Digitális technika 2.

BMEVIIIAA06

6. előadás

Assembly programozás

(veremkezelés, indirekt címzés és szubrutinok)

Láttuk eddig:

- A CPU csak utasításokat hajt végre
- Minden utasítás egy-egy szám a memóriában
  - » Mi írhatjuk oda, de megjegyezhető formában

Láttunk annyi utasítást, amivel már gyakorlatilag minden programozási feladat megoldható.

Szubrutin

- A programot általában több logikailag elkülönülő részre osztjuk

  → függvény, eljárás, metódus, stb.

  » Ezeket most mind szubrutinnak hívjuk, igazából csak a magas szintű nyelvben van köztük különbség.

- Szubrutin: Önálló, meghatározott funkcióval rendelkező programrészlet, amelynek egy meghatározott belépési pontja (kezdete) van.
- A szubrutinokat „meghívni” szoktuk, a funkciójuk elvégzése után pedig „visszatérnek” a hívásuk utáni utasításra.

További utasítások

Utasítás csoportok:

- Adatmozgató utasítások (ezeket láttuk)
- Aritmetikai műveletek
- Logikai műveletek
- Bitműveletek
- **Vezérlés átadás**
- Egyéb (vezérlő) műveletek

Speciális vezérlésátadás: Szubrutinhívás

- A feltétel nélküli ugráshoz hasonlít, annak megfelelően elvileg 4 fajtája lehetséges:
  - Feltételes vagy feltétel nélküli
  - Abszolút vagy relatív
- A PIC24 csak feltétel nélküli szubrutinhívást támogat
  - » A feltételes hívás egy feltételes ugrással és egy feltétel nélküli hívással kiváltható
    - » *Vannak processzorok, ahol van feltételes hívás is*
- Lényeges különbség: Az ugrásból nem térünk vissza, szubrutinból igen →meg kell jegyezni a visszatérés helyét.

Speciális vezérlésátadás: Szubrutinhívás

- Abszolút hívás:
  - » call címke ;A címkére ugrik, de előbb megjegyzi a *PC+4* címet
  - » call Wn ;"számított hívás" (*függvénypointer*), a Wn*2 címre ugrik
- Relatív hívás:
  - » rcall #konstans ;PC+(2*konstans) címre ugrik, megjegyzi *PC+2* címet
  - » rcall Wn ;"számított relatív hívás", PC+2*Wn-re ugrik.
    - » *Mindkét esetben a konstans, vagy Wn értéke előjeles (negatív irányba is ugorhat)*
- Miért van kettő?
  - » A címke értéke (abszolút program memória cím) lehet 23 bites is, ez nem fér az utasításba, ezért az abszolút hívás (ahogy az abszolút ugrás is) 2 utasítás-helyet, 48 bitet használ el.
  - » Ha csak +-32k db utasítást szeretnénk átugorni, az befér egy relatív utasításba is → spórolunk
  - » Időnként szeretnénk olyan szubrutin-csomagot (függvénykönyvtárat, „library") készíteni, ami több helyen, többször is felhasználható, „relokálható" → erre is jó a relatív ugrás/hívás.
- Mi a laborokon nagyrészt abszolút hívást fogunk használni...

Speciális vezérlésátadás: Visszatérés

- Két lehetőség:
  - return ;visszatér a „megjegyzett” címre
  - retlw #const,wn ;miközben visszatér wn-be berakja adott konstanst

- A függvényeknek általában egy visszatérési értéke van, azt így könnyen megadhatjuk
  - » *(és legalább használjuk valamire a gépi kód „paraméter” részét)*
- Ugyanaz a szubrutin akárhány helyről visszatérhet.
- *Hol jegyezzük meg a visszatérési címet?*

Visszatérési címek tárolása

- A visszatérési cím tárolására több lehetőség
  - Fix helyen (regiszterben)
    - Sok processzor ezt teszi, speciális regiszterbe ment a CALL, innen hoz vissza a RETURN
      - Egyszerű megvalósítás (lényegében PC mozgatása regiszterbe és vissza)
      - Külön meg kell oldani, ha egy szubrutin egy másik szubrutint hívna (már van egy visszatérési cím, azt nem szabadna felülírni)
        - » Hát még ha rekurziót szeretnénk…
  - Fixen beépített speciális tárolóban
    - Ilyen processzor is van, de egyre kevesebb
      - Lényegében több hardver regiszter, mindig a „legfelső”-t olvashatjuk
      - Ha írunk, akkor az eddigi „lejjebb” kerül
        - » Ilyen speciális adatstruktúra: Verem
      - Ha ez beépített hardver, annak korlátozott számú szintje van (2-8-16-32)
        - » Elég hamar elfogy → így sincs rekurzió
  - Verem szervezésben, de az adatmemóriában
    - Ez van a PIC24-ben is (és elég sok egyéb processzorban)

Verem

- Speciális adatstruktúra:
  - First in, last out (FILO): „Amit először beleraktál azt tudod utoljára kivenni”
    (LIFO: Last in First out… pontosan ugyanez másképp.)
      - » Pont ez kell, mert a legelső meghívott szubrutin fog utoljára visszatérni…
- Egy regiszter kell csak hozzá: verem mutató (stack pointer)
  - A mutató mindig a verem „tetejére” mutat
    - A teteje: Ahova épp elemet lehetne rakni (legelső üres hely).
  - Minden új elem berakás után a mutató növekszik
  - Minden elem kivétel előtt a mutató csökken
    - Lehetne pont fordítva is csinálni: a mutató berakás előtt csökken, és épp az utolsó adatra mutat, kivétel után nő
      - » Ilyenkor a stack a memória végéből az eleje felé „lefele” növekszik…
        - » Van ilyen processzor is, sőt olyan is, ahol a stack iránya állítható, de ez nem olyan…

Mutató: Indirekt címzés → ez is csak egy speciális indirekt címzés

Verem

– Minden új elem berakása után a mutató növekszik

```
mov ws, [wd++]
```

- ”indirekt posztinkremens” címzés:
  - „Tedd ws-t a wd által mutatott helyre, majd növeld meg wd-t”
  - Annyival növel, hogy a következő ugyanilyen adatra mutasson (1 ha bájt, 2 ha word)
  - wd csak azután változik meg, miután felhasználtuk

– Minden új elem kivétele előtt a mutató csökken

```
mov [--ws], wd
```

- „indirekt predekremens"
  - „Csökkentsd le ws-t, majd vedd ki ahova most mutat és tedd a wd-be”
  - Ws már azelőtt megváltozik, mielőtt fehasználtuk.

A PIC24 az összes lehetőséget támogatja akár egyszerre is (pl: mov [ws--],[++wd])

Ha akarunk, bármelyik „számozott” regiszterből csinálhatunk így „verem mutatót”.

→ de nem kell

Verem

- Alapértelmezett verem mutató: W15
  - » Láttuk: W15 nem egészen általános célú, mert csak páros számot tartalmazhat (a legalsó bitje fixen 0)
- Speciális verem kezelő „utasítások”:

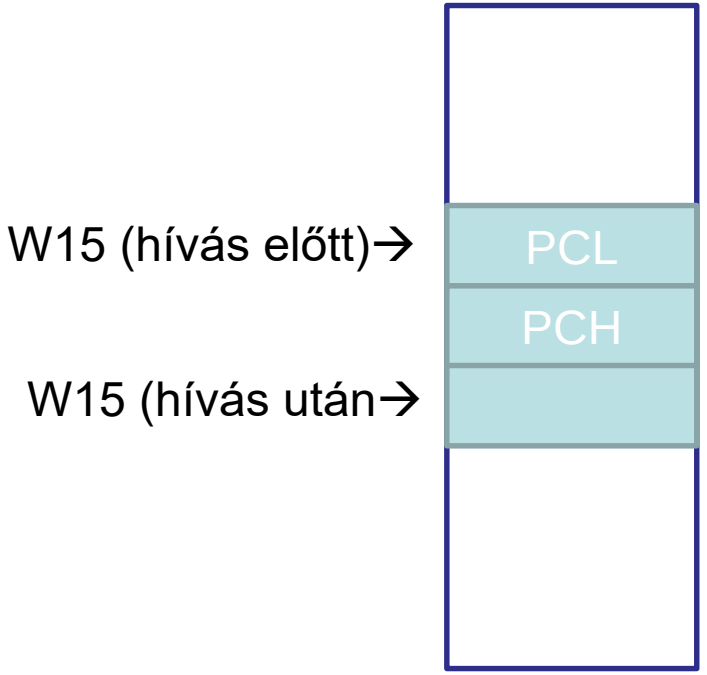
```
push ws          → igazából csak mov ws,[w15++]
pop wd           → igazából csak mov [--w15],wd
push változó
pop változó
```

A szubrutin hívás is w15-öt használja veremként.

Verem és a szubrutinhívás

`call címke`
- push PCL (+?)
- push PCH
- goto címke

Stack (RAM egy részlete):



- A call még helyettesíthető is lenne
  - De így egy utasítással hatékonyabb

`return`
- „pop PCH”
- „pop PCL”
- „goto PCH:PCL”

→ már ettől máshol folytatódna a program

→ilyet nem is lehetne, ha nem lenne speciális utasítás

Szubrutin hívás időigénye

- Láttuk: Minden utasítás egy ciklusig tart, kivéve ha az változtat a programszámlálón
  - » Mert a következő utasítás elolvasása is egy ciklus
- A call ilyen, tehát az pont két ciklus
  - » Közben bőven van idő menteni a RAM-ban lévő stack-be, igaz, hogy egyszerre két word-öt is el kell mentenünk, de pont ennyit tudunk
    - » minden ciklusban egy olvasás és egy írás lehet.
- Mi a helyzet a visszatéréssel:
  - » 1. ciklus, visszatöltjük PCH-t, még mást nem tudunk
  - » 2. ciklus, visszatöltjük PCL-t (most kezdhetünk ugrani, de az még 1 ciklus)
  - » 3. ciklus, elolvassuk azt az utasítást, ahova vissza fogunk térni
  - » … folytatódhat a program → a visszatérés 3 ciklus!, de több kivétel már tényleg nincs ☺

Paraméter átadás szubrutinnak

- Magas szintű programnyelvekben a szubrutinoknak (függvényeknek) szokott lenni paramétere és visszatérési értéke.
- Visszatérési érték általában egy van, azt regiszterben szoktuk visszaadni.
- Paramétereket több lehetőségünk is van átadni:
  - Regiszter(ek)ben
    - » Kevés van belőle, de gyorsabb és egyszerűbb (kis szubrutinoknál jellemző) ✓
  - Memóriában fix helyen
    - » Általában nem használjuk, mert nem lehetne rekurzív vagy reentráns a függvény ✗
      - » *A függvény különböző példányai ugyanazt használnák…*
  - Stack-en
    - » Van elég hely, a különböző példányok különböző értékekkel dolgozhatnak ✓
    - » Speciális indirekt címzéssel (konstans ofszet) mindig elérhetők a paraméterek
      - » Pl. `mov [w15-#6], w0`

Szubrutin általános szerkezete

```
cimke:
   push …
   push …
   mov [w15-?],…
   …
   mov …, w?
   pop …
   pop …
   return
```

- Belépési pont (szubrutin neve címkeként)
- Regiszterek mentése
  - általában nem szeretnénk, hogy a szubrutin elrontsa a regiszterek értékét
  - Ezeket a regisztereket lementhetjük a stack-be (mert ott biztosan van hely nekik)
- Paraméterek kiolvasása

  …
- Visszatérési érték állítása
- Regiszterek visszatöltése
  - Figyelni kell a fordított sorrendre!
- Visszatérés

Írjunk egy szubrutint

- Adjuk össze a w0-ban adott kezdőcímen lévő és w1-ben adott hosszú tömb elemeit (a túlcsordulással nem kell foglalkozzunk), az eredményt tegyük w0-ba.
- Itt most regiszterekben adjuk a paramétereket (kevés van belőlük)

- Hogyan fogjuk ezt használni:

```
.bss
        tomb: .space 100 ;ez egy 50 elemű 16 bites egészekből álló tömb
.text
…
mov #tomb,w0
mov #50,w1
call Tomb_osszead
mov w0, …
```

Írjunk egy szubrutint

- Adjuk össze a w0-ban adott kezdőcímen lévő és w1-ben adott hosszú tömb elemeit (a túlcsordulással nem kell foglalkozzunk), az eredményt adjuk vissza w0-ban.

```
Tomb_osszead:
        push w1                 ;ne rontsuk el w1-et
        push w2                 ;w2-t se
        clr w2
T_o_ciklus:
        add w2, [w0++], w2 ;használhatjuk a most tanult indirekt címzést
        dec w1, w1    ;w1 a tömb hossza, használható ciklusváltozónak
        bra NZ,T_o_ciklus ;amíg el nem fogy, folytassuk
        mov w2, w0    ;eredmény a helyére
        pop w2        ;visszatöltünk fordított sorrendben
        pop w1        ;w0-t nehogy visszatöltsük, azért dolgoztunk, hogy abban legyen
az eredmény!
        return
```

Írjunk egy szubrutint

- Adjuk össze 20 db 16 bites egész paraméter értékét 32 biten.
  - Ennyi regiszter nincs, veremben kapjuk a paramétereket
  - Az eredmény még mindig elfér regiszterben (w0, w1-ben, w1 legyen a nagyobbik helyiérték)

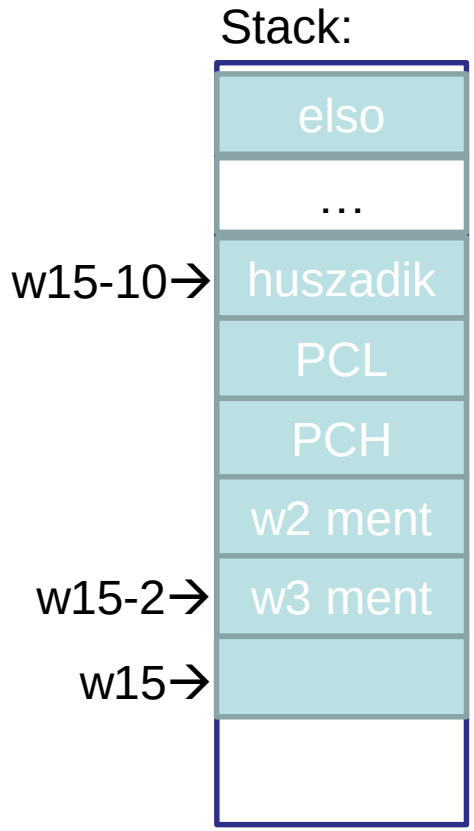
- Hogyan fogjuk ezt használni:

```
mov egyik,w0  ;a paraméterek akárhonnan jöhetnek
push w0
mov masik,w0
push w0
…
mov huszadik,w0
push w0
call Husz_osszead
mov w0, …
mov w1, …
```

Írjunk egy szubrutint

- Adjuk össze 20 db 16 bites egész paraméter értékét 32 biten.
  - Az eredmény még mindig elfér regiszterben (w0, w1-ben, w1 legyen a nagyobbik helyiérték)
  - Gyakorlatilag a veremben már egy tömbben vannak a paramétereink, végig mehetünk rajtuk sorban…

```
Husz_osszead:
        push w2
        push w3
        sub w15, #10, w2 ;most w2 mutat a legutolsó paraméterre
        mov #20,w3
        clr w0
        clr w1
H_o_ciklus:
        add w0,[w2--],w0 ;w2 kettesével fog csökkenni, 16 bites a művelet
        addc w1,#0,w1   ;Ha túlcsordult, akkor adjunk hozzá 1-et
        dec w3,w3
        bra NZ, H_o_ciklus
        pop w3
        pop w2
        return
```



Mi lesz, ha elfogy a stack?

- A stack a teljes RAM-ot felhasználhatja, ahol nincsenek változók.
- A memória (stack) mérete korlátozza a szubrutinok egymásba ágyazását
  - A fenti 20 db paraméterű szubrutin összesen 24 word = 48 bájt stack-et „fogyaszt” el, még így is kb. 340-szer meghívhatná önmagát… (de többször ne, mert elfogy a memória!)
  - Nyilván ha lehet ne használjunk rekurziót egy ilyen kis mikrokontrolleren
- Ha mégis elfogyna a stack, az komoly probléma…

Mi lesz, ha elfogy a stack?

Két eset okoz problémát a stack-el kapcsolatban:

1. A stack már felhasználta az egész memóriát és nincs hova írni → stack overflow
2. A stack teljesen üres, de mégis valaki olvasna belőle → stack underflow

Mindkettő biztosan csak programhiba esetén következik be, ha bekövetkezett már nincs mit tenni → általában reset.

Ha nem vesszük észre, hogy ez történik, akkor viszont a programunk „bármit” csinálhat, tehát észre kell venni.

Mindkettő észrevételére van egyszerű módszer.

Mi lesz, ha elfogy a stack?

Stack overflow észlelése:

- A PIC24 erre a célra egy SPLIM regisztert tartalmaz.
- Ha a w15 szerint indirekt címzés történik, és a cím nagyobb, mint SPLIM értéke, akkor azt a vezérlő egység észreveszi, megakadályozza, és *jelez*.
  - » Stack error trap → később lesz róla szó

Stack underflow észlelése:

- Ugyanez történik, ha a w15 értéke 0x800 alá csökken
  - » Stack error trap → később lesz róla szó

Hova tegyük a stack-et

Mivel a SPLIM határregiszter a stack végét tudja védeni, ezért célszerű a memória elejére. A mögötte lévő részt használhatjuk változóknak.

Az xc16 linker elhelyezi a változóinkat és a stack-et is, nekünk csak fel kell használni az értékeket:

```
mov #__SP_init,w15 ; stack pointer inicializálás
mov #__SPLIM_init,w0
mov w0,_SPLIM ; stack limit inicializálás
```

Ha ez előtt függvényt hívunk, az biztosan „Stack error”-t okoz

__SP_init és __SPLIM_init értékeit a linker kiszámítja és be fogja helyettesíteni a memóriahasználat (globális változók száma) függvényében.

További utasítások

Utasítás csoportok:

- Adatmozgató utasítások (ezeket láttuk)
- Aritmetikai műveletek
- Logikai műveletek
- Bitműveletek
- Vezérlés átadás
- **Egyéb (vezérlő) műveletek**

Egyéb (vezérlő) utasítások

- Speciális, máshova nem sorolható műveletet végeznek
  - `nop` ;semmit sem csinál
  - `reset`;újraindítja az egész mikrokontrollert
  - `pwrsav` ;alvó, kis fogyasztású módba lép
    - » nem folytatja a programfuttatást, majd még később látjuk ez mire jó
  - `repeat` ;a következő utasítást megismétli valahányszor
    - » néha nagyon hasznos!
- Van még néhány, amiről később lesz szó:
  - `clrwdt`
  - `disi`

Repeat utasítás

Két formája van:

```
repeat #const    ; a következő utasítást const+1-szer ismételi meg
repeat wn        ; a következő utasítást wn+1-szer ismétli meg
```

- A konstans csak 14 bites (maximum 16384-szer lehet ismételni így)
  - » A regiszter természetesen 65535 is lehet.
- A megadott regiszter nem változik.
- A repeat is 1 ciklusig eltart
- A következő utasítás egyszer mindenképp lefut (ezért a +1)
- A következő utasítás nem lehet: ugrás, függvényhívás, visszatérés, másik repeat…
- Láttuk, hogy a ciklusszervezés elég sok plusz időt vesz igénybe
  - Ezzel egyszerű egy utasításos „ciklus”-hoz nem is kell ciklus
- A repeat kell az osztás művelethez is → gyakorlaton látni fogjuk

Repeat utasítás

```
Tomb_osszead:       ;meghívni 2 Tcy
          push w1    ;1 Tcy
          push w2    ;1 Tcy
          clr w2     ;1 Tcy
T_o_ciklus:
          add w2, [w0++], w2 ;1 Tcy
          dec w1, w1           ;1 Tcy
          bra NZ,T_o_ciklus ;2 Tcy  (kivéve az utolsó)
          mov w2, w0           ;1 Tcy
          pop w2     ;1 Tcy
          pop w1     ;1 Tcy
          return     ;3 Tcy
```

```
Tomb_osszead:       ;meghívni 2 Tcy
          ;w1-et nem kell elmenteni, mert nem rontjuk el
          push w2   ;1 Tcy
          clr w2    ;1 Tcy
          dec w1,w1           ;mivel w1+1-szer ismételjük
          repeat w1 ;1 Tcy
          add w2, [w0++], w2 ;1Tcy
          inc w1,w1           ;vissza is kell növelni
          mov w2, w0          ;1 Tcy
          pop w2    ;1 Tcy
          ;w1-et visszatölteni sem kell
          return    ;3 tcy
```

Pl. 100 elemű tömb esetén:
Összesen 4*100-1+11 = 410 ciklus

Pl. 100 elemű tömb esetén:
Összesen 1*100+1+11 = 111 ciklus

Majdnem 4x gyorsabb…

A feladatmegoldásoknál mindig egy működő és sosem a legegyszerűbb megoldást kérjük.

Szoftveres időzítés

- A valóságban sokszor várni kell jelekre, vagy a külvilágra.
- Pl. az alábbi jelalakot kellene előállítani az A port 11-es lábán:

- A kimeneti pulzus hossza legyen minimum 1 µs

Szoftveres időzítés

- Pl. az alábbi jelalakot kellene előállítani az A port 11-es lábán:

- A kimeneti pulzus hossza legyen minimum 1 µs

```
bset LATA,#11   ;1 Tcy
blcr LATA,#11   ;1 Tcy
```

32MHz-es órajel → 16MHz utasításciklus, 1 Tcy = 62.5ns

Szoftveres időzítés

A kimeneti pulzus hossza legyen minimum 1 µs

1 µs = 1000 ns = 16*62.5ns = 16 ciklus

Egyszerű, csak semmit sem kell csinálni 16 utasítás idejéig:

```
repeat #14    ;1 Tcy
nop           ;1 Tcy, 15-ször lesz végrehajtva
```

Szoftveres időzítés

A kimeneti pulzus hossza legyen minimum 1 µs

1 µs = 1000 ns = 16*62.5ns = 16 ciklus

16-ból 1 ciklus már így is van, már csak semmit sem kell csinálni még 15 utasítás idejéig:

```
bset LATA,#11     ;1 Tcy
repeat #13    ;1 Tcy
nop           ;1 Tcy, 14-szer lesz végrehajtva
blcr LATA,#11     ;1 Tcy
```

Szoftveres időzítés

- Az ismételt nop maximum ideje is csak 65538 ciklus
  - » ebben már a repeat w0 és a w0 konstanssal feltöltése is benne van
  - » Ez kb 4 ms, ha ennél több kell, valahogy máshogyan kell megoldani.
- Lefelé számláló ciklus:

```
Delay:              ;a függvény meghívása 2Tcy lesz
    push w0       ;1Tcy    mentsük el W0-t mert elrontjuk
    mov #const,w0  ;1Tcy
D_ciklus:
    sub  w0,#1,w0   ;1 Tcy
    bra NZ,D_ciklus  ;2 Tcy
    pop w0       ;1Tcy
    return         ;3Tcy
```

- Ez még mindig ugyanaz csak kevésbé hatékony
  - » és most pont ez a cél ☺
  - » 7+const*3 Tcy
  - » Maximum időhöz 0 konstans is lehet (előbb csökkent!)
  - » Maximum ez is csak 12,2 ms

Szoftveres időzítés

- Ha nem lenne elég:
  - » 32 biten is lehet lefele számolni

```
Delay1s:          ;a függvény meghívása 2Tcy lesz
    push w0       ;1Tcy    mentsük el W0-t és W1-et is, mert elrontjuk
    push w1           ;1 Tcy
    mov #constL, w0   ;1Tcy
    mov #constH, w1       ;1Tcy
dcikl:
    sub  w0,#1,w0    ;1 Tcy
    subb w1,#0,w1   ;1 Tcy  /C-t is kivonja ezért mi már csak 0-t vonunk ki, a Z-t csak törli, nem
állítja
    bra NZ,dcikl   ;2 Tcy ha még mindig 1 a Z flag akkor az egész összeg 0 volt
    pop w1       ;1Tcy
    pop w0       ;1Tcy
    return         ;3Tcy
```

- (9 + (constH:constL)*4) Tcy
- Így már a maximum idő: kb. 1073,7 s, vagyis majdnem 18 perc
  - » Ennyit azért ritkán kell így várni…