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

Rendszerarchitektúrák

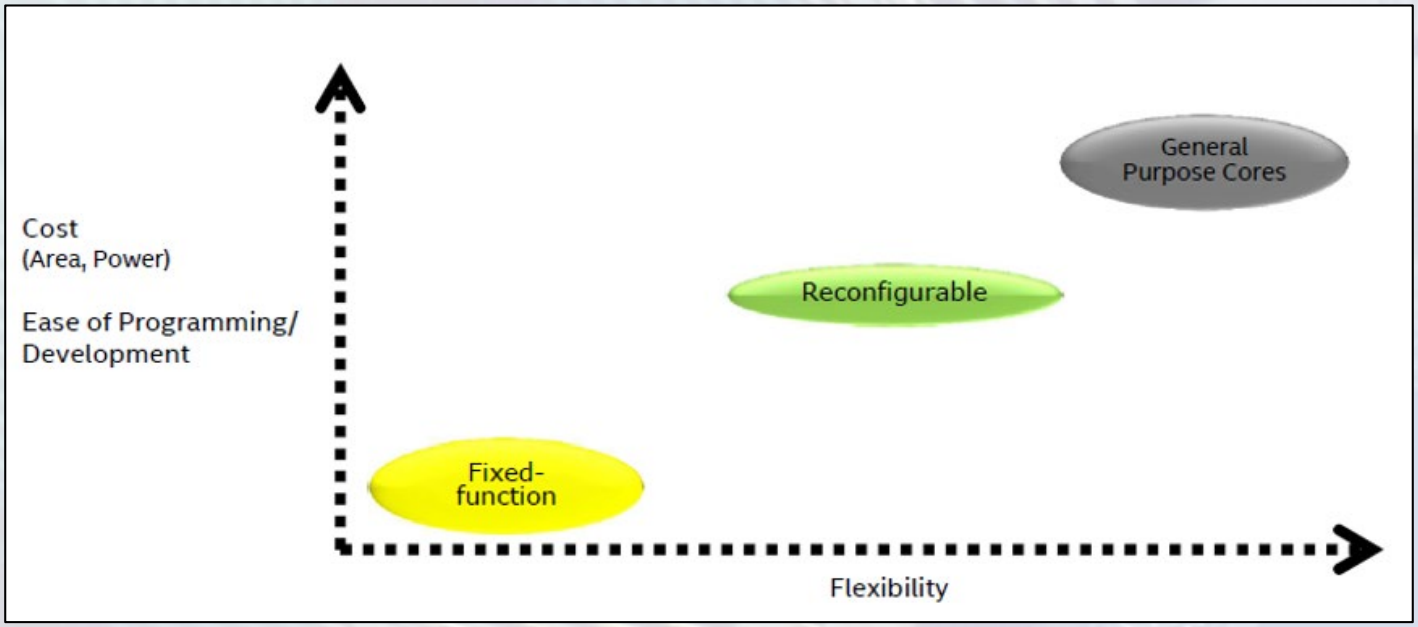
Adatfeldolgozó architektúrák vizsgálata

2021. tavaszi félév

Szántó Péter
Raikovich Tamás

Technológiai áttekintés

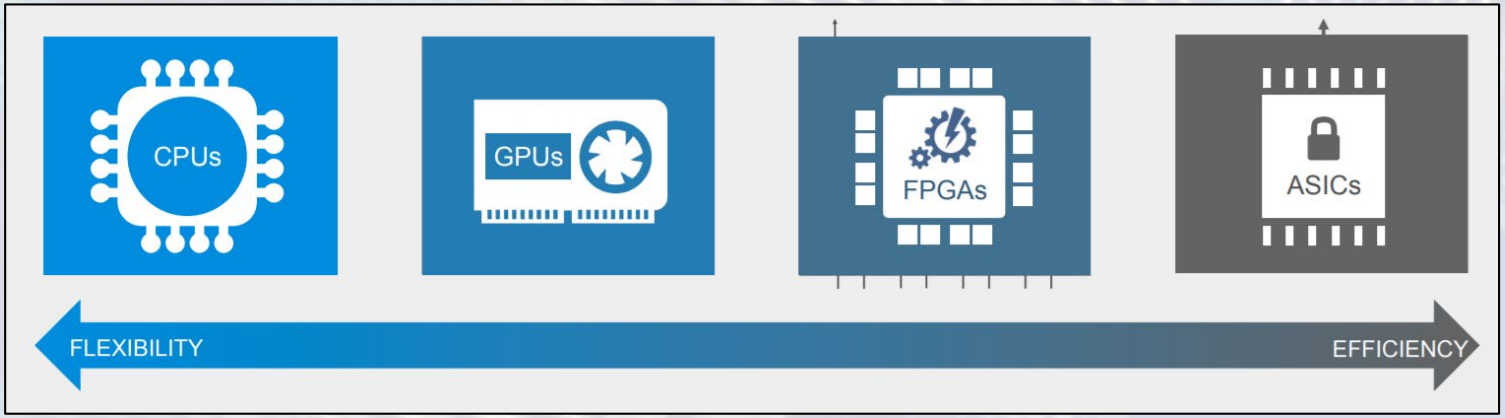
- **Félvezető technológia lehetőségei, korlátai**
- **Rendszer megvalósítási lehetőségek**
  - Egyedi dedikált megoldás
  - Általános célú megoldás



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



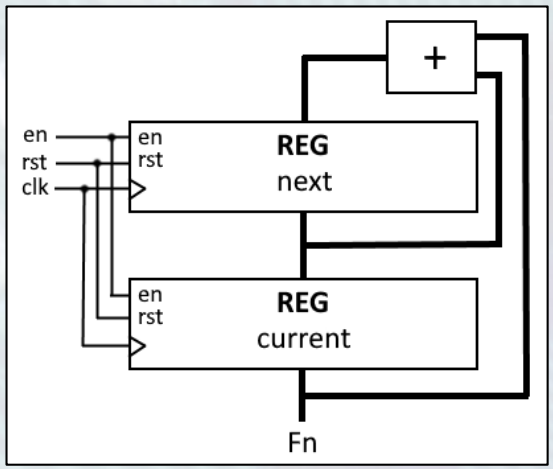
Példa feladatok: Fibonacci sorozat

- **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**

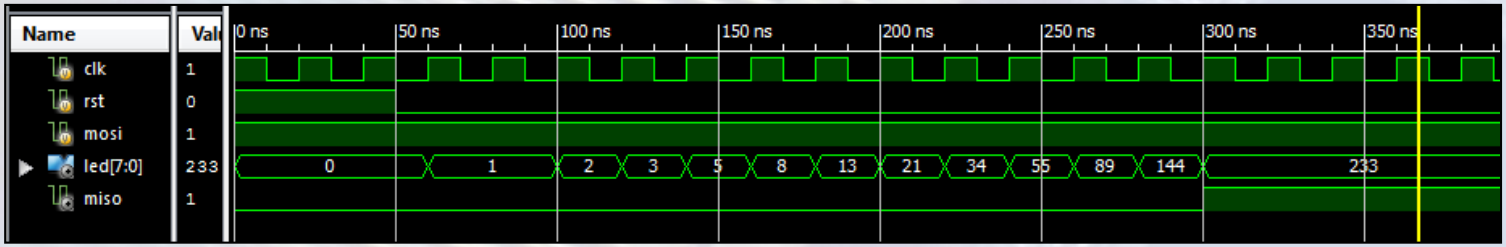
Példa feladatok: Fibonacci sorozat

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.

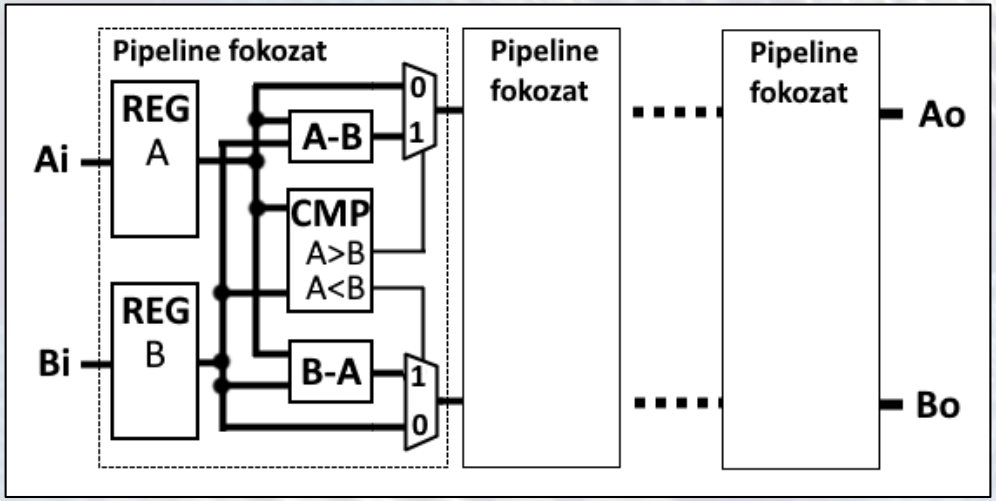
Példa feladatok: LNKO (GCD)

- **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

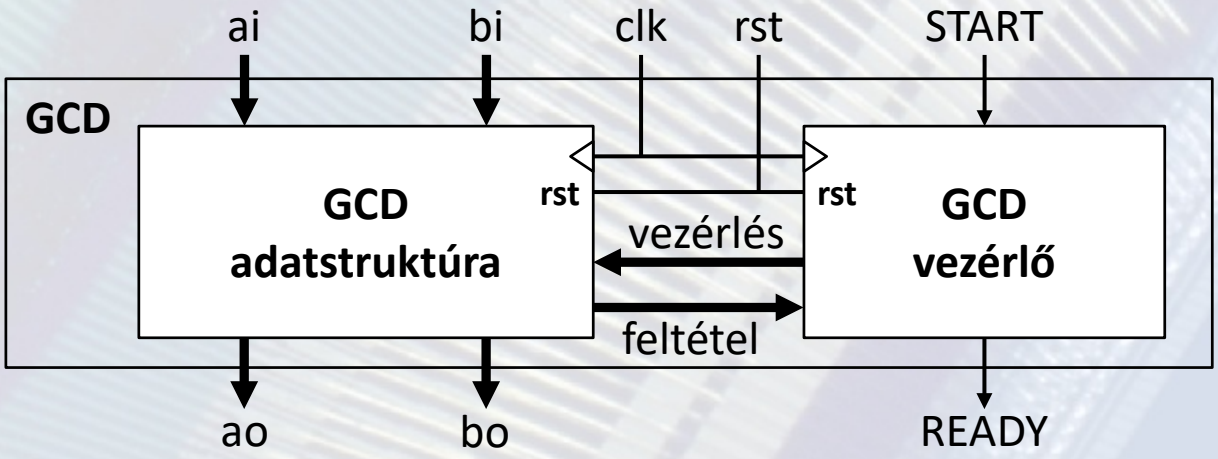
- **1 fokozat egy részszámítást végez el**
- **Minden órajelciklusban új bemenet és eredmény**
- **Fokozatok száma: N bites adatok esetén 2^N**
  - A legrosszabb esetre kell felkészülni
- **Gyors, de nagy erőforrás igényű**



Példa feladatok: LNKO (GCD)

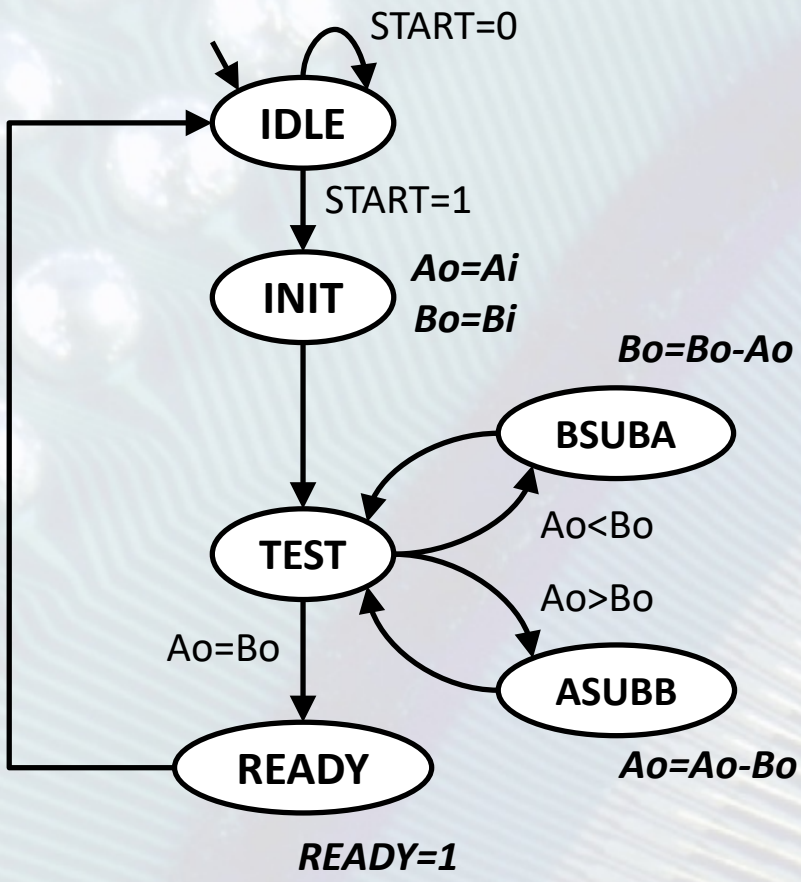
Soros végrehajtású HW megvalósítás

- **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

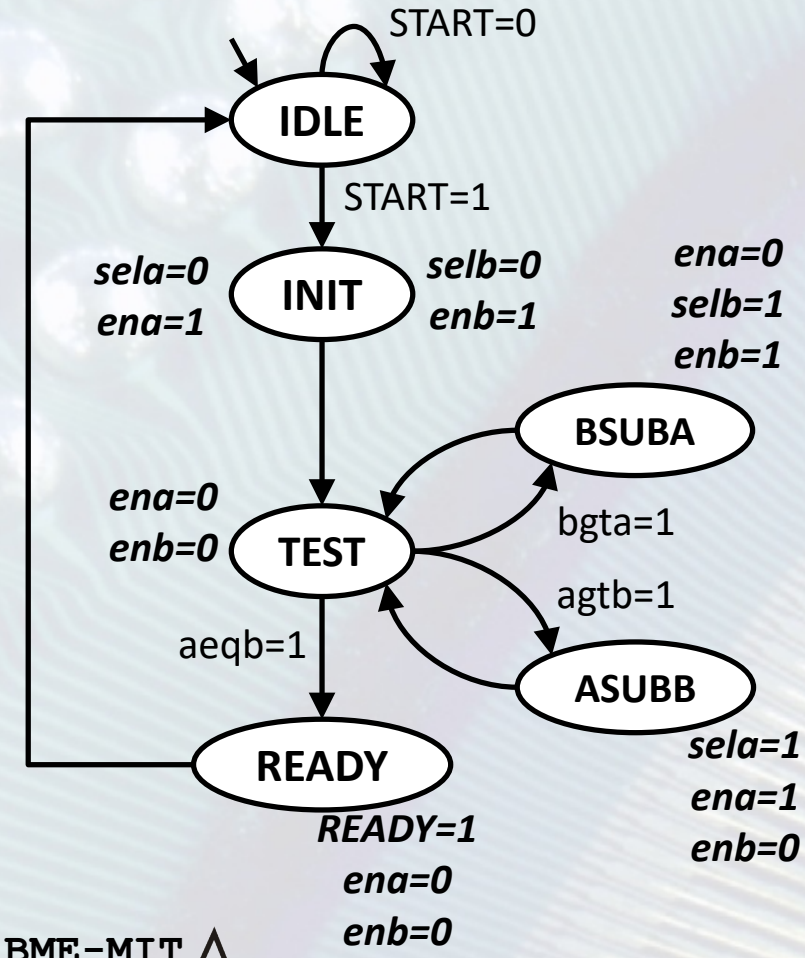
- Magasszintű műveletek megvalósítása
- Vezérlő jelek (←FSM)
- Feltétel jelek (→FSM)

Vezérlő

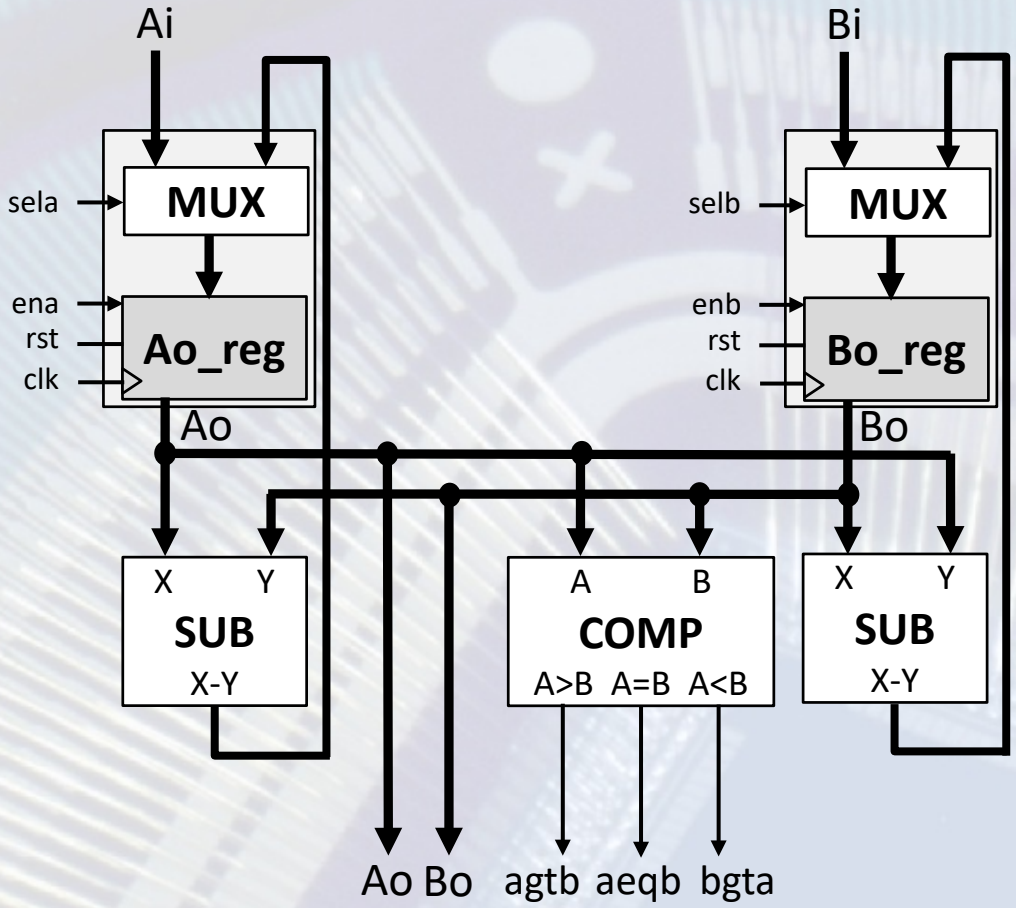
- Véges automata (FSM)
- Bitszintű műveletek
- Külső parancs és státusz jelek

Példa feladatok: LNKO (GCD)

A vezérlő állapotdiagramja

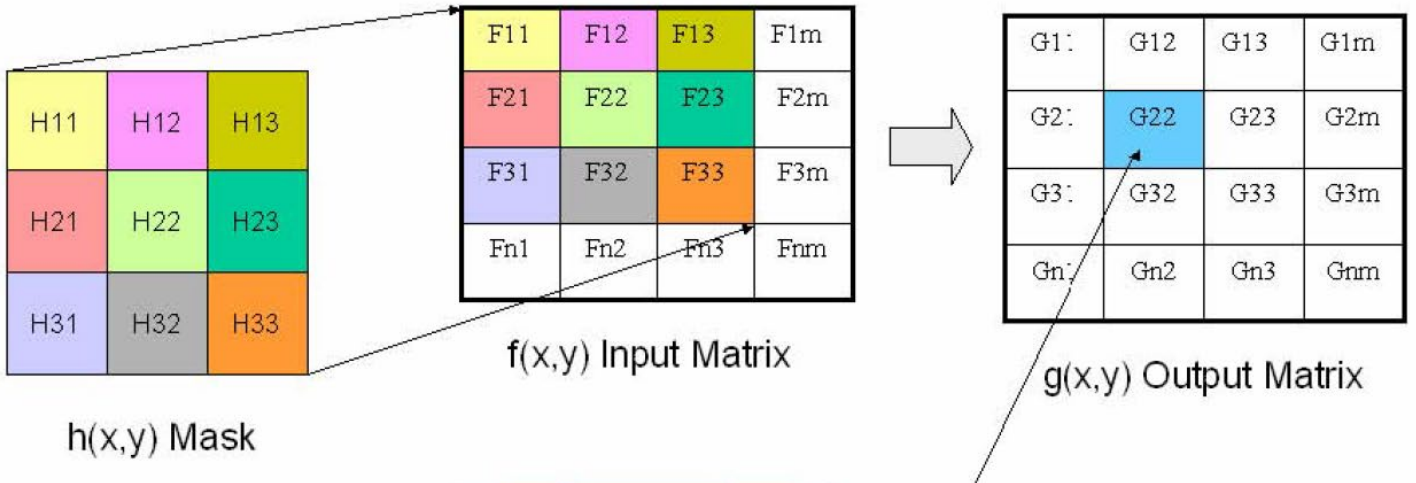


Az adatstruktúra egységei



Példa feladatok: 2D konvolúció

$$g(x,y) = \sum_{i=-1}^{i=1} \sum_{k=-1}^{k=1} h(i,k) f(x-i, y-k)$$

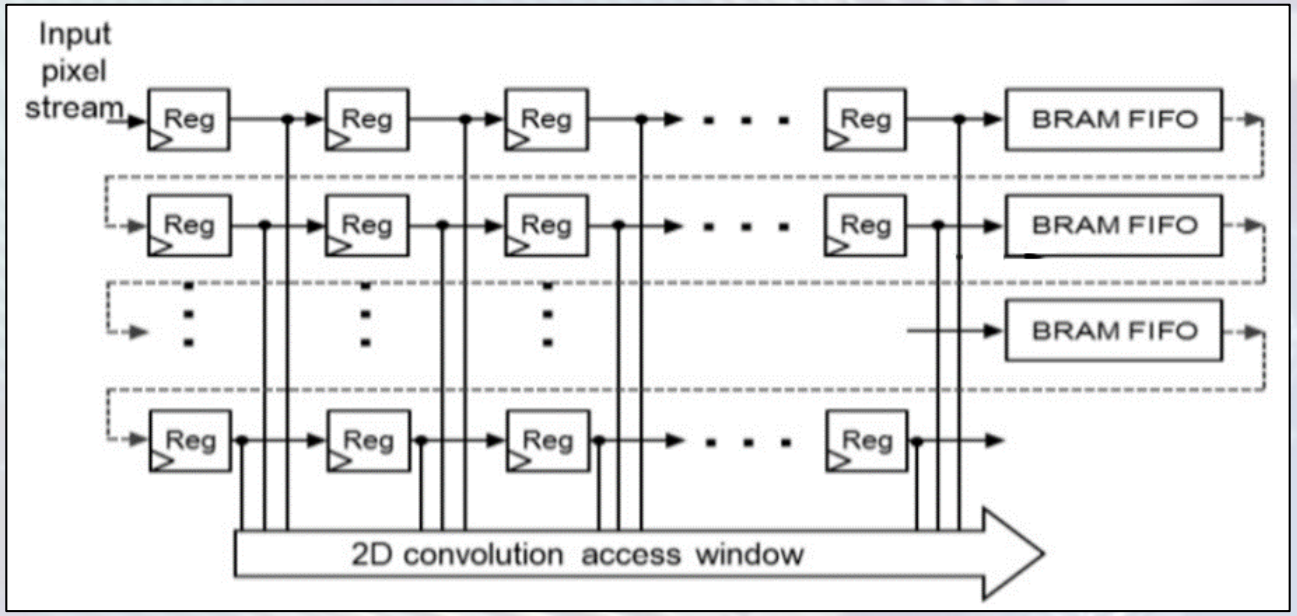


Computations required: 9 multiplies and 8 accumulates required for each output point

H11*F11 + H12*F12 + H13*F13 + H21*F21 + H22*F22 + H23*F23 + H31*F31 + H32*F32 + H33*F33

Példa feladatok: 2D konvolúció

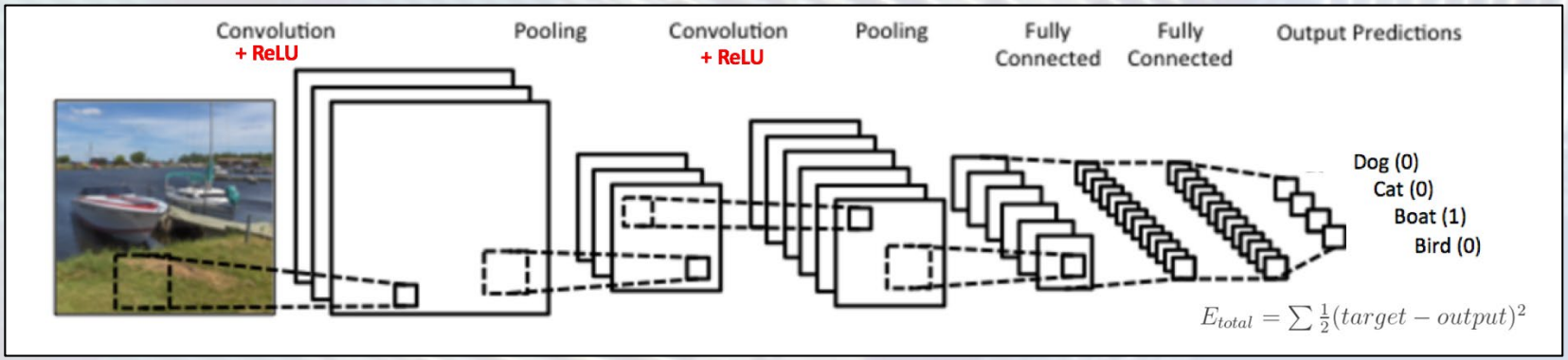
- Jelentős aritmetikai műveletigény
- Szorzás és összeadás (Multiply and Accumulate, MAC)
- A DSP (Digital Signal Processzor) eszközök specialitása
- Közvetlen HW megvalósítás: sorpufferek + N^2 MAC



Példa feladatok: 2D konvolúció

2D konvolúció az AI/ML alkalmazásokban

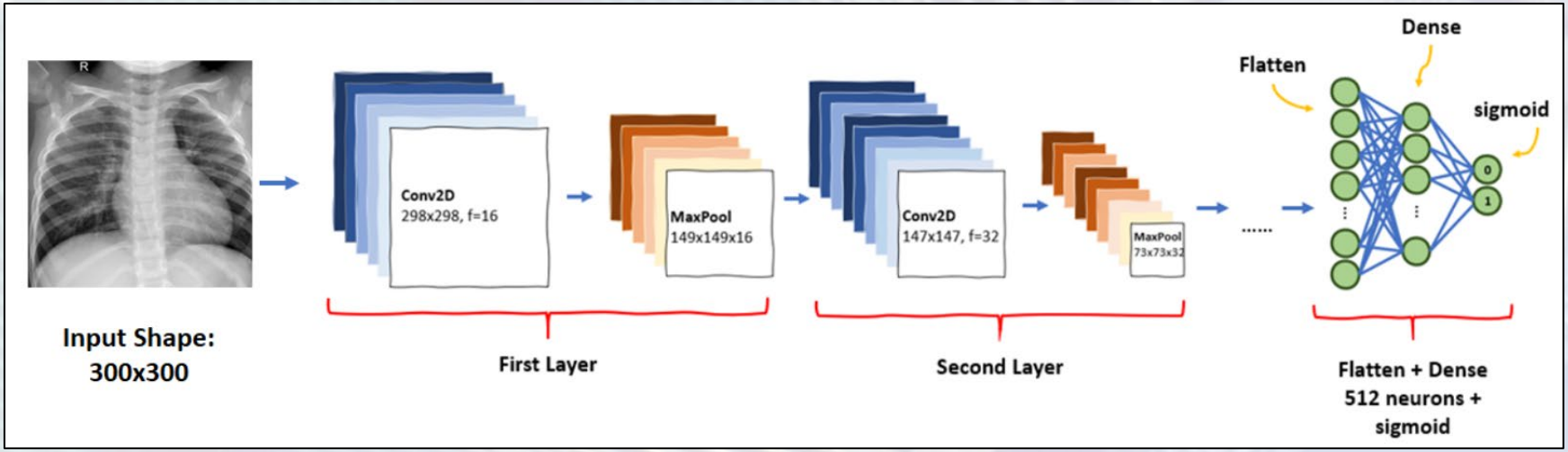
- **Konvolúciós neurális hálózatok (CNN)**
- **Nagyon jelentős aritmetikai műveletigény**
  - Nagyméretű forrásadatok
  - Jelentős számú együttható
  - Extra méretű részeredmények
- **Hatékony megvalósításuk gondos tervezést igényel**



Példa feladatok: 2D konvolúció

Tüdőgyulladás detektálás CNN-nel

- **3 darab 2D konvolúciós réteg, 3x3-as ablak**
  - Minden réteg negyedeli a pixelek számát
  - Szűrők: 1. rétegen 16, 2. rétegen 32, 3. rétegen 64
- **Szükséges MAC műveletek száma: kb. 22 millió (konv.)**



https://towardsdatascience.com/chest-x-rays-pneumonia-detection-using-convolutional-neural-network-63d6ec2d1dee

Példa feladatok: 2D konvolúció

Egyéb példák a szükséges MAC műveletek számára:

Metrics	LeNet 5	AlexNet	Overfeat fast	VGG 16	GoogLeNet v1	ResNet 50
Top-5 error†	n/a	16.4	14.2	7.4	6.7	5.3
Top-5 error (single crop)†	n/a	19.8	17.0	8.8	10.7	7.0
Input Size	28×28	227×227	231×231	224×224	224×224	224×224
# of CONV Layers	2	5	5	13	57	53
Depth in # of CONV Layers	2	5	5	13	21	49
Filter Sizes	5	3,5,11	3,5,11	3	1,3,5,7	1,3,7
# of Channels	1, 20	3-256	3-1024	3-512	3-832	3-2048
# of Filters	20, 50	96-384	96-1024	64-512	16-384	64-2048
Stride	1	1,4	1,4	1	1,2	1,2
Weights	2.6k	2.3M	16M	14.7M	6.0M	23.5M
MACs	283k	666M	2.67G	15.3G	1.43G	3.86G
# of FC Layers	2	3	3	3	1	1
Filter Sizes	1,4	1,6	1,6,12	1,7	1	1
# of Channels	50, 500	256-4096	1024-4096	512-4096	1024	2048
# of Filters	10, 500	1000-4096	1000-4096	1000-4096	1000	1000
Weights	58k	58.6M	130M	124M	1M	2M
MACs	58k	58.6M	130M	124M	1M	2M
Total Weights	60k	61M	146M	138M	7M	25.5M
Total MACs	341k	724M	2.8G	15.5G	1.43G	3.9G
Pretrained Model Website	[56]‡	[57, 58]	n/a	[57–59]	[57–59]	[57–59]

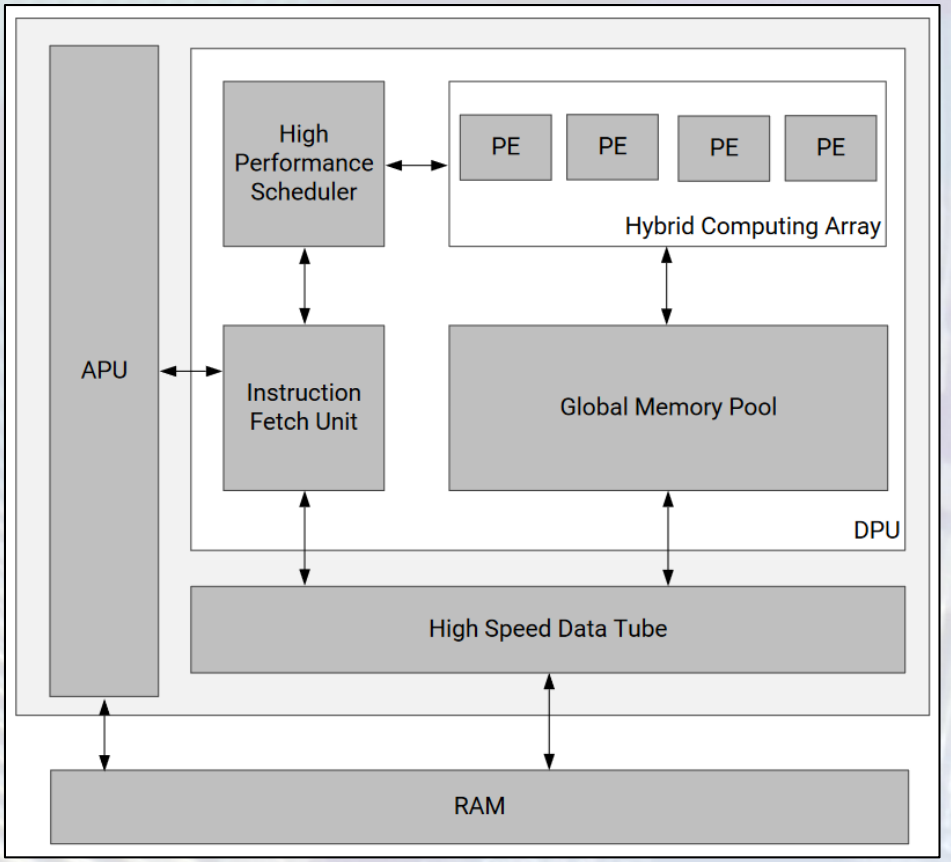
Példa feladatok: 2D konvolúció

- **Dedikált hardveres megvalósítás: FPGA, ASIC**
  - Teljesen párhuzamos: pipeline
    - Konstans együtthatók → egy adott feladatra alkalmas
    - Nagyon nagy erőforrásigény
  - Nem teljesen párhuzamos
    - Együttható ROM-ok és MAC egységek
    - Együttható RAM-ok és MAC egységek → konfigurálható
- **Programozható hardveres megvalósítás**
  - Xilinx DPU core
  - Xilinx Versal AI core
  - Grafikus processzor (GPU)

Példa feladatok: 2D konvolúció

Xilinx Deep Learning Processor Unit (DPU)

- Szintetizálható hardver gyorsító konvolúciós neurális hálózatokhoz
- Programozható
- Dokumentáció: PG338

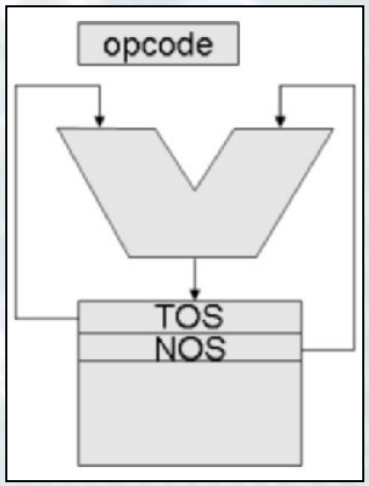


Processzor adatstruktúrák

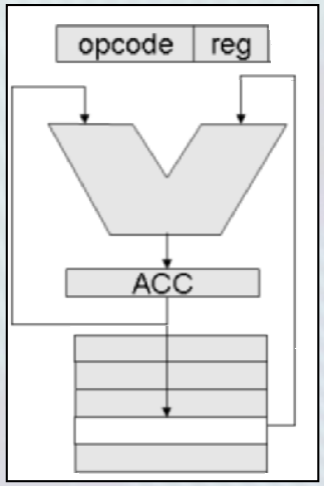
- **Előző példák: dedikált műveletvégző egységek**
- **Általános célú műveletvégző: programozható CPU**
  - (Általános célú) adatstruktúra + vezérlő
  - Adatstruktúra
    - Adatmozgatás
    - Aritmetikai: összeadás, kivonás (szorzás, osztás)
    - Logikai: bitenkénti AND, OR, XOR
    - Léptetés, forgatás
    - 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

Általános adatstruktúrák

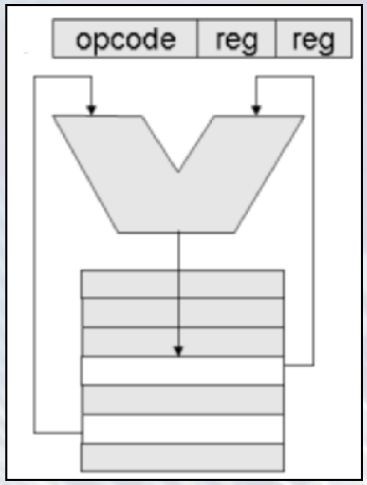
STACK



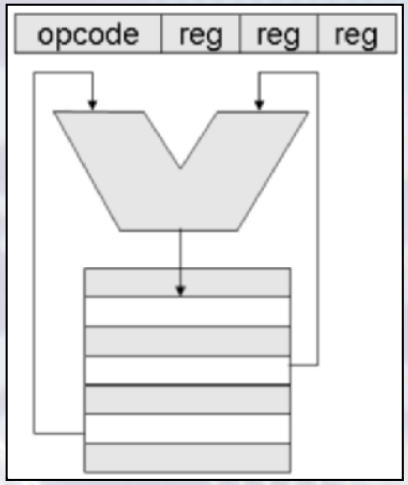
ACC



2REG



3REG

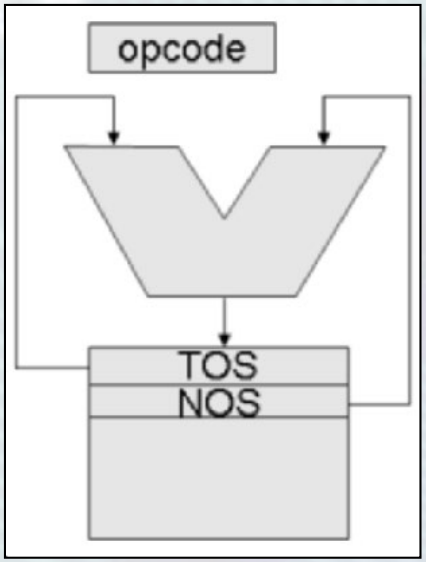


- **STACK: forrás a legfelső 2 adat, cél a verem teteje**
- **ACC: az egyik forrás és a cél mindig az akkumulátor**
- **2REG: az egyik forrás és a cél operandus azonos**
- **3REG: külön forrás és cél operandusok**

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**

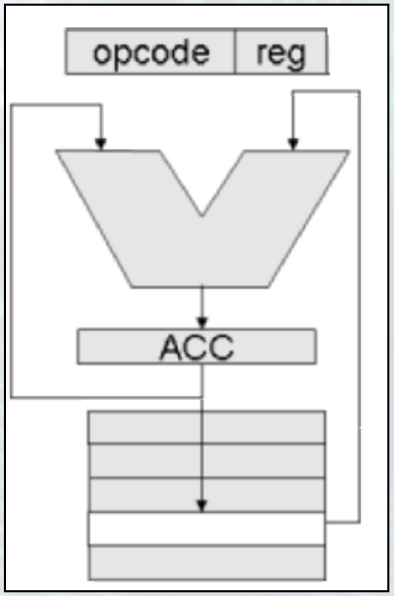



```
;*****************************************************************************
;*  Fibonacci sorozat stack alapú architektúrával PSEUDÓKÓD                  *
;*  Az első néhány elemre, bináris aritmetikával                             *
;*****************************************************************************
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**

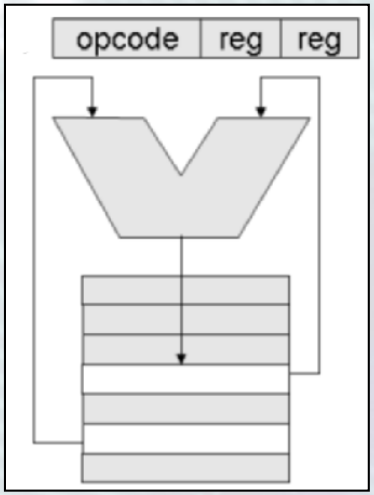



```
;*************************************************************************
;*  Fibonacci sorozat  akkumulátoros architektúrával                     *
;*  A demonstráció miatt az akkumulátor az r7 regiszter
;*  Az első néhány elemre, bináris aritmetikával                         *
;*************************************************************************
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**

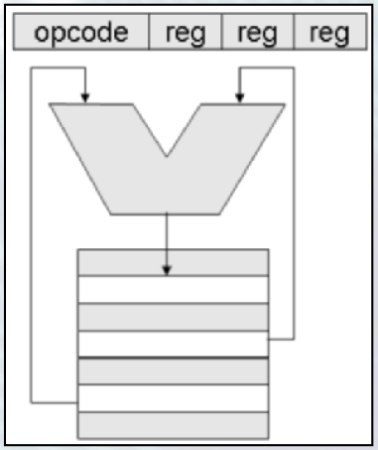



```
;*************************************************************************
;*  Fibonacci sorozat 2R architektúrával                                 *
;*  Az első néhány elemre, bináris aritmetikával                         *
;*************************************************************************
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**

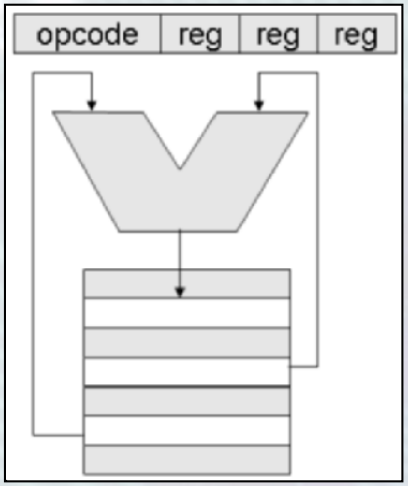



```
;*************************************************************************
;*  Fibonacci sorozat 3R architektúrával                                 *
;*  Az első néhány elemre, bináris aritmetikával                         *
;*************************************************************************
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