BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM
VILLAMOSMÉRNÖKI ÉS INFORMATIKAI KAR
MÉRÉSTECHNIKA ÉS INFORMÁCIÓS RENDSZEREK TANSZÉK

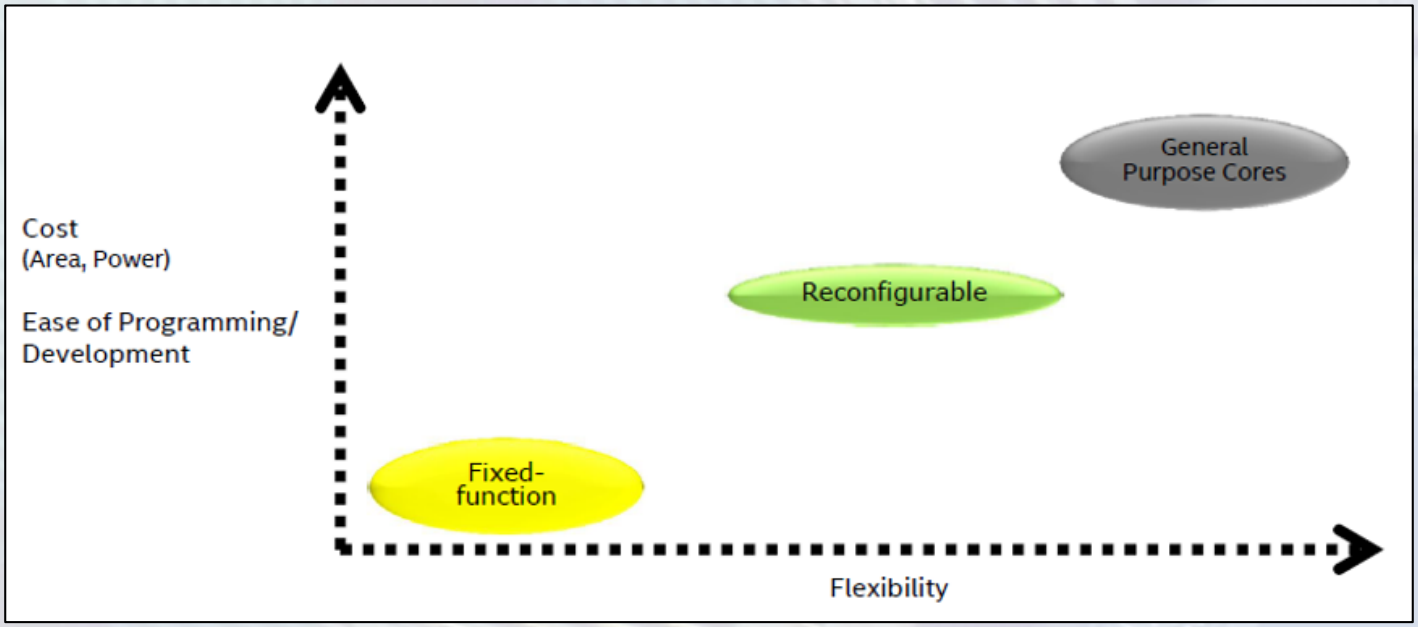
Mikrorendszerek tervezése

Általános bevezetés

Fehér Béla
Raikovich Tamás

Technológiai áttekintés

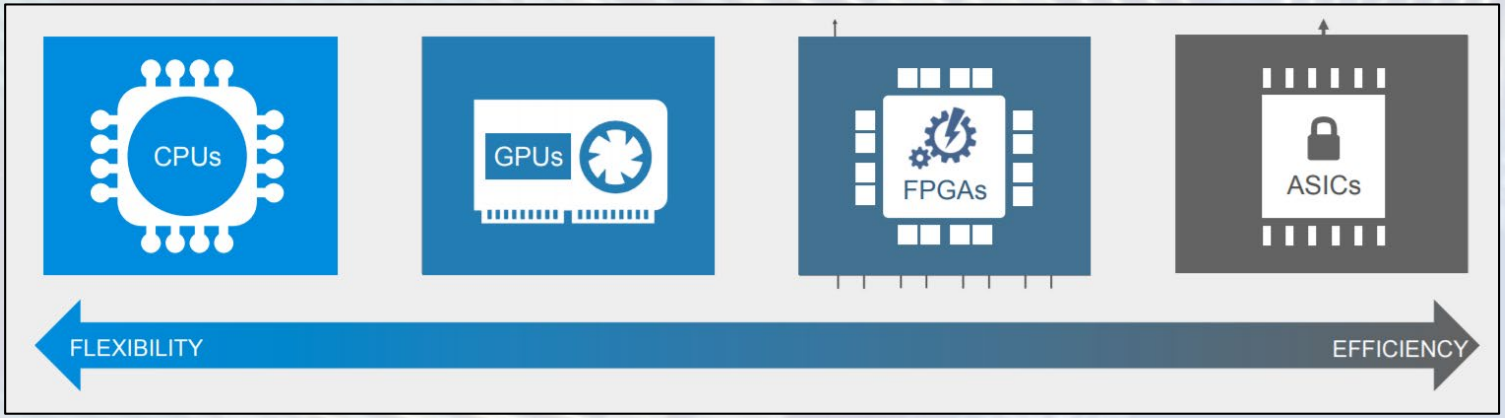
- **Félvezető technológia lehetőségei, korlátai**
- **Rendszer megvalósítási lehetőségek**
  - Egyedi dedikált megoldás
  - Általános célú megoldás



Technológiai áttekintés

Működési paraméterek

- **Általános célú megoldás**
  - Egyszerű, szoftveresen programozható eszköz
  - Ciklikus iteratív feladatmegoldás, alacsony teljesítmény
- **Optimalizált megoldás**
  - Közvetlen algoritmus-architektúra leképzés
  - Nagy sebesség, kis késleltetés, kis fogyasztás



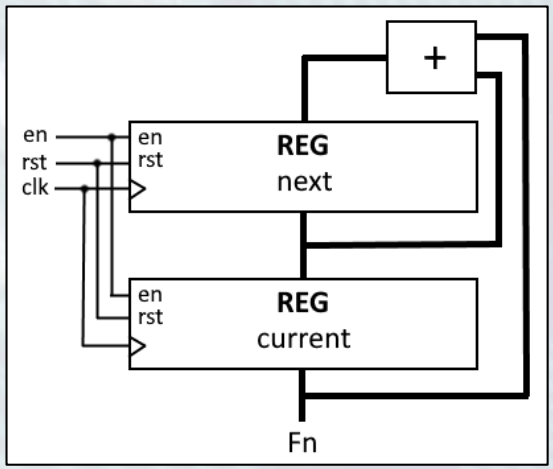
Példa feladatok: Fibonacci sorozat

- **Fibonacci sorozat generálása**
  - $F_n = F_{n-1} + F_{n-2}$ és $F_0 = 0$, $F_1 = 1$
  - 8 bites számábrázolás esetén:
    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233
- **Vizsgált verziók**
  - Hardver: Verilog HDL, 2 regiszter és egy összeadó
  - Verem alapú + program
  - Akkumulátor alapú + program
  - 2 regisztercímes + program
  - 3 regisztercímes + program
- **Szempontok: sebesség és kódméret**

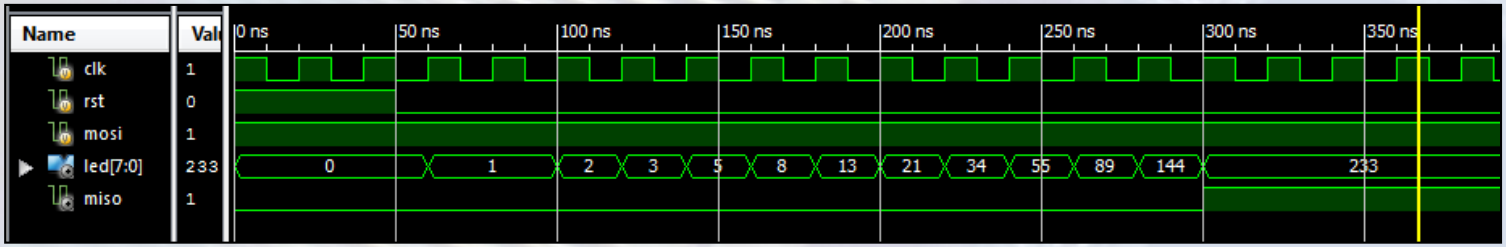
Példa feladatok: Fibonacci sorozat

Hardveres realizáció (Verilog HDL):

- Költség: 2 db 8 bites regiszter és 1 db 8 bites összeadó
- Ha engedélyezett, akkor minden órajelre egy új F_n érték




```
reg [7:0] current, next;            // Belső változók a Fibonacci sorozat elemeihez
wire en = mosi;                     // Léptetés engedélyezése

always @ (posedge clk)              // Minden működtető órajelre lép egyet
begin
   if (rst)                         // RESET estén inicializálás
      begin
         current <= 8'b0;
         next <= 8'b1;
      end
   else
      if (en)                       // Ha engedélyezett a lépés
      begin
         current <= next;           // megújítja az aktuális értéket
         next <= current + next;    // és kiszámolja az következőt
      end
end
```



Példa feladatok: LNKO (GCD)

- **Egy számpár legnagyobb közös osztóját keressük**
- **GCD(a,b): euklideszi algoritmus → maradékos osztás**
  - ***a = b*q1 + r1,*** ***r1 = a % b***
  - ***b = r1*q2 +r2,*** ***r2 = b % r1***
  - ***r1 = r2*q3 +r3*** ***r3 = r1 % r2***
  - ahol a > b > r1 > r2 … ≥ 0 ahol % a Verilog mod op.
  - A GCD az utolsó nem nulla maradék
  - Jó algoritmus, viszonylag gyorsan konvergál
- **DE: A Verilog HDL % operátor nem szintetizálható!**
  - Tervezzünk egy osztó modult? Lehet, de nem könnyű.

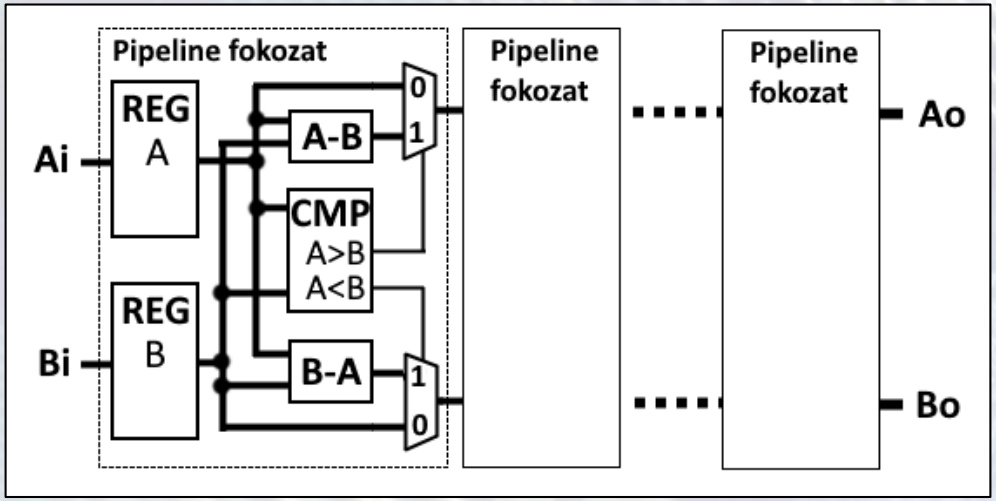
Példa feladatok: LNKO (GCD)

- **Egyszerűsítsük az algoritmust szintetizálható műveletre**
  - GCD(a,b) = GCD(a-b, b), ha a > b Művelet: a-b → a
  - GCD(a,b) = GCD(a, b-a), ha b > a Művelet: b-a → b
  - Leállás adott lépés után, ha ao = bo, ez a GCD(a,b)
- **Ez már szintetizálható, egyszerűen tervezhető**
  - De a komplexitás exponenciális lesz!

Példa feladatok: LNKO (GCD)

Teljesen párhuzamos HW megvalósítás: pipeline

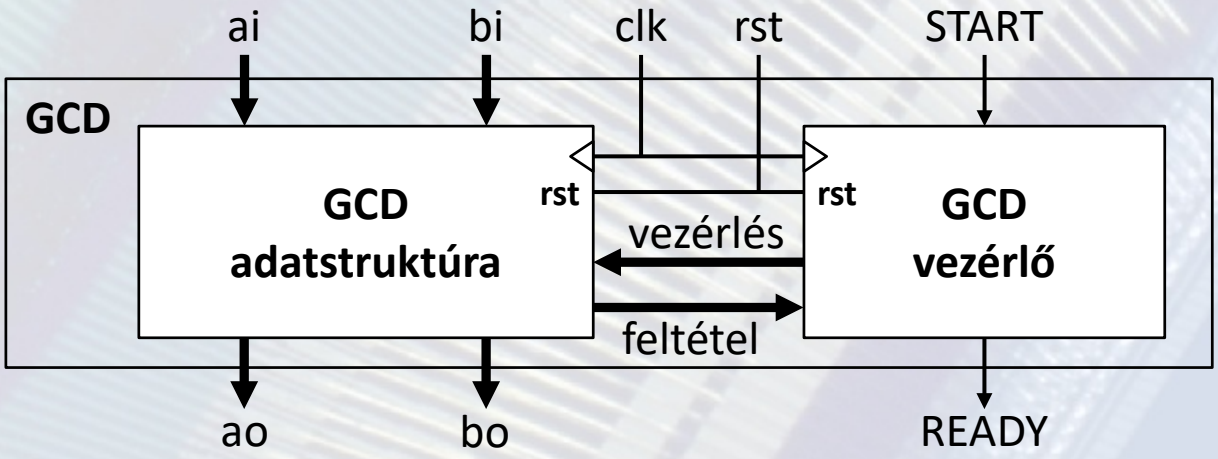
- **1 fokozat egy részszámítást végez el**
- **Minden órajelciklusban új bemenet és eredmény**
- **Fokozatok száma: N bites adatok esetén 2^N**
  - A legrosszabb esetre kell felkészülni
- **Gyors, de nagy erőforrás igényű**



Példa feladatok: LNKO (GCD)

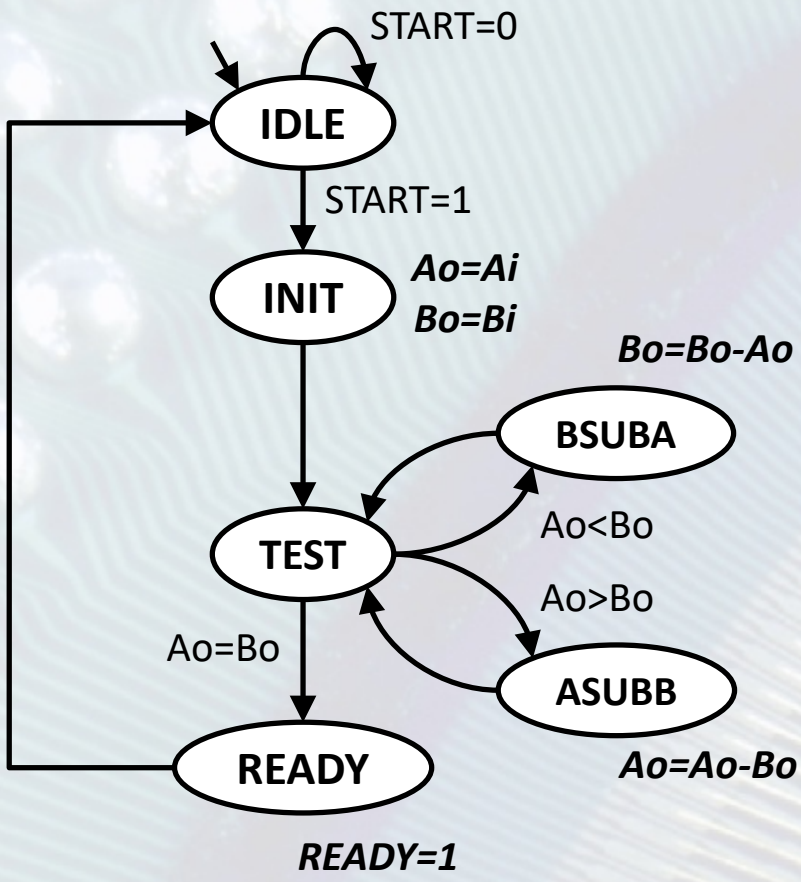
Soros végrehajtású HW megvalósítás

- **Szétválasztás adatstruktúrára és vezérlőre**
- **Adatstruktúra: az elemi műveletek megvalósítása**
- **Vezérlő: az adatstruktúra vezérlése, indítás (START) és kész jelzés (READY) kezelése**
- **Lassabb, de kevesebb erőforrást igényel**
  - N bites bemenet esetén a lépésszám $O(2^N)$



Példa feladatok: LNKO (GCD)

Magasszintű állapotgép



Interfész és belső regiszterek

- Bemenetek: START, Ai, Bi
- Kimenetek: READY, Ao, Bo
- Regiszterek: Ao, Bo

Adatstruktúra

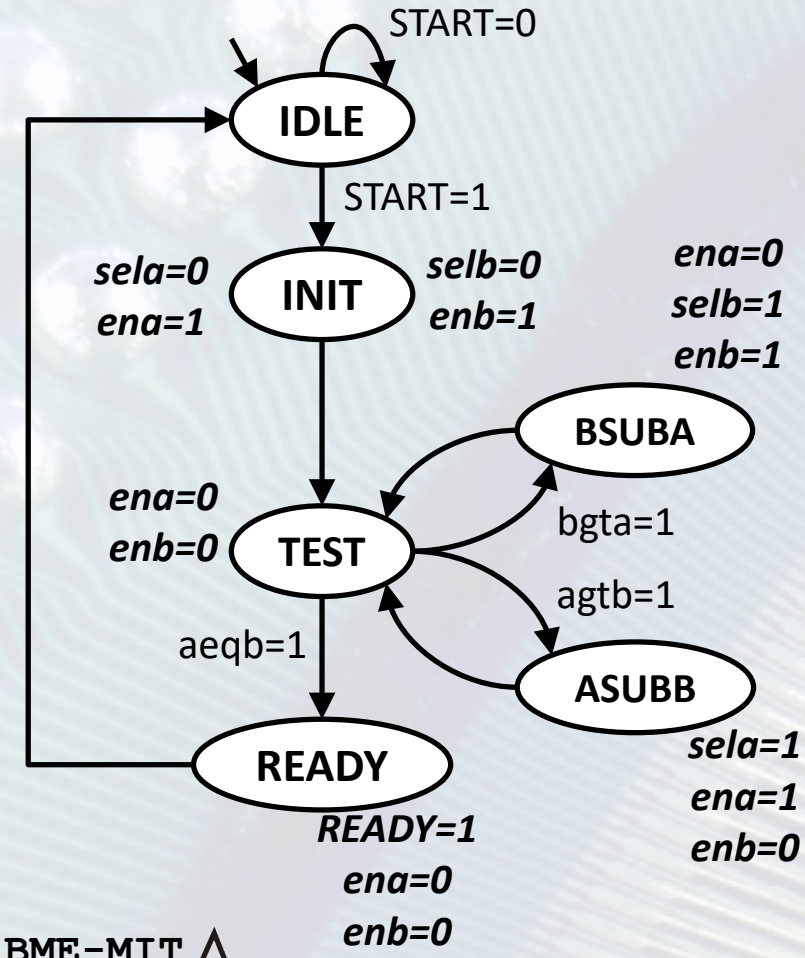
- Magasszintű műveletek megvalósítása
- Vezérlő jelek (←FSM)
- Feltétel jelek (→FSM)

Vezérlő

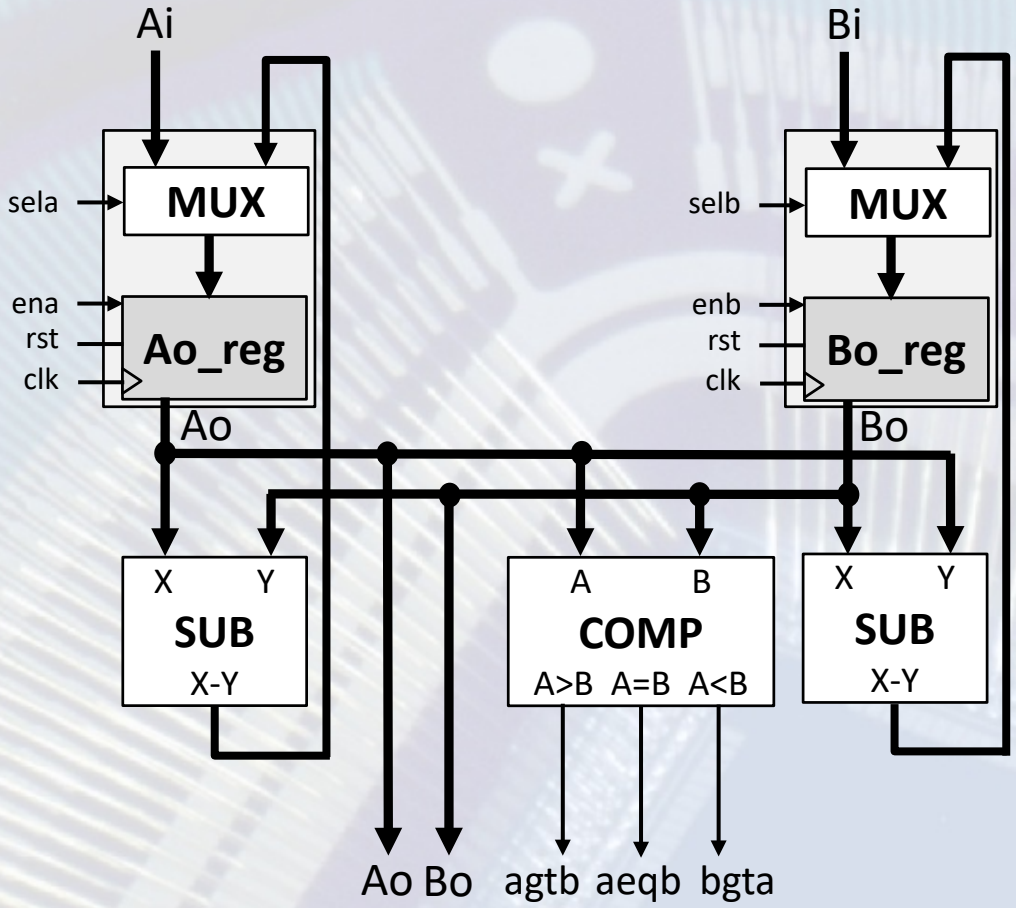
- Véges automata (FSM)
- Bitszintű műveletek
- Külső parancs és státusz jelek

Példa feladatok: LNKO (GCD)

A vezérlő állapotdiagramja



Az adatstruktúra egységei

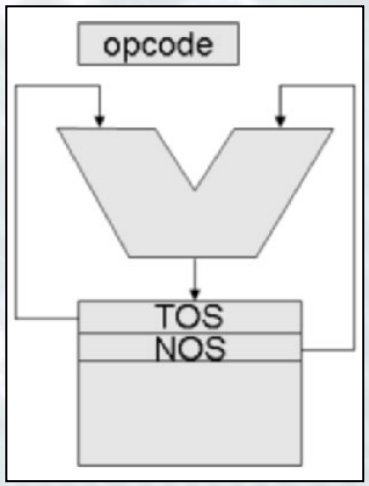


Processzor adatstruktúrák

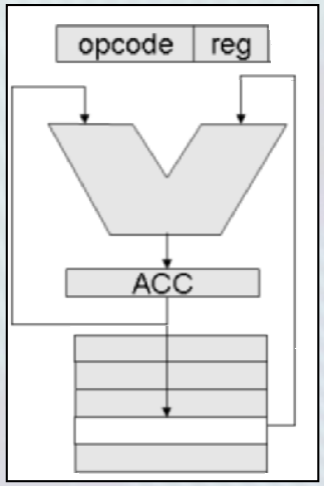
- **Előző példák: dedikált műveletvégző egységek**
- **Általános célú műveletvégző: programozható CPU**
  - (Általános célú) adatstruktúra + vezérlő
  - Adatstruktúra
    - Adatmozgatás
    - Aritmetikai: összeadás, kivonás (szorzás, osztás)
    - Logikai: bitenkénti AND, OR, XOR
    - Léptetés, forgatás
    - Státusz bitek: zero (Z), carry (C), negative (N), overflow (V)
  - Vezérlő
    - Az utasítások beolvasása, értelmezése és végrehajtása az adatstruktúrán
    - Feltétel nélküli és feltételes ugrás, szubrutinhívás
- **Programozás**
  - Az adott feladat leképzése a processzor utasításkészletére

Általános adatstruktúrák

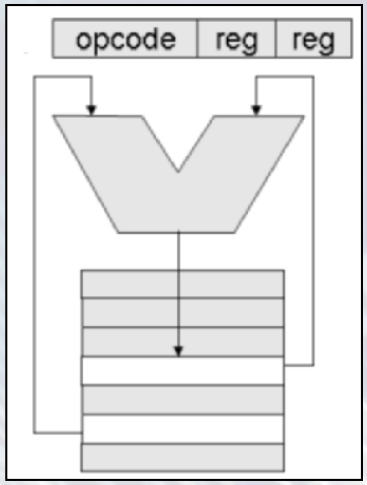
STACK



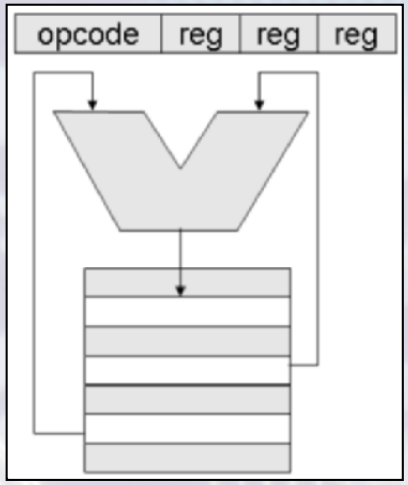
ACC



2REG



3REG



- **STACK: forrás a legfelső 2 adat, cél a verem teteje**
- **ACC: az egyik forrás és a cél mindig az akkumulátor**
- **2REG: az egyik forrás és a cél operandus azonos**
- **3REG: külön forrás és cél operandusok**

CISC processzorok

- **Complex Instruction Set Computer (↔ RISC)**
- **Ez a típus volt előbb**
- **Egy utasítás több alacsonyszintű műveletből állhat**
  - Pl. memóriában tárolt érték megnövelése:
    - Memória olvasás, aritmetikai művelet, memória írás
  - Változó órajelciklus igény
  - Magasszintű nyelvi elemek közvetlen támogatása
- **Sokféle címzési mód**
- **Nagyszámú és változó méretű utasításkészlet**
- **Kevés belső regiszter (akkumulátor alapú)**
- **Neumann architektúra: egyetlen közös memória interfész**
- **Mikroprogramozott vezérlő (mikrokód)**
- **Példák: MOS 6502, Intel 8051, Intel x86, Zilog Z80**
  - A mai x86 processzorok belül már RISC felépítésűek!

RISC processzorok

- **Reduced Instruction Set Computer (↔ CISC)**
- **A CISC utasítások helyettesítése egyszerűbb műveletekkel**
  - 1 órajel alatt végrehajthatók
  - Pipeline felépítés és utasítás végrehajtás
- **Kevés címzési mód, ortogonális utasításkészlet**
  - Az utasításoknál használható az összes címzési mód
- **Fix méretű utasításkészlet**
- **Load-store architektúra**
  - Műveletvégzés csak belső regisztereken
  - CISC: memóriában tárolt érték növelése → RISC: 3 utasítás
- **Sok belső regiszter**
- **Harvard architektúra: külön program- és adatmem. interfész**
- **Példák: ARM, RISC-V, IBM PowerPC, MIPS, Xilinx MicroBlaze**

CISC processzorok – MOS 6502

- **1975-ben készült 8 bites CISC processzor**
- **16 bites cím és 8 bites adat interfész**
- **Kb. 3500 tranzisztor**
- **Az olcsó ára miatt ($25) nagyon népszerű volt**
  - Commodore 64, Nintendo Entertainment System
- **Regiszterkészlet**
  - Egy 8 bites akkumulátor (A)
  - Két 8 bites index regiszter (X és Y)
  - Egy 8 bites veremmutató (SP)
  - Egy 16 bites programszámláló (PC)
  - Egy 8 bites státusz regiszter (P)

CISC processzorok – MOS 6502

- **Az memória alsó 256 bájtos tartományának kitüntetett a szerepe**
  - Zero Page (ZP), 256 elemű 8 bites „regisztertömb”
- **Címzési módok**
  - ***Implicit:*** nincs operandus
  - ***Akkumulátor – A:*** a művelet az akkumulátoron hajtódik végre
  - ***Immediate – #imm:*** konstans ALU operandus
  - ***Zero page – d:*** 8 bites ZP memóriacím, ZP[d]
  - ***Abszolút – a:*** 16 bites (teljes) memóriacím
  - ***Relatív – r:*** a PC-hez relatív 8 bites memóriacím PC=PC + r
  - ***Indirekt – (a):*** az operandus beolvasása 16 bites mem. címről
  - ***Zero page indexelt – d, X / d, Y:*** ZP[d + X] vagy ZP[d + Y]
  - ***Abszolút indexelt – a, X / a, Y:*** MEM[a + X] vagy MEM[a + Y]
  - ***Indexelt indirekt – (d, X):*** MEM[(ZP[d + X + 1] << 8) + ZP[d + X]]
  - ***Indirekt indexelt – (d), Y:*** MEM[((ZP[d + 1] << 8) + ZP[d]) + Y]

CISC processzorok – Intel 80386

- **Regiszterkészlet (32/16 bites) – akkumulátor alapú**
  - Fő regiszterek: **EAX/AX**, EBX/BX, ECX/CX, EDX/DX
  - Index regiszterek: ESI/SI, EDI/DI, EBP/BP, ESP/SP
  - Programszámláló: EIP/IP
  - Szegmens regiszterek (16 bitesek): CS, DS, ES, FS, GS, SS
  - Státusz regiszter
- **Memória címzési módok (változó hosszúságú utasítások!)**
  - 16 vagy 32 bites abszolút
  - Regiszter indirekt
  - Regiszter + offszet indirekt
  - Regiszter + regiszter indirekt
  - Regiszter + regiszter + offszet indirekt
  - Regiszter + (regiszter * skála) + offszet indirekt
    - A skálatényező 1, 2, 4 vagy 8 lehet

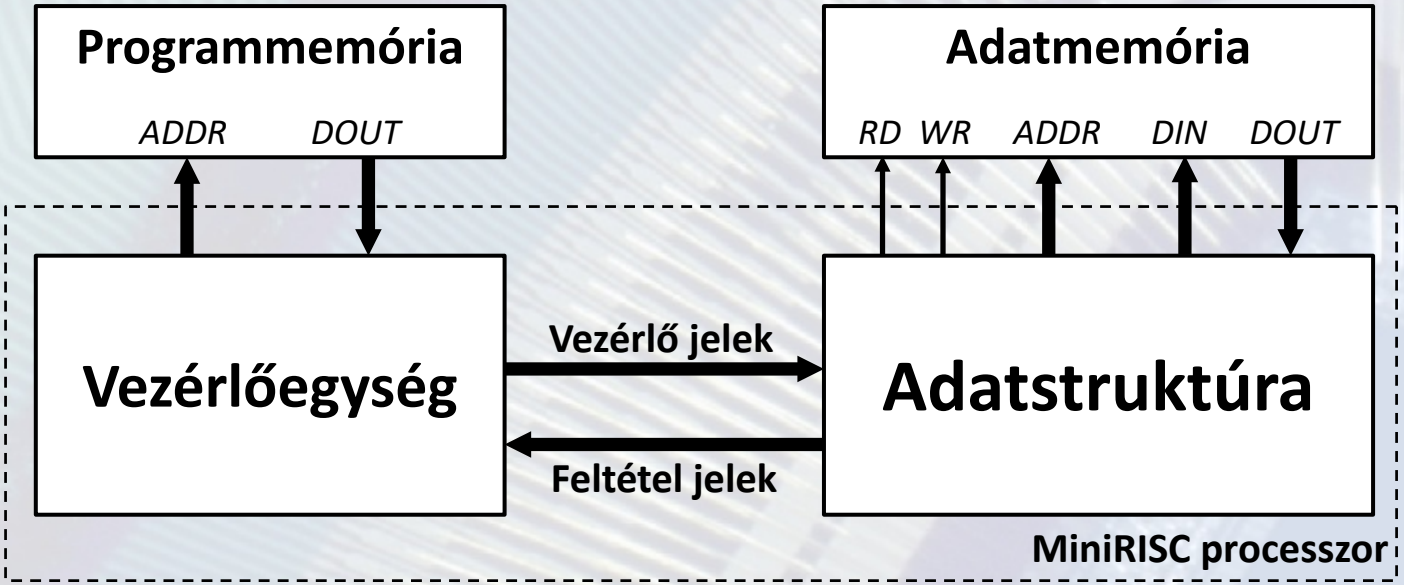
RISC processzorok – Atmel AVR

- **8 bites RISC mikrovezérlő**
- **16 bites cím és 8 bites adat interfész**
- **32 darab 8 bites általános célú regiszter**
  - Indexeléshez: r27:r26 (X), r29:r28 (Y), r31:r30 (Z)
- **Fix 16 bites utasításméret**
  - 2 regisztercímes architektúra
- **Kétfokozatú pipeline**
  - Utasítás lehívás (fetch), végrehajtás (execute)
- **Memória címzési módok**
  - Abszolút
  - Regiszter indirekt (X, Y, Z)
  - Regiszter indirekt poszt inkremens (X+, Y+, Z+)
  - Regiszter indirekt pre dekremens (-X, -Y, -Z)
  - Regiszter indirekt + offszet (Z + offszet)

Programozható processzorok - Felépítés

Felépítése követi az adatstruktúra-vezérlő szemléletet

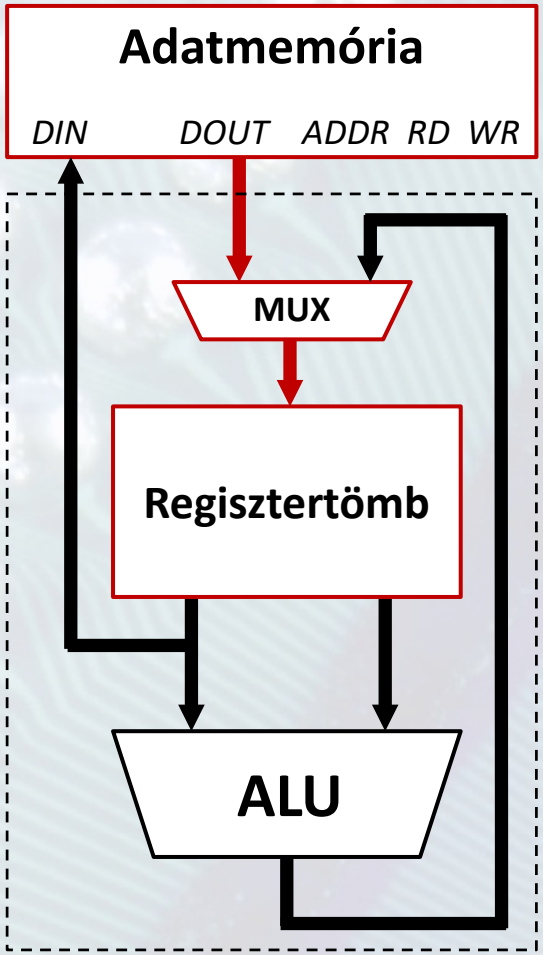
- ***Vezérlő:*** **az utasítások beolvasása, feldolgozása és ennek megfelelően az adatstruktúra vezérlése**
- ***Adatstruktúra:*** **műveletek végrehajtása az adatokon**



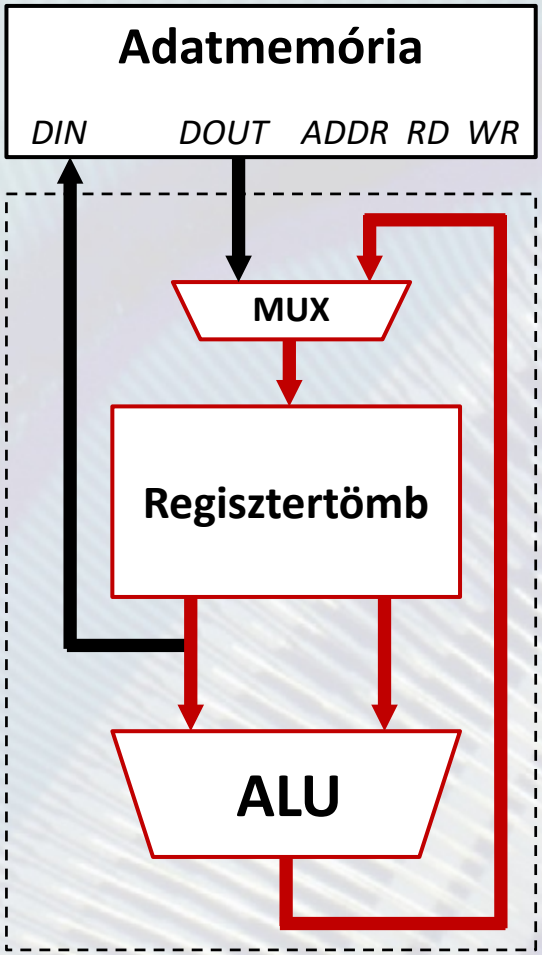
Programozható processzorok - Felépítés
(Adatstruktúra)

- **Az adatstruktúrában történik a műveletek végrehajtása az adatokon. Az adatfeldolgozás fő lépései a következők:**
  1. A feldolgozandó adatok betöltése
  2. Ezen adatok átalakítása
  3. Az eredmény elmentése
- **Az alap adatstruktúra ennek megfelelően:**
  - Olvassa és írja az adatmemóriát, ahol a fő adatok vannak
  - Tartalmaz regisztertömböt az adatok lokális tárolásához
  - Tartalmaz aritmetikai-logikai egységet (ALU-t) a lokális adatok átalakításához

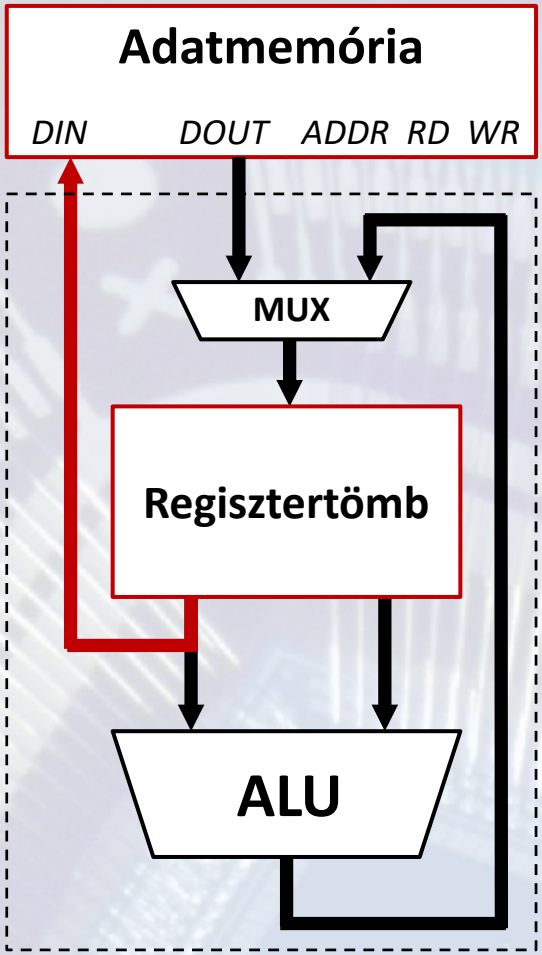
Programozható processzorok - Felépítés
(Adatstruktúra)



Adatmemória olvasás (load)



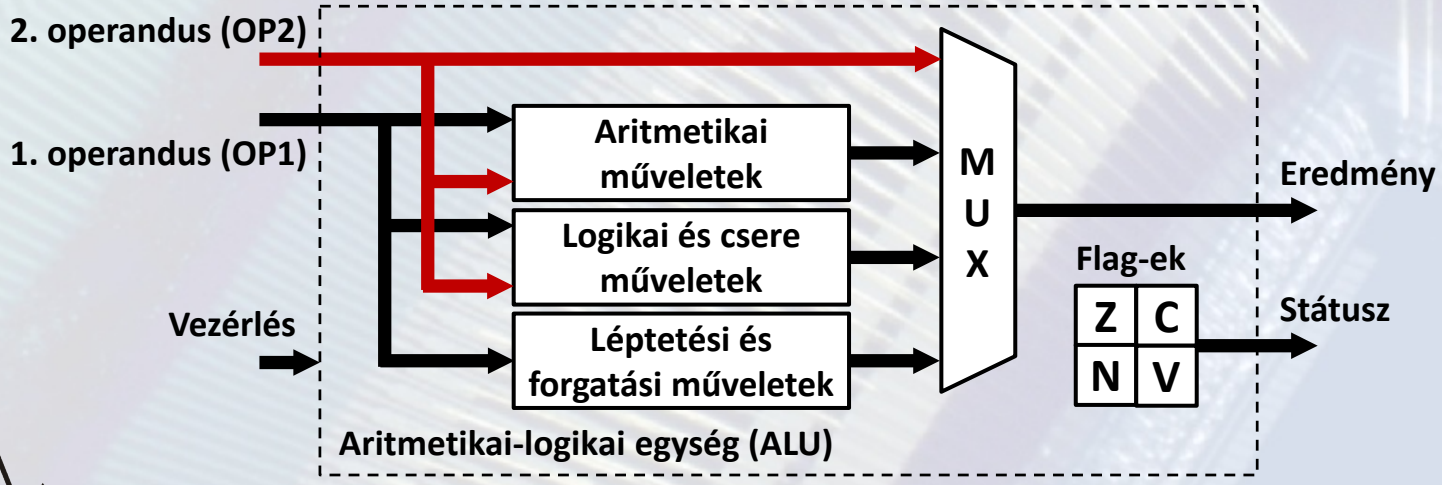
Lokális adat átalakítása (ALU művelet)



Adatmemória írás (store)

Programozható processzorok - Felépítés (Adatstruktúra)

- **Az ALU műveleteket végez a lokális adatokon**
  - Adatmozgatás: nincs műveletvégzés, eredmény az OP2
  - Aritmetikai: összeadás és kivonás átvitel bittel vagy anélkül
  - Logikai: bitenkénti AND, OR, XOR
  - Léptetés és forgatás, csere
- **Státusz bitek: a műveletvégzés eredményéről adnak információt**
  - Zero (Z), carry (C), negative (N) és overflow (V) státusz bitek
  - Értéküket a feltételes ugró utasításokkal tesztelhető

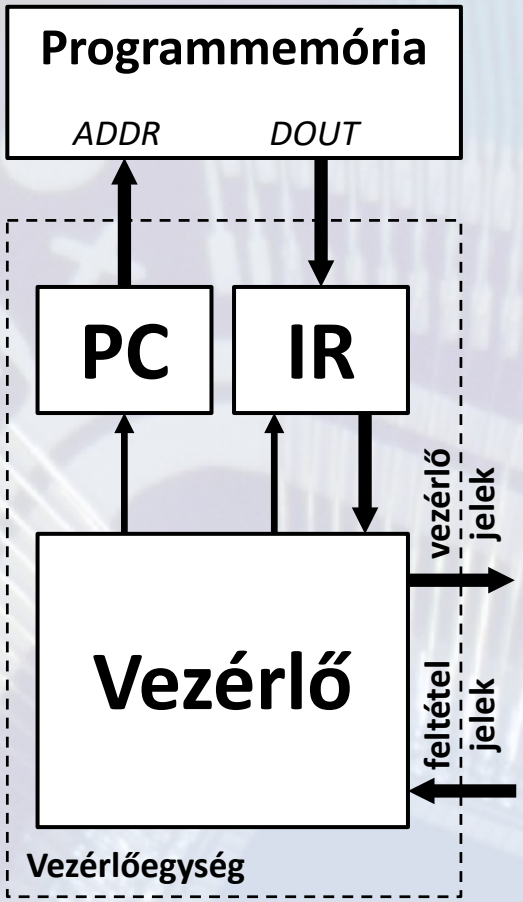


Programozható processzorok - Felépítés (Adatstruktúra)

- **Az ALU státusz bitek a művelet eredményéről adnak információt**
- ***Zero bit (Z)***
  - Jelzi, ha az ALU művelet eredménye nulla
  - Az adatmozgatás nem változtatja meg az értékét
- ***Carry bit (C)***
  - Aritmetikai művelet esetén jelzi, ha átvitel történt
  - Shiftelés vagy forgatás esetén felveszi a kiléptetett bit értékét
- ***Negative bit (N)***
  - Kettes komplemens előjelbit, az eredmény MSb-je (7. bitje)
  - Az adatmozgatás nem változtatja meg az értékét
- ***Overflow bit (V)***
  - A kettes komplemens túlcsordulást jelzi
    - Az aritmetikai művelet eredménye nem fér el 8 biten

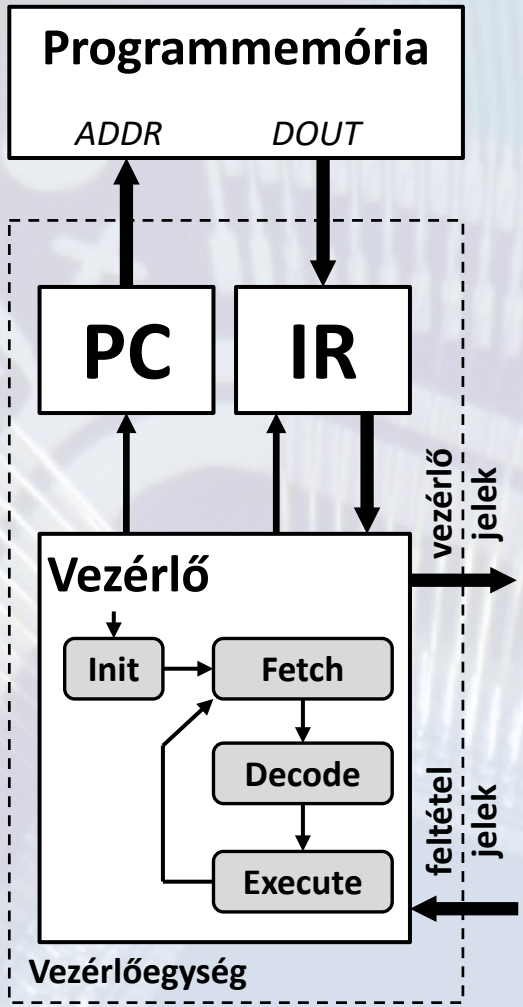
Programozható processzorok - Felépítés
(Vezérlőegység)

- **Példa: DMEM[3] = DMEM[0] + DMEM[1], ez négy adatstruktúra műveletet igényel:**
  1. REG[0] = DMEM[0] (load)
  2. REG[1] = DMEM[1] (load)
  3. REG[1] = REG[0] + REG[1] (ALU művelet)
  4. DMEM[3] = REG[1] (store)
- **Utasítás: a processzor által végrehajtható művelet**
- **Program: utasítások sorozata**
  - A végrehajtandó feladatot a processzor által támogatott utasításokra kell lebontani
- **A programot a programmemória tárolja**
- **A vezérlőegység beolvassa az utasításokat és végrehajtja azokat az adatstruktúrán**
  - ***Programszámláló (Program Counter, PC):*** a beolvasandó utasítás címét állítja elő
  - ***Utasításregiszter (Instruction Register, IR):*** a beolvasott utasítást tárolja



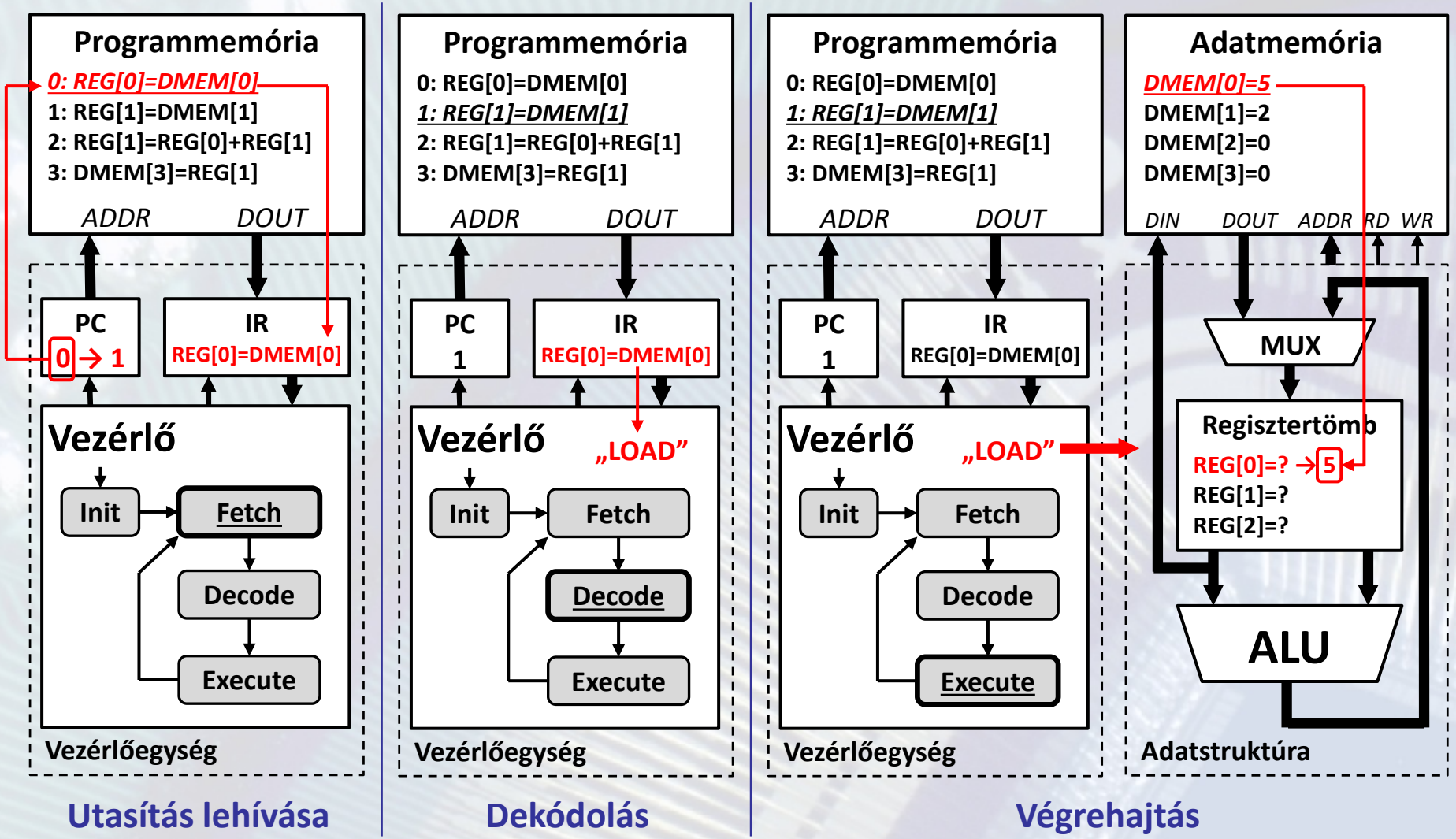
Programozható processzorok - Felépítés
(Vezérlőegység)

- **Minden egyes utasítás végrehajtásánál a vezérlőegységnek a következő lépéseket kell elvégeznie:**
  - ***Lehívás (fetch):*** az utasítás beolvasása a programmemóriából és a PC növelése
  - ***Dekódolás (decode):*** a művelet és az operandusok meghatározása
  - ***Végrehajtás (execute):*** az utasításhoz tartozó művelet végrehajtásra kerül az adatstruktúrán
- **A vezérlő lehet például egy állapotgép**
  - A fenti lépésekhez a vezérlő állapotgép egy-egy állapota rendelhető hozzá
  - Ekkor egy utasítás végrehajtása három órajelciklust igényel



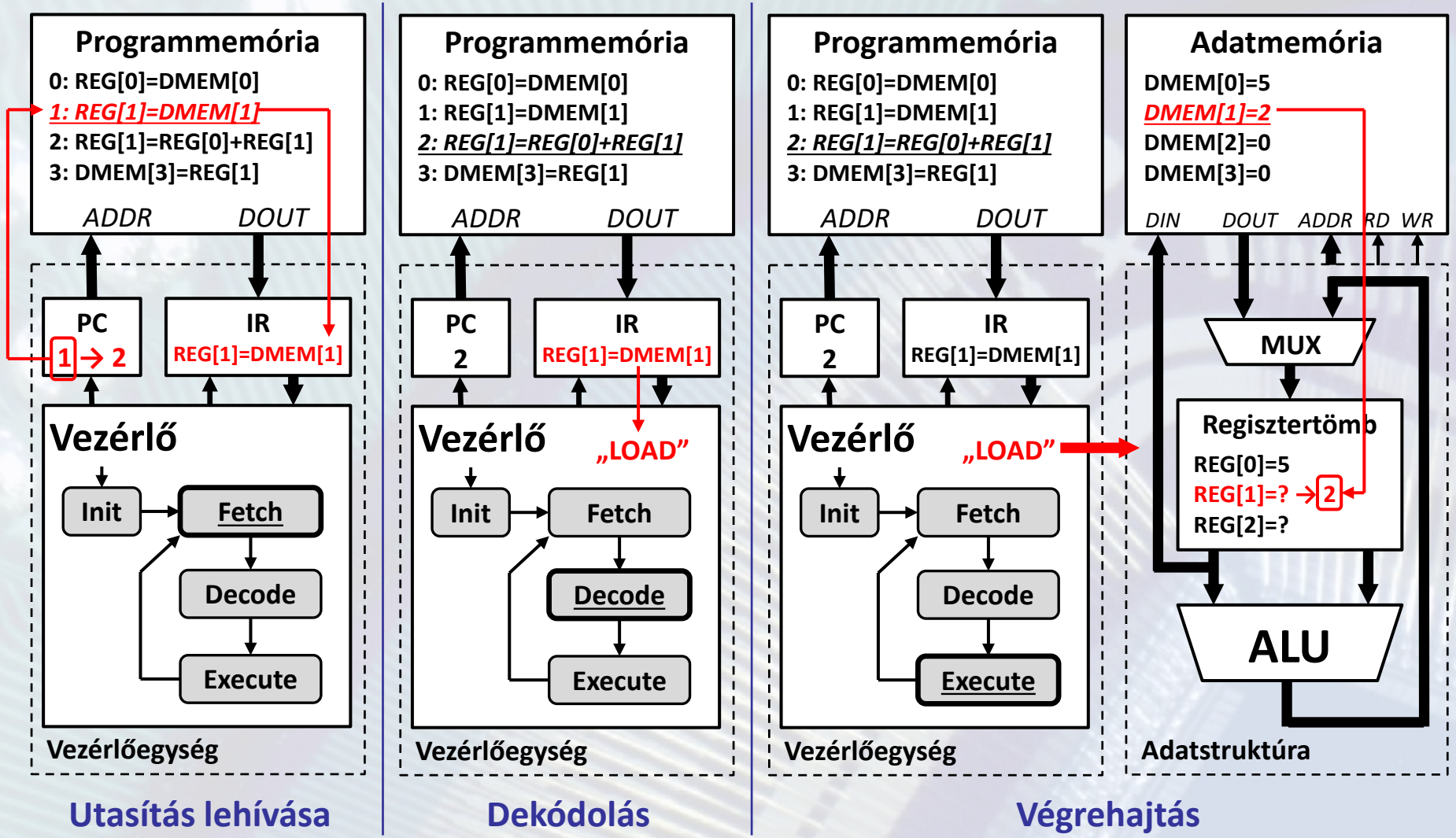
Programozható processzorok - Felépítés

(Vezérlőegység)



Programozható processzorok - Felépítés

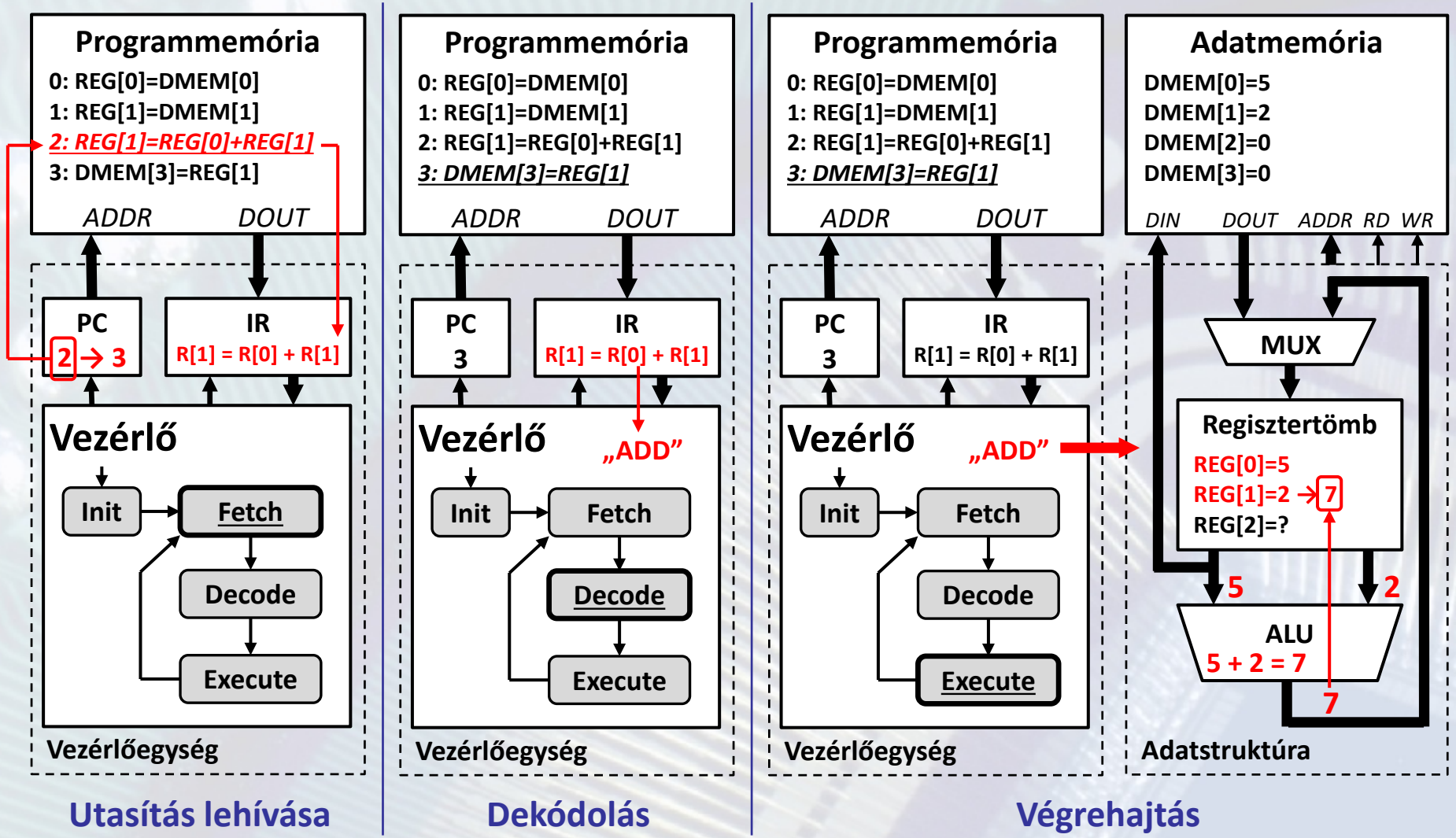
(Vezérlőegység)



Utasítás lehívása | Dekódolás | Végrehajtás

Programozható processzorok - Felépítés

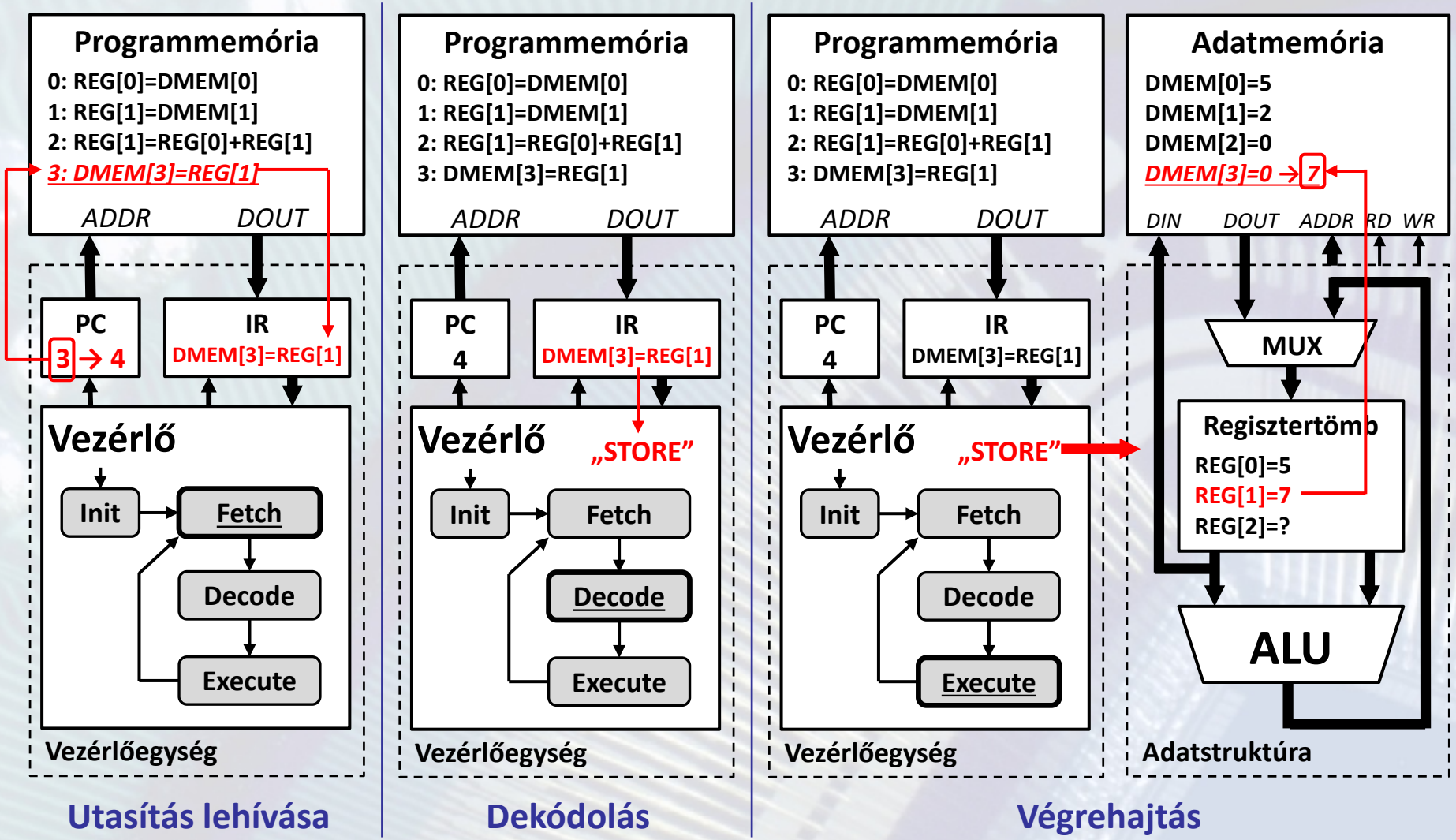
(Vezérlőegység)



Utasítás lehívása | Dekódolás | Végrehajtás

Programozható processzorok - Felépítés

(Vezérlőegység)



Programozható processzorok - Felépítés
(Vezérlőegység)

- **A programmemória bináris kód formájában tartalmazza a végrehajtandó utasításokat (gépi kód)**
  - Magasabb szintű leírást a processzor nem tud értelmezni
- **Minden utasítás tartalmazza a művelet leírását és a művelet elvégzéséhez szükséges egyéb adatokat**

művelet (4 bit) regisztercím (4 bit) memóriacím (8 bit)

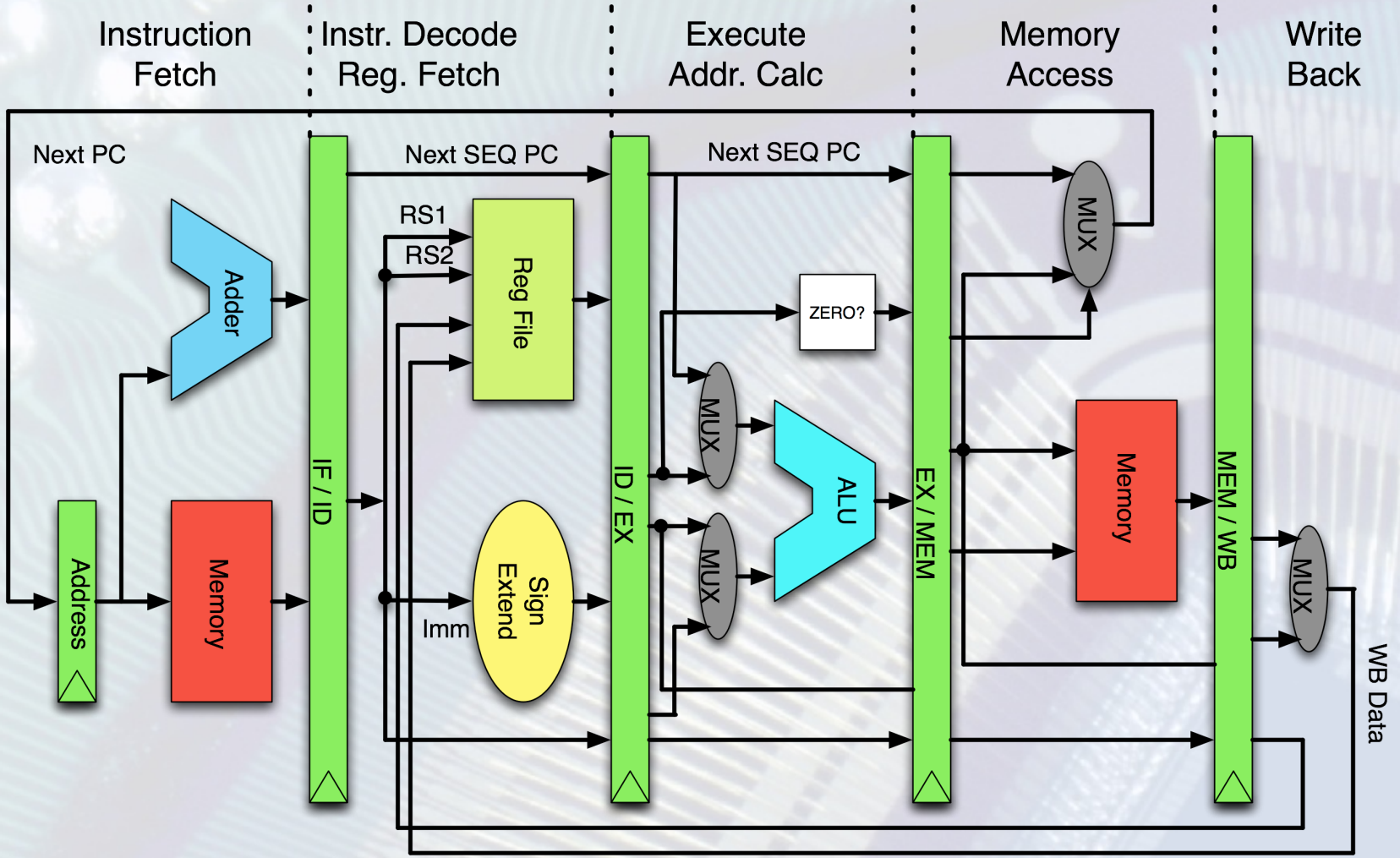
Programmemória

Cím	Művelet	Gépi kód (16 bit)	Assembly kód
0:	REG[0] = DMEM[0]	1101000000000000	MOV r0, 0x00
1:	REG[1] = DMEM[1]	1101000100000001	MOV r1, 0x01
2:	REG[1] = REG[0] + REG[1]	1111000100000000	ADD r1, r0
3:	DMEM[3] = REG[1]	1001000100000011	MOV 0x03, r1

Pipeline utasítás végrehajtás

- **Az utasítások végrehajtása során az egyes lépésekhez egy-egy pipeline fokozat tartozik**
  - Jó esetben minden órajelciklusban elkezdhető egy újabb utasítás feldolgozása
- **Pl. 3 fokozatú pipeline**
  - Utasítás elővétel (IF, fetch)
    - Programmemória olvasás
    - Beírás az utasítás regiszterbe
    - A programszámláló növelése
  - Utasítás dekódolás (DC, decode)
    - Az elvégzendő művelet meghatározása
    - A műveleti operandusok meghatározása, beolvasása
  - Utasítás végrehajtás (EX, execute)
    - A művelet végrehajtása és az eredmény visszaírása

Pipeline utasítás végrehajtás



Pipeline utasítás végrehajtás

- **Problémák – külső memória hozzáférés**
  - Az adat rendelkezésre állásáig a pipeline leáll
  - Gyorsítótár (cache) alkalmazása
- **Problémák – adat versenyhelyzet**
  - Az előző eredmény még nincs visszaírva, de operandusa a következő utasításnak
  - Megoldás
    - Az utasítások átrendezése (assembler)
    - A végrehajtás felfüggesztése, amíg az eredmény nem érhető el
    - Adat előrecsatolás (egy pipeline fokozat átugrása)

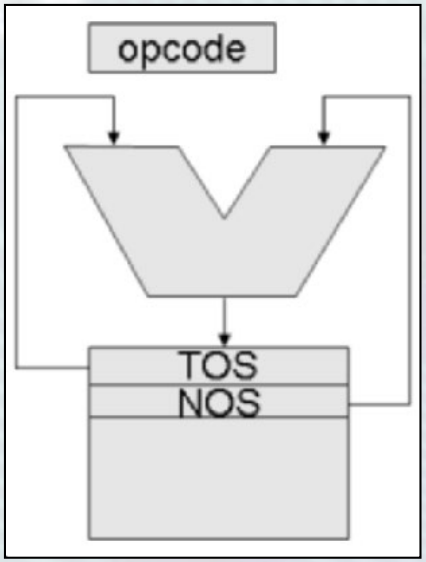
Pipeline utasítás végrehajtás

- **Problémák – ugrás**
  - A PC új értéket kap, emiatt a már lehívott utasítások érvénytelenek lesznek
  - Megoldás
    - Pipeline kiürítése: kihasználatlan órajel ciklusok
    - Delay slot: a már lehívott utasítás(ok) végrehajtása, programszervezés kérdése (a fordító feladata)
    - Ugrásbecslés: a CPU megpróbálja megjósolni az ugrási címet, helyes becslés esetén nincs veszteség
- **Problémák – megszakítás, kivétel**
  - Nem tudni, hogy mikor történik
  - Pipeline kiürítése az egyetlen lehetőség

Verem alapú adatstruktúra

Fibonacci generátor SW realizációja (pszeudó kód!)

- **Verem (stack) alapú adatstruktúra**
- **Program méret: 6 (keskeny) utasításszó**
  - Utasítás formátum: műveleti kód (nincs regisztercím)
- **Végrehajtás: 3 utasításonként egy új érték**

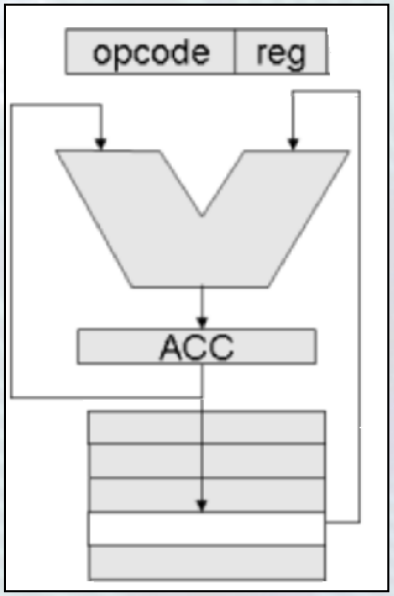



```
;*************************************************************************
;*  Fibonacci sorozat stack alapú architektúrával PSEUDÓKÓD               *
;*  Az első néhány elemre, bináris aritmetikával                          *
;*************************************************************************
DEF LD  0x80                ; LED kimeneti regiszter          (írható/olvasható)

    CODE

;*************************************************************************
;* A program kezdete.       3 előkészítő utasítás + 3 utasítás a ciklusban
;*************************************************************************
start:
    push    #0              ; Az F(0) elem a stack tetején, ez mindig az aktuális F(n)
    mov     LD, top         ; Első elem kijelzése
    push    #1              ; Az F(1) elem a stackbe, ez lesz a következő elem F(n+1)
                            ; Stack tetején 1, alatta 0
loop:
    mov     LD, top         ; Kijelezzük az aktuális elemet
    add                     ; top = top + (top-1) és automatikus push
    jmp     loop            ; Iteráció ismétlése frissített F(n), F(n+1) elemekkel
```

Akkumulátor alapú adatstruktúra

Fibonacci generátor SW realizációja (pszeudó kód!)

- **Akkumulátor alapú adatstruktúra**
- **Program méret: 12 (kis méretű) utasításszó**
  - Utasítás formátum: ACC = ACC művelet REG1 (1 regisztercím)
- **Végrehajtás: 8 utasításonként egy új érték**

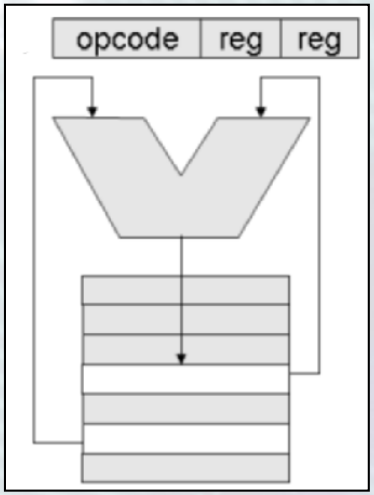



```
;*************************************************************************
;*  Fibonacci sorozat  akkumulátoros architektúrával                     *
;*  A demonstráció miatt az akkumulátor az r7 regiszter
;*  Az első néhány elemre, bináris aritmetikával                         *
;*************************************************************************
DEF LD  0x80                ; LED kimeneti regiszter          (írható/olvasható)

    CODE

;*************************************************************************
;* A program kezdete.       4 előkészítő utasítás + 8 utasítás a ciklusban
;*************************************************************************
start:
    mov     r7, #0          ; ACC beállítása 0-ra
    mov     r0, r7          ; Az F(0) elem értéke, ez lesz mindig az aktuális F(n)
    mov     r7, #1          ; ACC beállítása 0-ra
    mov     r1, r7          ; Az F(1) elem értéke, ez lesz a következő elem F(n+1)
loop:
    mov     r7, r0          ; Aktuális elem másolása ACC-ba
    mov     LD, r7          ; Kijelezzük az aktuális elemet
    mov     r2, r7          ; Aktuális elem átmeneti tárolása
    mov     r7, r1          ; Aktuális frissítése következő elemmel F(n+1)->F(n)
    mov     r0, r7          ; és mentése ACC-ból
    add     r7, r2          ; Következő új értékének számítása F(n+1)+F(n)->F(n+1)
    mov     r1, r7          ; Következő elem átmásolása ACC-ból
    jmp     loop            ; Iteráció ismétlése frissített F(n), F(n+1) elemekkel
```

2 regisztercímes adatstruktúra

Fibonacci generátor SW realizációja (pszeudó kód!)

- **2 címes regisztertömb alapú adatstruktúra**
- **Program méret: 7 (közepes méretű) utasításszó**
  - Utasítás formátum: REG1 = REG1 műv. REG2 (2 reg. cím)
- **Végrehajtás: 5 utasításonként egy új érték**

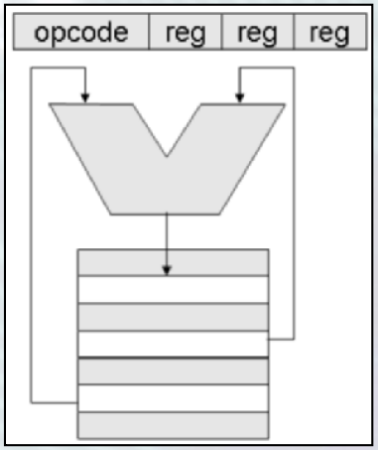



```
;*************************************************************************
;*  Fibonacci sorozat 2R architektúrával                                 *
;*  Az első néhány elemre, bináris aritmetikával                         *
;*************************************************************************
DEF LD   0x80                ; LED kimeneti regiszter          (írható/olvasható)

    CODE

;*************************************************************************
;* A program kezdete.       2 előkészítő utasítás + 5 utasítás a ciklusban
;*************************************************************************
start:
    mov     r0, #0           ; Az F(0) elem értéke, ez lesz mindig az aktuális F(n)
    mov     r1, #1           ; Az F(1) elem értéke, ez lesz a következő elem F(n+1)
loop:
    mov     LD, r0           ; Kijelezzük az aktuális elemet
    mov     r2, r0           ; Aktuális elem átmeneti tárolása
    mov     r0, r1           ; Aktuális frissítése következő elemmel F(n+1)->F(n)
    add     r1, r2           ; Következő új értékének számítása F(n+1)+F(n)->F(n+1)
    jmp     loop             ; Iteráció ismétlése frissített F(n), F(n+1) elemekkel
```

3 regisztercímes adatstruktúra

Fibonacci generátor SW realizációja (pszeudó kód!)

- **3 címes regisztertömb alapú adatstruktúra**
- **Program méret: 7 (nagyméretű) utasításszó**
  - Utasítás formátum: REG3 = REG1 műv. REG2 (3 reg. cím)
- **Végrehajtás: 5 utasításonként egy új érték**

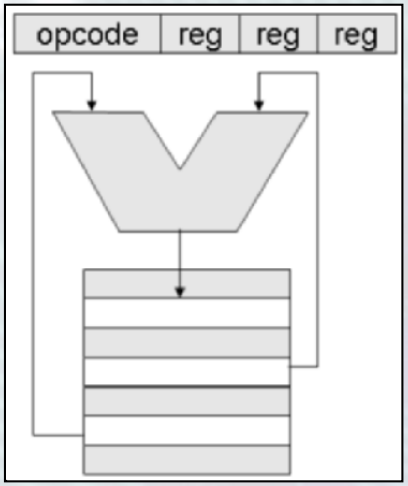



```
;*************************************************************************
;*  Fibonacci sorozat 3R architektúrával                                 *
;*  Az első néhány elemre, bináris aritmetikával                         *
;*************************************************************************
DEF LD  0x80                ; LED kimeneti regiszter          (írható/olvasható)

    CODE

;*************************************************************************
;* A program kezdete.       2 előkészítő utasítás + 5 utasítás a ciklusban
;*************************************************************************
start:
    mov     r0, #0          ; Az F(0) elem értéke, ez lesz mindig az aktuális F(n)
    mov     r1, #1          ; Az F(1) elem értéke, ez lesz a következő elem F(n+1)
loop:
    mov     LD, r0          ; Kijelezzük az aktuális elemet
    add     r2, r0, r1      ; Közvetlenül számoljuk a köv. értéket F(n+1)+F(n)->F(n+2)
    mov     r0, r1          ; Aktuális frissítése következő elemmel F(n+1)->F(n)
    mov     r1, r2          ; Következő új érték frissítése F(n+2) -> F(n+1)
    jmp     loop            ; Iteráció ismétlése frissített F(n), F(n+1) elemekkel
```

3 regisztercímes adatstruktúra

Fibonacci generátor SW realizációja (pszeudó kód!)

- 3 címes regisztertömb alapú adatstruktúra
- Hatékonyabb programkialakítás
- Program méret: 9 (nagyméretű) utasításszó
  - Utasítás formátum: REG3 = REG1 műv. REG2 (3 reg. cím)
- Végrehajtás: kb. 2 utasításonként egy új érték (!)




```
;*************************************************************************
;*  Fibonacci sorozat 3R architektúrával, ciklus kiterítéses lineáris kódolással
;*  Az első néhány elemre, bináris aritmetikával                          *
;*************************************************************************
DEF LD  0x80                ; LED kimeneti regiszter          (írható/olvasható)

    CODE

;*************************************************************************
;* A program kezdete.       2 előkészítés + 7 utasítás a ciklusban 3 elemre    *
;*************************************************************************
start:
    mov     r0, #0          ; Az F(0) elem értéke, ez lesz mindig az aktuális F(n)
    mov     r1, #1          ; Az F(1) elem értéke, ez lesz a következő elem F(n+1)
loop:
    mov     LD, r0          ; Kijelezzük az aktuális elemet r0-ból
    add     r2, r0, r1      ; Közv. számoljuk a köv. értéket r2-be F(n+1)+F(n)->F(n+2)
    mov     LD, r1          ; Kijelezzük az aktuális elemet r1-ből
    add     r0, r1, r2      ; Közv. számoljuk a köv. értéket r0-ba F(n+2)+F(n+1)->F(n+3)
    mov     LD, r2          ; Kijelezzük az aktuális elemet r2-ből
    add     r1, r2, r0      ; Közv. számoljuk a köv. értéket r1-be F(n+3)+F(n+2)->F(n+4)
    jmp     loop            ; 3-as iteráció ismétlése
```

Processzor adatstruktúrák

- **A vizsgálatok alapján egy adott mintafeladat különböző feltételekkel realizálható**
- **A megvalósítás fontos jellemzője (költsége): a program teljes memória igénye („memory footprint”)**
  - Asztali vagy nagygépes környezetben nem kritikus
    - De persze nem elhanyagolható
  - Beágyazott vagy mobil környezetben ma is fontos!
    - Pl. Internet of Things (IoT) eszközök memória korlátja
- **A Fibonacci sorozat generátor példára, átlagos adatokkal:**
  - 6 bit utasításkód (kb. 50 utasítás), 4 bites reg. cím (16 reg.)
  - STACK: 6 ut. * 6 bit = 36 programmemória bit
  - ACCU: 12 ut. * (6+4) bit = 120 programmemória bit
  - 2REG: 7 ut. * (6+4+4) bit = 98 programmemória bit
  - 3REG: 7 ut. * (6+4+4+4) bit = 126 programmemória bit