felső 4 bitje), **STATUS** pedig az Aritmetikai és Logikai Egység (ALU) állapotjelző bitjeit tartalmazza.

Adatmozgató utasítások

Az adatmozgató utasítások feladata az, hogy átmásolják az adatot a forrás helyéről a célhelyre. Az 'adatmozgatás' elnevezés, vagy az emlékeztető kódja (MOVE) megtévesztő elnevezés, mert arra utal, mintha az adatot elvennénk a forráshelyéről és átraknánk a célhelyre. Valójában másolásról van szó, azaz a forrás helyén érintetlenül megmarad az adat. A forrás jelölésére az alábbiakban az angol 'source' elnevezés rövidítését használjuk: src, vagy csak s. A célhely jelölése pedig, ugyancsak az angol 'destination' elnevezés alapján: dst, vagy röviden d. Szimbolikus jelöléssel így is írhatjuk az adatmozgatást:

(src) → dst ahol '()' jelentése = "tartalma" vmi-nek.

Ez a művelet **két operandust** használ (a forrás és a célhely címét). Az egyszavas utasításokban azonban csak egy memóriacím megadására van lehetőség. A másik cím ezért vagy egy kitüntetett regiszter (a **WREG** munkaregiszter), vagy a forrás és a cél is ugyanaz a memóriarekesz.

Megjegyzés: Bár értelmetlenségnek látszik az olyan "adatmozgatás", ahol a forrás és a cél címe ugyanaz, mellékhatásai miatt mégsem haszontalan, mivel beállítja a **STATUS** regiszter **Z** és **N** állapotjelző bitjeit. Ezek segítségével megállapítható, hogy a "helybenjárással mozgatott" adatbájt nulla-e (ha Z=1), vagy negatív előjelű (ha N=1).

Tágabb értelemben az adatmozgató utasításokhoz soroljuk azokat az utasításokat is, ahol a forrásadat magában az utasításkódban szerepel (azonnali címzés), vagy ha egy regiszteren belül történik adatmozgatás. Ezeket az utasításokat is figyelembevéve, az alábbi adatmogató utasításokkal fogunk megismerkedni:

2. táblázat: A PIC18 mikrovezérlők adatmozgató utasításai

Emlékeztető kód	Rövid leírás
lfsr f, k	FSRx feltöltése konstanssal: k → FSRx
movf f, d, a	Adatmozgatás f címről: (f) → f vagy (f) → WREG
movff fs, fd	Adatmozgatás fs címről fd címre: (fs) → fd
movwf, f,a	WREG to f: (WREG) → f
movlb k	BSR<3:0> feltöltése konstanssal: k → BSR<3:0>
movlw k	WREG feltöltése konstanssal: k → WREG
swapf f, d, a	f félbájtjainak felcserélése

Megjegyzés: az **lfsr f, k** valamint a **movff fs, fd** utasítások kétszavasak (32 bit)

A MOVF utasítás - memóriarekesz tartalmának másolása

Az utasítás általános formája:

movf f, d, a

3. táblázat: a MOVF utasítás gépi kódja

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
0	1	0	1	0	0	d	a	f	f	f	f	f	f	f	f

Az utasításban **f** a forráscím. Az **f** rövidítés a memóriarekeszek "file registers" elnevezéséből ered. Az egybites **d** kapcsoló a célhelyet jelöli ki: d=0 esetén az adat a **WREG** munkeregiszterbe 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 vessszük figyelembe **BSR** tartalmát, a=1 esetén pedig a **BSR** által kijelölt memórialapot használjuk. Olvashatóbb lesz a programunk, ha **f, d** és **a** helyére nem számmal adjuk meg, hanem előre definiált szimbólumokat használunk. Például d=0 esetén **d** helyére a **W** szimbólumot írjuk, d=1 esetén pedig **F**-et. Hasonlóan, **a** helyére **ACCESS** írható 0 helyett és **BANKED** írható 1 helyett.

Az access bit alapértelmezett értéke

Ha az **a** módosítót elhagyjuk az utasításból (nem adjuk meg), akkor **a** alapértelmezett értéke az alábbi szabályok szerint alakul:

Ha **f** a 0x00-0x5F vagy a 0xF60-0xFFF tartományba esik, akkor a=0 (**ACCESS**) az alapértelmezett érték.

Ha pedig **f** a 0x060-0xF5F tartományba esik, akkor a=1 (**BANKED**) az alapértelmezett érték.

4. táblázat: Néhány movf utasítás gépi kódja alapértelmezett ACCESS bit használatával.

Gépi kód	Emlékeztető kód	Megjegyzés
0x5070	movf 0x070,w	(0x070) → WREG, ACCESS (a=0)
0x5170	movf 0x170,w	(0x70) → WREG, BANKED (a=1)
0x5170	movf 0x270,w	(0x70) → WREG, BANKED (a=1)
0x5090	movf 0xF90,w	(0xF90) → WREG, ACCESS (a=0)

Megjegyzés: Az első és a negyedik utasítás gépi kódja abszolút címet jelöl ki az adatmemóriában. A második és a harmadik gépi utasítás relatív címeket jelöl ki, a **BSR** által kiválasztott aktuális memórialapon. A 0x170 és a 0x270 cím tényleges eléréséhez tehát az is szükséges, hogy az adatmozgatás előtt **BSR** alsó bitjeit a megfelelő lapcímmel feltöltsük, a **MOVLB** utasítás segítségével.

A MOVLB utasítás - a BSR regiszter beállítása

Figyeljük meg, hogy a 4. táblázatban szereplő movf 0x170,w és movf 0x270,f utasításoknak ugyanaz a gépi kódja, annak ellenére, hogy az emlékeztető (mnemonikus) kódjuk különböző. Ez azért van, mert az utasítás gépi kódjában a címnek csak az alsó 8 bitje kap helyet, melynek értéke mindkét esetben 0x70. Ahhoz, hogy az eredeti szándéknak megfelelően működjön a **movf 0x170,w** utasítás, a **BSR** regiszternek alsó négy bitjének 1-et kell tartalmaznia, hogy az 1-es memórialapot (Bank 1) válassza ki. Hasonló módon, **BSR<3:0>** tartalmának 2-nek kell lennie a **movf 0x270,f** utasítás megfelelő működéséhez.

A **movlb** utasítást használhatjuk arra, hogy a **BSR** regiszter tartalmát beállítsuk. Az utasítás formája:

movlb k jelentése: k → BSR<3:0>

Az adatmozgató utasításokban felbukkanó **l** betű a 'literális' elnevezés rövidítése, a **movlb** emlékeztető kód pedig az angol "move literal to bank (register)" megnevezést takarja, melyet így fordíthatunk magyarra: mozgasd a literális értéket a lapváltó regiszterbe. Az utasítás operandusa egy **k** konstans, melynek értéke 0..15 közötti szám lehet. Természetesen ezek közül csak annak van értelme, amelyhez ténylegesen tartozik fizikai memória (**PIC18F14K50** esetében 0,1,2 és 15, **PIC18F4550** esetében pedig 0, 1, 2,...,7, valamint 15).

5. táblázat: a MOVLB k utasítás gépi kódja

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	0	0	0	1	0	0	0	0	k	k	k	k

Az alábbi kódrészlet bemutatja a **movlb** és a **movf** utasítások használatát a 0x270 címen elhelyezkedő memóriarekesz eléréséhez, melynek tartalmát a **WREG** munkaregiszterbe másoljuk.

```
Gépi kód    Emlékeztető kód   Megjegyzés
------------------------------------------------------
0x0102      movlb 2           ;2 → BSR<3:0>
0x5170      movf 0x270,w      ;(0x270) → WREG, BANKED
```

Megjegyzés: a pontosvesszővel kezdődő szöveg csupán megjegyzés, nem tartozik a kódhoz!

Természetesen nem szükségesl minden **movf** utasítás előtt beállítani a **BSR** lapválasztó regisztert, de akkor tudnunk kell (nekünk kell fejben tartani), hogy éppen melyik lapra mutat.

A MOVWF utasítás - a munkaregiszter tartalmának másolása

A **movwf** utasítás a WREG regiszter tartalmát másolja az operandusként megadott adatmemória címre. Az utasítás általános formája:

movwf f, a jelentése: (WREG) → f

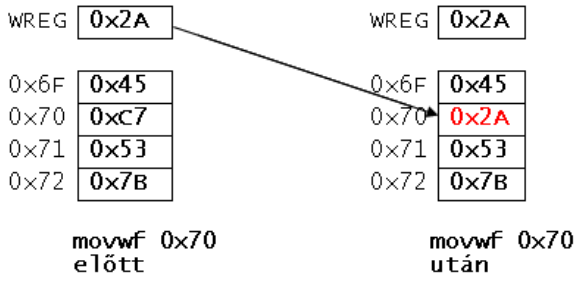
Az **a** (=**ACCESS** vagy **BANKED**) módosító elhagyható, ez esetben az alapértelmezett érték a **movf** utasításnál leírt szabályok szerint lesz meghatározva. Az utasítás gépi kódja az alábbi táblázatban látható.

6. táblázat: a MOVWF f utasítás gépi kódja

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
0	1	1	0	1	1	1	a	f	f	f	f	f	f	f	f

Az alábbi ábrán a **movwf 0x70** utasítás hatását láthatjuk. Képzeljük el, hogy a WREG regiszter és a memóriának egy szelete a baloldali oszlopban látható számokkal van feltöltve. A **movwf 0x70** utasítás hatására csak a 0x70 címen található memóriarekesz tartalma változik meg, ahogy az ábra jobboldali oszlopa mutatja.

3. ábra: A movwf 0x70 utasítás hatásának szemléltetése



A MOVLW utasítás - konstans betöltése a munkaregiszterbe

A fenti példában feltételeztük, hogy a **W** regiszter tartalma 0x2A. De hogy kerül bele az adat? Nos, a 2.táblázatban találunk egy olyan utasítást (**movlw k**), amely pont arra való, hogy egy konstanssal feltölthessük a **W** munkaregisztert. Az utasítás formája, jelentése és gépi kódja az alábbiak szerint néz ki:

movlw k jelentése: k → WREG

7. táblázat: a MOVLW k utasítás gépi kódja

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	1	1	1	0	k	k	k	k	k	k	k	k

Mintapélda: az alábbi kódrészlet 0x5A értéket tölt a munkaregiszterbe, majd az értéket eltároljuk a memória 0x050 című rekeszébe.

```
movlw 0x5A        ; 0x5A → WREG
movwf 0x050       ; WREG → 0x050
```

A MOVFF utasítás - adatmozgatás memórialapok között

Tegyük fel, hogy át akarjuk másolni az adatmemória 0x1A0 címén levő bájtot a 0x23F címre. Az 0x1A0 cím az 1-es sorszámú memórialapon (bank 1) a 0x23F cím pedig a 2-es memórialapon (bank 2) helyezkedik el. Az eddig tanult utasítások felhasználásával így végezhetjük el az adatmozgatást:

```
movlb 0x1         ; bank 1 választása
movf  0x1A0,w     ; (0x1A0)→ WREG
movlb 0x2         ; bank 2 választása
movwf 0x23F       ; WREG → 0x23f
```

Nehezen olvasható, nem túl hatékony kód. De van jobb megoldás: **MOVFF** utasítással ugyanez így írható:

```
movff 0x1A0,0x23F  ; (0x1A0) → 0x23f
```

A **movff fs,fd** utasítás a megadott **fs** forráscím tartalmát átmásolja az **fd** célhelyre. Minkét cím 12 bites, így az utasítás csak két szóban (4 bájt) fér el. A movff utasítás gépi kódja

8. táblázat: a MOVFF fs,fd utasítás gépi kódja

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
1	1	0	0	fs	fs	fs	fs	fs	fs	fs	fs	fs	fs	fs	fs
1	1	1	1	fd	fd	fd	fd	fd	fd	fd	fd	fd	fd	fd	fd

Az alábbi listán láthatunk egy konkrét példát, a **movff 0x1A0,0x23F** utasítás gépi kódját

```
Gépi kód    Emlékeztető kód    Megjegyzés
------------------------------------------------------
0xC1A0      movff 0x1A0,0x23F  ; (0x1A0) → 0x23f
0xF23F
```

Megjegyzés: a **MOVFF** utasításon kívül csak a **GOTO**, **CALL** és az **LFSR** utasítások kétszavasak.

A SWAPF utasítás - a regiszter félbájtjainak felcserélése

A **swapf f** utasítás felcseréli az **f** regiszter alsó és felső 4 bitjét. BCD számábrázolás esetén ez egy kétjegyű szám számjegyeinek felcserélését jelenti. Az utasítás általános formája:

swapf f,d,a jelentése: (f<3:0>) → dest<7:4>; (f<7:4>) → dest<3:0>

Ha d=0, akkor az eredmény a **W** munkaregiszterbe kerül. Ha d=1, az eredmény az **f** regiszterbe íródik vissza (**d** elhagyása esetén ez az alapértelmezett).

Ha a=0, akkor az Access Bank lesz kiválasztva, ha pedig a=1, akkor a **BSR** regiszter által kijelölt memórialapon történik a műveletvégzés.

9. táblázat: a SWAPF f utasítás gépi kódja

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
0	0	1	1	1	0	d	a	f	f	f	f	f	f	f	f

Mintapélda:

```
movlw 0x53
swapf WREG
```

A swapf utasítás végrehajtása után **WREG** tartalma 0x35 lesz.

Az LFSR utasítás - az FSR mutató beállítása

Az **LFSR** utasítás segítségével az indirekt memóriacímzéshez használható **FSR** mutatók valamelyikét tölthetjük fel egy megadott értékkel. Az utasítás formája:

lfsr f,k jelentése: k → FSRf (részletesebben: k<11:8> → FSRfH; k<7:0> → FSRfL)

ahol **f** az **FSRf** mutató sorszáma (f = 0,1,2 lehet), **k** pedig egy 12 bites adatmemória-rekesz cím.

10. táblázat: az LFSR f,k utasítás gépi kódja

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
1	1	1	0	1	1	1	0	0	0	f	f	k	k	k	k
1	1	1	1	0	0	0	0	k	k	k	k	k	k	k	k

Az utasítás első szavában (az 5. és 4. bitben) helyezkedik el a célregisztert kijelölő **f** paraméter, két biten ábrázolva, valamint a **k** konstans legmagasabb helyiértékű 4 bitje. Az utasítás második szavának alsó bájtjában helyezkedik el a **k** konstans maradék 8 bitje.

Mintapélda: Az alábbi utasítás hatására **FRS2H** = 0x01, **FSR2L** = 0xAB lesz.

```
lfsr FSR2,0x1AB
```

Megjegyzés: Sokkal szemléletesebb, ha a célregisztert kijelölő **f** paraméter helyére a fenti példához hasonlóan számkonstans helyett az előre definiált **FSR0, FSR1, FSR2** szimbólumok megfelelőjét írjuk.

Indirekt címzés

Minden mikrovezérlő különféle címzésmódokat használ az operandusok eléréséhez. A PIC18 mikrovezérlők regiszter közvetlen, azonnali, inherens, közvetett és bit-közvetlen címzést használnak. Az előzőekben tárgyalt adatmozgató utasításnál legtöbbször a **regiszter közvetlen címzéssel** találkoztunk, amikor az operandus egy regiszter, amely lehet speciális funkciójú, vagy általános célú. Az utasítás egy 8 bites értékkel adja meg az adatregiszter címét. Ha a megcímzett regiszter az Access Bank-ban helyezkedik el, akkor ez a 8 bit elegendő információ az eléréshez, s a **BSR** lapválasztó regiszter értékét figyelmen kívü hagyjuk. Amennyiben nem az Access Bank-ben levő regisztert címzünk meg, akkor az utasításban megadott 8 bites címet a **BSR** regiszter alsó 4 bitje egészíti ki.

Az azonnali (immediate) címzéssel is találkoztunk már, a **movlw**, **movlb** és az **lfsr** utasítások kapcsán. Ezeknél az utasításoknál a feldolgozandó adatot az utasítás kódja tartalmazza, tehát nincs szükség külön memóriaciklusra az adat elővételéhez, mert az adat az utasítás értelmezésekor már az utasításdekóderben rendelkezésre áll, azonnal felhasználható. Néhány példa:

```
movlw 0x15    ; 0x15 értéket tölt be a W munkaregiszterbe
movlb 2       ; a 2. memórialapot választja ki
lfsr FSR1,0x125  ; FRS1H = 0x01, FSR1L = 0x25 lesz.
```

A fenti példákban megbúvó inherens címzésmód azt jelenti, hogy külön nem adjuk meg az operandus címét, az csupán a műveleti kódból derül ki. A fenti utasításoknak az azonnali címzéssel megadott számkonstanson kívül van egy másik operandusa is, ahová az adatot el akarjuk tárolni. A célhely a **movlw** utasítás esetében a **W** regiszter, a **movlb** utasítás esetén a **BSR** regiszter, az **lfsr** utasítás esetén pedig az **FSRnH** és **FSRnL** regiszterek. Vegyük észre, hogy ezen regiszterek címe (pl. **WREG** címe 0xFE8, **BSR** címe 0xFE0) sehol sem szerepel explicit módon a fenti utasításokban!

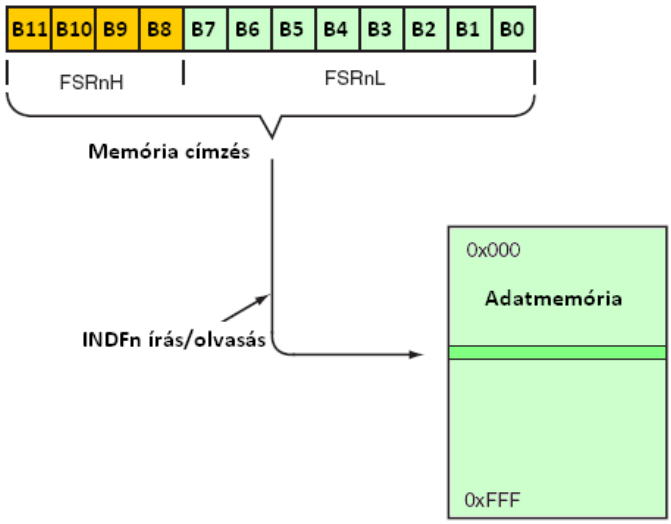
Most pedig az indirekt (közvetett) címzéssel fogunk megismerkedni. Ennél egy speciális funkciójú regiszterpárt használunk mutatóként, amely az adatmemória valamely rekeszét címzi meg. Mivel a követett módon megcímzett regiszter a statikus RAM területen van, a tartalma módosítható a program által (írható, olvasható). Ez a címzésmód nagyon alkalmas táblázatok, tömbök vagy szoftveres veremtárak kezelésére.

A PIC18 mikrovezérlők három ilyen, indirekt címzésre alkalmas mutatóval rendelkeznek: **FSR0**, **FSR1** és **FSR2**. Mivel a teljes adatmemória-címtartomány eléréséhez 12 bitre van szükség, mindegyik mutató két-két regiszterben tárolódik (lásd 1. táblázat):

1. **FSR0: FSR0H**-ből és **FSR0L**-ből áll
2. **FSR1: FSR1H**-ből és **FSR1L**-ből áll
3. **FSR2: FSR2H**-ből és **FSR2L**-ből áll

Miután valamelyik **FSRx** regisztert beállítottuk (pl. az **lsfsr** utasítás segítségével), írni vagy olvasni kell az adott **FSR** regiszterhez tartozó viruális (fizikailag nem létező) regisztert (tehát **INDF0**, **INDF1**, **INDF2** valamelyikét).

Ha például egy utasítás az **INDF0** címreír, akkor az írás ténylegesen az **FSR0H:FSR0L** regiszterpárok által megcímzett memóriarekeszbe történik. Hasonlóan, ha pl. az **INDF2** címről olvasunk, akkor valójában az **FSR2H:FSR2L** regiszterpárral megcímzett memóriarekesz tartalmát olvassuk. Az indirekt címzés folyamatát az alábbi ábrán szemléltetjük:



4. ábra: Az indirekt címzés szemléltetése

Az indirekt címzés használatának hatékonyságát növeli, hogy a mutató adatátvitel előtt vagy után automatikusan növelhető, adatátvitel után automatikusan csökkenthető, vagy az adatátvitelnél a **W** munkaregiszter tartalma eltolási címként figyelembe vehető. Ezek használata az **FSR** regiszterekhez tartozó további speciális címek segítségével történik. Az 1. táblázatban láthatjuk, hogy összesen öt speciális, fizikailag nem realizált regiszter címe tartozik. Ezek használatakor az **FSR** mutató értéke az alábbiak szerint alakul:

1. **INDFn** (n = 0 . . . 2) megcímzésekor az adatátvitel az **FSRn** regiszterpár által megcímzett memóriarekeszben történik. Az adatátvitel során **FSRn** tartalma nem változik.
2. **POSTDECn** (n = 0 . . . 2) megcímzésekor az adatátvitel az **FSRn** regiszterpár által megcímzett memóriarekeszben történik. Az adatátvitel *után* **FSRn** tartalma eggyel csökken (dekrementálódik).
3. **POSTINCn** (n = 0 . . . 2) megcímzésekor az adatátvitel az **FSRn** regiszterpár által megcímzett memóriarekeszben történik. Az adatátvitel *után* **FSRn** tartalma eggyel növekszik (inkrementálódik).
4. **PREINCn** *előtt* **FSRn** tartalma eggyel növekszik (inkrementálódik). Az adatátvitel tehát már az **FSRn** regiszterpár megnövelt tartalmával megcímzett memóriarekeszben történik.
5. **PLUSWn** (n = 0 . . . 2) megcímzésekor az adatátvitel a **WREG** előjeles tartalmának, mint eltolási címnek figyelembevételével történik (**WREG** tartalma előjelesen hozzáadódik az **FSRn**-ben tárolt címhez). Az adatátvitel nem módosítja **FSRn** tartalmát.

Az alábbi példák a közvetett (indirekt) címzésmód használatát mutatják be:

movwf INDF0

> A **WREG** tartalmát az **FSR0** regiszterpárral (**FSR0H:FSR0L**) megcímzett memóriarekeszbe másolja. Az utasítás végrehajtása során **FSR0** tartalma nem változik meg.

movwf POSTDEC0

A **WREG** tartalmát az **FSR0** regiszterpárral (**FSR0H:FSR0L**) megcímzett memóriarekeszbe másolja. Az adatmozgatás után **FSR0** tartalma eggyel csökken, így egy esetleges következő közvetett címzésnél már más memórirekesz lesz megcímezve.

movwf PREINC0

Az utasítás először eggyel megnöveli **FSR0** tartalmát, majd **WREG** tartalmát az **FSR0** regiszterpárral (**FSR0H:FSR0L**) megcímzett memóriarekeszbe másolja.

clrf PLUSW0

Törli azt a memóriarekeszt, melynek címe a **WREG** regiszter és **FSR0** regiszterpár tartalmának összegével egyenlő (0 → FSR0+WREG). Az utasítás során sem **WREG** sem **FSR0** tartalma nem változik.

A fenti példákban természetesen nem kell megadni, hogy a célhely az Access Bankban van-e, mivel az indirekt címzéshez komplett, 12 bites regisztereket használunk (az **FSR0...FSR2** regiszterpárok valamelyikét).

Matematikai utasítások: összeadás és kivonás

Minden mikrovezérlő képes a legalapvetőbb matematikai műveletek végzésére: összeadás, kivonás, inkrementálás (+1) és dekrementálás (-1). Egy későbbi fejezetben látni fogjuk, hogy a PIC18 mikrovezérlők a fentiek mellett hardveres szorzásra is képesek: két 8 bites, előjel nélküli szám összeszorzását egyetlen utasításciklus alatt elvégzik. Most természetesen az elemi műveletekkel kezdjük az ismerkedést.

Összeadás

Az összeadó utasítások általános neve ADD, s két vagy három operandusuk lehet. Ahogy a konkrét utasítások emlékeztető kódja is jelzi, az összeadás egyik operandusa mindig a W munkaregiszter. A második operandus egy megnevezett memóriacím lehet, vagy egy számkonstans (literális érték). A harmadik operandus a STATUS regiszter C (carry = átvitel) bitje lehet, melynek figyelembevételére a következő fejezetben tárgyalt több-bájtos összeadásoknál lesz szükségünk. A PIC18 mikrovezérlők utasításkészletében az alábbi összeadó utasításokkal találkozunk:

11. táblázat: A PIC18 mikrovezérlők összeadó utasításai

Emlékeztető kód	Rövid leírás
addlw k	számkonstans hozzáadása WREG-hez
addwf f, d, a	WREG és f tartalmának összeadása
addwfc f, d, a	WREG, f és C tartalmának összeadása

Az ADDLW utasítás

Az utasítás formája: **addlw k** jelentése: (W) + k → W
A **W** munkaregiszter tartalmához hozzáadja a 8 bites (k=0..255) **k** konstans értékét.
Az utasítás gép kódja: 0000 1111 kkkk kkkk

Mintapélda: Az alábbi utasítások hatására **WREG** tartalma 0x10-ről 0x25-re változik.

```
movlw 0x10      ; WREG = 0x10
addlw 0x15      ; WREG új értéke 0x10 + 0x15 = 0x25 lesz
```

Az ADDWF utasítás

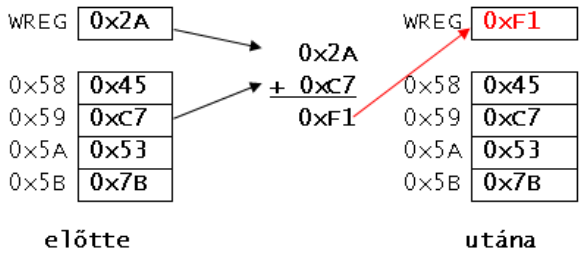
Az utasítás formája: **addwf f,d,a** jelentése: (W) + (f) → cél

A **W** munkaregiszter tartalmához hozzáadja az **f** memóriarekesz tartalmát. Ha d=0, akkor az eredmény a **W** munkaregiszterbe kerül. Ha pedig d=1, akkor az eredmény az **f** regiszterbe íródik vissza (**d** elhagyása esetén ez az alapértelmezett). Ha a=0, akkor az Access Bank lesz kiválasztva, ha pedig a=1, akkor a **BSR** regiszter által kijelölt memórialapon történik a műveletvégzés.

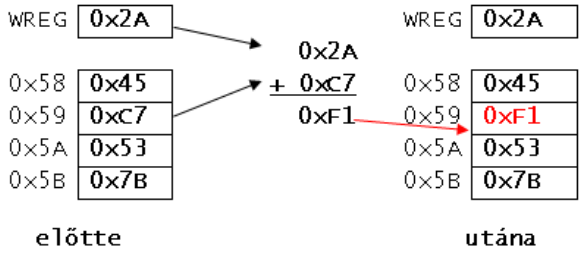
Az utasítás gép kódja: 0010 01da ffff ffff

Az alábbi mintapéldában figyeljük meg jól a **d** módosító szerepét!

a. Az addwf 0x59,0 utasítás hatása



b. Az addwf 0x59,1 utasítás hatása



5. ábra: Mintapélda az addwf utasítás hatásának szemléltetésére

Megjegyzések:

1. A fenti példában nem volt szükség az 'a' címmódosító bit megadására, mert az 0x59-es című memóriarekesz alapértelmezett módon is elérhető.
2. A fenti példában numerikusan adtuk meg a d módosító érékét, ami rontja a program olvashatóságát. Helyesebb lett volna az előredefiniált W és F szimbólumok használata: **addwf 0x59,W** és **addwf 0x59,F**.

Az **ADDWFC** utasítás használatával majd a következő fejezetben ismerkedünk meg.

Kivonás

A kivonást végző utasítások általános neve SUB, s két vagy három operandusuk lehet. Ahogy a konkrét utasítások emlékeztető kódja is jelzi, az összeadás egyik operandusa mindig a W munkaregiszter. A második operandus egy megnevezett memóriacím lehet, vagy egy számkonstans (literális érték). A harmadik operandus a STATUS regiszter C bitje lehet, melynek figyelembevételére a következő fejezetben tárgyalt több-bájtos műveleteknél lesz szükségünk (több-bájtos kivonásnál a C státuszbit logikai negáltja a borrow=áthozat bit). A PIC18 mikrovezérlők utasításkészletében az alábbi kivonást végző utasításokkal találkozunk:

12. táblázat: A PIC18 mikrovezérlők kivonást végző utasításai

Emlékeztető kód	Rövid leírás
sublw k	WREG kivonása a k számkonstansból
subwf f, d, a	WREG kivonása az f regiszterből
subwfb f, d, a	WREG és az áthozat kivonása az f regiszterből
subfwb f, d, a	f regiszter és az áthozat kivonása WREG-ből

A SUBLW utasítás

utasítás formája: **sublw k** jelentése: k - (W) → W

A 8 bites (k=0..255) **k** konstans értékéből kivonja a **W** munkaregiszter tartalmát, s az eredmény a **W** regiszterbe kerül.

Az utasítás gép kódja: 0000 1000 kkkk kkkk

Mintapélda: Az alábbi utasítások hatására **WREG** tartalma 0x10-ről 0x15-re változik.

```
movlw 0x10      ; WREG = 0x10
sublw 0x25      ; WREG új értéke 0x25 - 0x10 = 0x15 lesz
```

A SUBWF utasítás

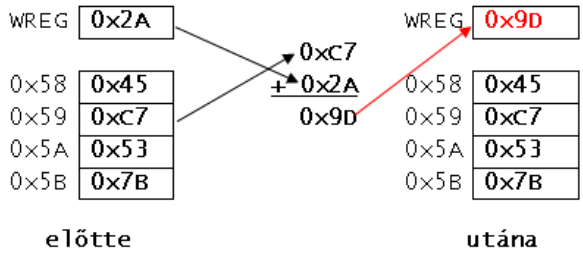
Az utasítás formája: **subwf f,d,a** jelentése: (f) - (W) → cél

Az **f** memóriarekesz tartalmából kivonja a **W** munkaregiszter tartalmát. Ha d=0, akkor az eredmény a **W** munkaregiszterbe kerül. Ha pedig d=1, akkor az eredmény az **f** regiszterbe íródik vissza (**d** elhagyása esetén ez az alapértelmezett). Ha a=0, akkor az Access Bank lesz kiválasztva, ha pedig a=1, akkor a **BSR** regiszter által kijelölt memórialapon történik a műveletvégzés.

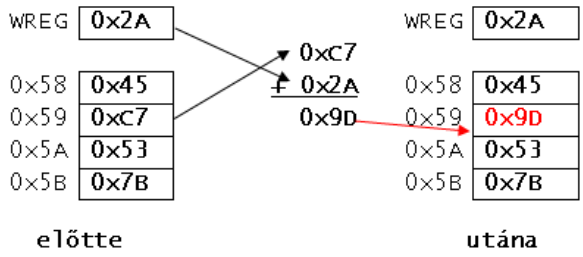
Az utasítás gép kódja: 0101 11da ffff ffff

Az alábbi mintapéldában figyeljük meg jól a **d** módosító szerepét! Most az előre definiált **W** és **F** szimbólumok segítségével adjuk meg d értékét.

a. A `subwf 0x59,W` utasítás hatása



b. A `subwf 0x59,F` utasítás hatása



6. ábra: Mintapélda a subwf utasítás hatásának szemléltetésére

Megjegyzés: A fenti példában nem volt szükség az 'a' címmódosító bit megadására, mert az 0x59-es című memóriarekesz alapértelmezett módon is elérhető.

A **SUBWFB** és **SUBFWB** utasítások használatával majd a következő fejezetben ismerkedünk meg.

Inkrementáló utasítás

A matematikai utasítások speciális esetei az egyoperandusos inkrementáló és dekrementáló utasítások,amelyek eggyel növelik vagy csökkentik a megadott címen található regiszter tartalmát.

Az inkrementáló utasítás formája: **incf f,d,a** jelentése: (f) + 1 → cél

Az **f** memóriarekesz tartalmát megnöveli eggyel. Ha d=0, akkor az eredmény a **W** munkaregiszterbe kerül. Ha pedig d=1, akkor az eredmény az **f** regiszterbe íródik vissza (**d** elhagyása esetén ez az alapértelmezett). Ha a=0, akkor az Access Bank lesz kiválasztva, ha pedig a=1, akkor a **BSR** regiszter által kijelölt memórialapon történik a műveletvégzés.

Az utasítás gép kódja: 0010 10da ffff ffff

Mintapélda:

```
incf 0x059, f   ; 0x059 ←(0x059) +1
incf 0x059, w   ; w ←(0x059) + 1
```

Dekrementáló utasítás

Eggyel csökkenti a forrásként szolgáló operandus tartalmát, és eltárolja a célként kijelölt regiszterbe. Az utasítás alakja és jelentése:

decf f,d,a jelentése: (f) - 1 → cél

Az **f** memóriarekesz tartalmát eggyel csökkenti. A célhelyet a **d** bit választja ki: ha d=0, akkor az eredmény a **W** munkaregiszterbe kerül. Ha pedig d=1, akkor az eredmény az **f** regiszterbe íródik vissza (**d** elhagyása esetén ez az alapértelmezett). Ha a=0, akkor az Access Bank lesz kiválasztva, ha pedig a=1, akkor a **BSR** regiszter által kijelölt memórialapon történik a műveletvégzés.

Az utasítás gép kódja: 0000 01da ffff ffff

Mintapélda:

```
decf 0x056, f   ; 0x059 ←(0x056) - 1
decf 0x056, w   ; w ←(0x056) - 1
```

A GOTO utasítás

A program menetét befolyásoló utasítások közül az egyik legegyszerűbb a **GOTO** utasítás, amellyel beállíthatjuk a PC utasításszámlálót. Amikor a mikrovezérlőt bekapcsoljuk, vagy újraindítjuk, a programszámláló értéke nullázódik, így a legelső végrehajtandó utasítást a programmemória legelejéről veszi a CPU. Minden utasítás végrehajtásakor a PC programszámláló inkrementálódik, így - ha nem lennének vezérlő utasítások a programban - a mikrovezérlő szigorúan az elhelyezés sorrendjében hajtaná végre az utasításokat. A **GOTO** utasítás ezzel szemben arra utasítja a CPU-t, hogy a programfutás egy megadott címen folytatódjon.

A GOTO utasítás az operandus értékét beírja a programszámlálóba, kijelölve ezzel a következő végrehajtandó utasítás címét. Az operandus értéke az az előjel nélküli 21 bites páros szám, amelyet a fordító a **GOTO** utáni kifejezés kiértékeléséből előállít. Az utasítás alakja:

GOTO Expr lit21 → PC

ahol az Expr kifejezés vagy egy címke, vagy egy kifejezés, amit a futtatható programot összeállító linker program egy 21 bites memóriacímként (aminek páros számnak kell lennie!) értelmezni tud. A lit21 jelölés ezt a 21 bites értéket jelöli.

A párosság követelményét a matematikában szokásos módon jelezve (2k), a GOTO utasítást így is írhatjuk:

GOTO 2k k → PC[20:1]; PC[0] =0

Magyarázat: a programtároló memóriában az utasítások 16 bites szavanként helyezkednek el, a memória azonban bájtonként címezhető. A **GOTO Expr** utasításban az Expr kifejezés és a **PC** utasításszámláló is bájtonkénti címzést használ. Mivel az utasítások mindig páros címen kezdődnek, mind **PC**, mind Expr tartalma páros szám kell, hogy legyen (a legalsó helyiértékű bit nulla). A **GOTO** utasítás ezért a legalsó bitet (PC[0]) nem is tárolja, hanem csak a PC[20:1]-be írandó utasításszó-címet, melyet úgy kapunk, hogy az Expr értékét kettővel osztjuk (egészosztás!).

A **GOTO** utasítás - a benne tárolt hosszú cím miatt - csak két utasításszóban fér el: az első szóban az utasításkód mellett az utasításszó-cím alsó 8 bitje helyezkedik el. A magasabb helyiértékű maradék 12 bit a következő utasításszó alsó 12 bitjében helyezkedik el.

13. táblázat: A GOTO utasítás assembly és gépi alakja, s egy konkrét példa

Assembly utasítás	Gépi kód
GOTO 2k	1111 1110 kkkk kkkk 1111 kkkk kkkk kkkk
GOTO 0x0208	0xFE04; 0xF001

A GOTO utasítás két utasításszóban tárolódik, a végrehajtása két utasításciklust vesz igénybe és feltétel nélküli ugrást eredményez.

Mintapélda: A mikrovezérlő programok általában végtelen ciklusban futnak. Az alábbi bugyuta programocska is végtelen ciklusban fut.

```
Main:  incf WREG,W     ; WREG = WREG + 1
       goto Main       ; visszaugrik az előző utasításra
```

A GOTO utasításnál címként a '$' speciális szimbólumot is használhatjuk, ami az aktuális utasítás címét jelenti. Ennek segítségével relatív ugrásokat is megadhatunk. Néhány (értelmetlen) példa:

```
  goto $       ; önmagára ugrik. Ez a legegyszerűbb végtelen ciklus
  goto $-2     ; a GOTO utasítást megelőző utasításra ugrik
  goto $+4     ; a GOTO utasítást követő utasításra ugrik
```

Megjegyzések:

1. Előre ugrásnál ne felegdük, hogy a GOTO utasítás két utasításszót (4 bájtot) foglal le, ezért a rákövetkező utasítás $+4 címen kezdődik.
2. A goto $+4 utasításnak nincs sok értelme, mert a programfutás egyébként is a soronkövetkező utasítással folytatódna.

Egy egyszerű program anatómiája

Írjunk egy egyszerű utasítás-sorozatot! Ha C nyelven írnánk, akkor valahogy így nézne ki:

```
unsigned char i,j,k; // Helyet foglalunk az adatoknak: előjel nélküli 8 bites változók

i = 100;             // i = 100
i = i + 1;           // i++, i = 101
j = i;               // j értéke 101
j = j - 1;           // j--, j értéke 100
k = j + i;           // k = 100 + 101 = 201
```

Természetesen ez nem egy komplett program, értelme sem sok van, de ez most nem fontos. Nézzük meg, hogy hogyan lehet ezeket a C nyelvű parancsokat az eddig tanult assembly utasításokkal megfogalmazni! Előbb azonban tisztáznunk kell, hogy a változók értékei hol legyenek tárolva. Egy logikus válasz erre az, hogy tároljuk ezeket az adatmemória szabad területének (ami a 0x000 címen kezdődik) első néhány bájtján. Ez a választás azért is kézenfekvő, mert így memórialap-váltás nélkül közvetlenül elérhetjük az adatokat. Rendeljük tehát a változókhoz a következő memória címeket:

i címe legyen 0x000
j címe legyen 0x001
k címe legyen 0x002.

A fentiek felhasználásával megírhatjuk az itt pirossal jelölt C utasításoknak megfelelő **assembly utasításokat.** Az alábbi listában a pontosvesszővel kezdődő megjegyzések nem tartoznak a programhoz, csak az áttekintést segítik.

```
; i = 100
movlw 0x64           ; W = 0x64 = 100
movwf 0x000          ; i = 100
```

```
; i = i + 1
incf  0x000           ; i = i + 1

; j = i
movff 0x000,0x001     ; j = i

; j = j - 1
decf  0x001           ; j= j - 1

; k = j + i
movf  0x000,W         ; W = i
addwf 0x001,W         ; W = W+j
movwf 0x002           ; k = W
```

Mint látható, az assembly programrész nem túl olvasmányos, nehezen áttekinthető. Az is látszik, hogy a C-hez képest az assembly nyelv kevésbé hatékony, ugyanazt a műveletet több utasítással tudjuk csak megfogalmazni.
Az assembly utasításokból a gépi kódot elvileg kézzel is elő tudnánk állítani (az egyes utasítások gépi kódjait tartalmazó táblázat alapján), de sokkal kényelmesebb és hatékonyabb ezt az MPLAB fejlesztői környezet Assembler fordító programjának segítségével végezni. Ehhez azonban a fenti utasítás sorozatot további információkkal kell kiegészíteni, hogy a fordítóprogram és a végrehajtható programot összeállító (linker) program számára értelmezhető és egyértelmű legyen a feladat. Magunk számára pedig azzal javítjuk a program áttekinthetőségét, hogy az adattárolásra lefoglalt tárhelyekhez szimbolikus neveket (i, j, k) rendelünk.

```
  #include "p18f14k50.inc"

; Adatterület lefoglalása
  CBLOCK 0x000     ; az adatmemória 0x000 címén kezdődik
  i, j,k           ; helyet foglalunk három egybájtos változónak
  ENDC

  ORG    0         ; Reset vektor a programmemória elején
  goto main        ; interruptnak fentartott hely (0x0-0x1FF) átugrása

  ORG 0x0200       ; főprogram kezdete
main
 ; i = 100
  movlw 0x64       ; W = 0x64 = 100
  movwf i          ; i = 100

; i = i + 1
  incf  i          ; i = i + 1

; j = i
  movff i,j        ; j = i

; j = j - 1
  decf  j          ; j= j - 1

; k = j + i
  movf  i,W        ; W = i
  addwf j,W        ; W = W+j
  movwf k          ; k = W

ciklus
  goto   ciklus    ; végtelen ciklus
  END
```

A programban szereplő, csupa nagybetűvel írt szavak nem a mikrovezérlőnek szánt utasítások, hanem a fordítónak szóló eligazítások, úgynevezett direktívák. Ilyen például az *INCLUDE "p18f14k50.inc"*, amely azt az eszközleíró állományt csatolja be, amely tartalmazza az adott mikrovezérlő regisztereinek és konfigurációs bitjeinek szimbolikus neveit.

A fordító figyelmen kívül hagyja a pontosvesszővel kezdődő jelsorozatokat, egészen az adott sor végéig. A pontosvesszővel kezdődő karaktersorozat tehát csupán megjegyzés (comment), a programra nézve nincs hatása.

A *CBLOCK 0x000* direktíva azt jelzi, hogy az utána következő helyfoglalások az adatmemória területére vonatkoznak. Ha másképp nem rendelkezünk, akkor minden új szimbólum felsorolása egy bájtnyi memóriaterületet foglal le, folytatólagosan. Így most, mivel az első lefoglat hely a 0x000 címen van, az **i** változó címe 0x000, **j** címe 0x001, **k** címe pedig 0x002 lesz. A A *CBLOCK 0x000* direktívával kezdődő adatblokkot *ENDC* zárja le.

Az *ORG 0* direktíva programmemória kezdetét jelzi, ahol a reset vektor és az interrupt vektorok számára fentartott hely kezdődik. Ha betöltőprogramot használunk (bootloader), akkor az annak fenttartott helyet is át kell ugranunk. Most ezekkel nem foglalkozva, önkényesen a 0x200-as címe helyeztük el a főprogram elejét az *ORG 0x0200* direktívával.

Fussuk át röviden a program többi részletét!

```
; i = 100
movlw 0x64            ; W = 0x64 = 100
movwf i               ; i = 100
```

Az első utasítás a W regiszterbe betölti egy számkonstans értékét (0x64 lierális a decimális 100 értéknek felel meg). Itt arra kell ügyelnünk, hogy a számkonstans értéke ne haladja meg a 8 biten történő ábrázolhatóság felső korlátját, a decimális 255 értéket!

A második utasítás a W regiszter tartalmát (tehát a fent említett 100-at) bemásolja az i változóba (az adatmemória 0x000 című bájtjába). Ezzel a két utasítással tehát megtörtént az i = 100 értékadás.

Az i = i +1 értékadás lényegében egy inkrementálással elintézhető:

```
; i = i + 1
incf  i                 ; i = i + 1
```

A j = i értékadásnál nem muszáj a W munkaregiszteren keresztül mozgatni az adatot, közvetlenül is lehet:

```
; j = i
movff i,j               ; j = i
```

Hasonló egyszerűséggel, egyetlen dekrementáló utasítással elvégezhető a j = j -1 művelet is:

```
; j = j - 1
decf  j                 ; j= j - 1
```

Valamivel komplikáltabb eset a k = i + j összeadás. A komplikációt az okozza, hogy az adatok a memóriában vannak, nem a munkaregiszterben, ezért elő kell venni az összeadandókat, illetve a végén el kell tárolni az eredményt.

```
; k = j + i
movf  i,W               ; W = i
addwf j,W               ; W = W+j
movwf k                 ; k = W
```

Először az i változó értékét bemásoljuk a W regiszterbe, majd hozzáadjuk a j változó tartalmát, s a végeredményt átmásoljuk a k változóba. Itt is mindenhol 8 bites műveleteket végzünk!

Megjegyzés: Ügyeljünk rá, hogy a fenti **movf** és **addwf** utasításokból ne hagyjuk ki a d=0 (**W**) címmódosítót, mert az alapértelmezett **F** érték most nemkívánt eredményre vezetne!

A programot egy végtelen ciklussal zárjuk (az utolsó utasítás önmagára ugrik vissza), nehogy "kiszaladjunk" a memóriából. A forráskód legvégét pedig az *END* direktíva zárja.

```
ciklus
    goto    ciklus      ; végtelen ciklus
    END                 ; program vége
```

Megjegyzés: A fenti goto utasítás előtti sorban 'ciklus' egy címke, amely a sor legelején kezdődik és kis-/nagybetű-érzékeny. Az utasítások és a fordítónak szóló direktívák viszont nem kezdődhetnek a sor legelején. A címke értéke az a memóriacím, ahol a soron következő utasítást elhelyezi a fordító/linker.

Ha a program sikeres lefordítása után megnézzük a generált kódot, akkor láthatjuk, hogy a 'ciklus' címkét követő utasítás a 0x0212 címen kezdődik, így a 'ciklus' címke értéke is ez lesz, s a **'goto ciklus'** utasítás a **goto 0x0212** utasítással egyenértékű.

```
                                        12:         ORG 0x0200      ; főprogram kezdete
                                        13:    main
                                        14:
                                        15:    ; i = 100
0200    0E64    MOVLW 0x64              16:         movlw 0x64      ; W = 0x64 = 100
0202    6E00    MOVWF 0, ACCESS         17:         movwf i         ; i = 100
                                        18:
                                        19:    ; i = i + 1
0204    2A00    INCF 0, F, ACCESS       20:         incf  i         ; i = i + 1
                                        21:
                                        22:    ; j = i
0206    C000    MOVFF 0, 0x1            23:         movff i,j       ; j = i
0208    F001    NOP
                                        24:
                                        25:    ; j = j - 1
020A    0601    DECF 0x1, F, ACCESS     26:         decf  j         ; j= j - 1
                                        27:
                                        28:    ; k = j + i
020C    5000    MOVF 0, W, ACCESS       29:         movf  i,W       ; W = i
020E    2401    ADDWF 0x1, W, ACCESS    30:         addwf j,W       ; W = W+j
0210    6E02    MOVWF 0x2, ACCESS       31:         movwf k         ; k = W
                                        32:
                                        33:    ciklus
0212    EF09    GOTO 0x212              34:       goto   ciklus   ; végtelen ciklus
0214    F001    NOP
```

Mennyi idő alatt fut le a programunk?

A program futási idejének kiszámításához két dolgot kell tisztázni:

- Időegységenként hány utasításciklust hajt végre a mikrovezérlőnk?
- Hány utasításciklust vesznek igénybe az egyes utasítások?

A **PIC18F14K50** és **PIC18F4550** mikrovezérlők maximális órajel frekvenciája 48 MHz, de egy utasításciklus négy órajelperiódust igényel. Így a 48 MHz-en futó mikrovezérlő másodpercenként 12 millió utasításciklust hajt végre. Mivel a periódusidő a frekvencia reciproka (T = 1/f), a mikrovezérlőnk egy órajelperiódusa T = 1 /48

000 000 = 20,833.. ns (nanoszekundum, 1 ns = 10^{-9} s). Egy utasításciklus pedig négy órajel periódusig tart, tehát 83,33 ns-ot vesz igénybe.

Az, hogy az egyes utasítások hány utasításciklust vesznek igénybe, kiolvasható az utasításkészletre vonatkozó táblázatokból, amelyek a mikrovezérlő adatlapjában találhatók. A legtöbb utasítás egy utasításciklust vesz igénybe. Azok az utasítások, amelyek megváltoztatják a programszámláló értékét, általában két, vagy három utasításciklust igényelnek. A movff utasítás is két utasításciklust vesz igénybe.

Az említett források felhasználásával számoljuk most össze az **mptst.asm** program utasításainak végrehajtási idejét! Természetesen csak a main és a ciklus címkék közötti utasításokat vesszük figyelembe. A ciklus címke utáni végtelen ciklust még 48 MHz-en is túl lassan hajtja végre a mikrovezérlő... :-)

Az utasítás kódja	utasításciklusok
movlw 0x64	1
movwf i	1
incf i	1
movff i,j	2
decf j	1
movf i,W	1
addwf j,W	1
movwf k	1
Összesen:	**9 utasításciklus**

Tehát 48 MHz-es órajelet feltételezve 9 x 83,33 = 750 ns alatt lefut a fenti program, a milliomod másodperc háromnegyede is elég neki!

Kapcsolat: pic24@esca.atomki.hu