PANNON EGYETEM, Veszprém
Villamosmérnöki és Információs Rendszerek Tanszék

Digitális Rendszerek és Számítógép Architektúrák
(Levelező BSc)

3. előadás: Utasítás végrehajtás folyamata: címzési módok, RISC-CISC processzorok

Előadó: Dr. Vörösházi Zsolt
voroshazi.zsolt@mik.uni-pannon.hu

Jegyzetek, segédanyagok:

- Könyvfejezetek:
  - http://www.virt.uni-pannon.hu
    → Oktatás → Tantárgyak → Digitális Rendszerek és Számítógép Architektúrák (Kiegészítő Levelező)
    (chapter04.pdf)
- Fóliák, óravázlatok .ppt (.pdf)
- Frissítésük folyamatosan

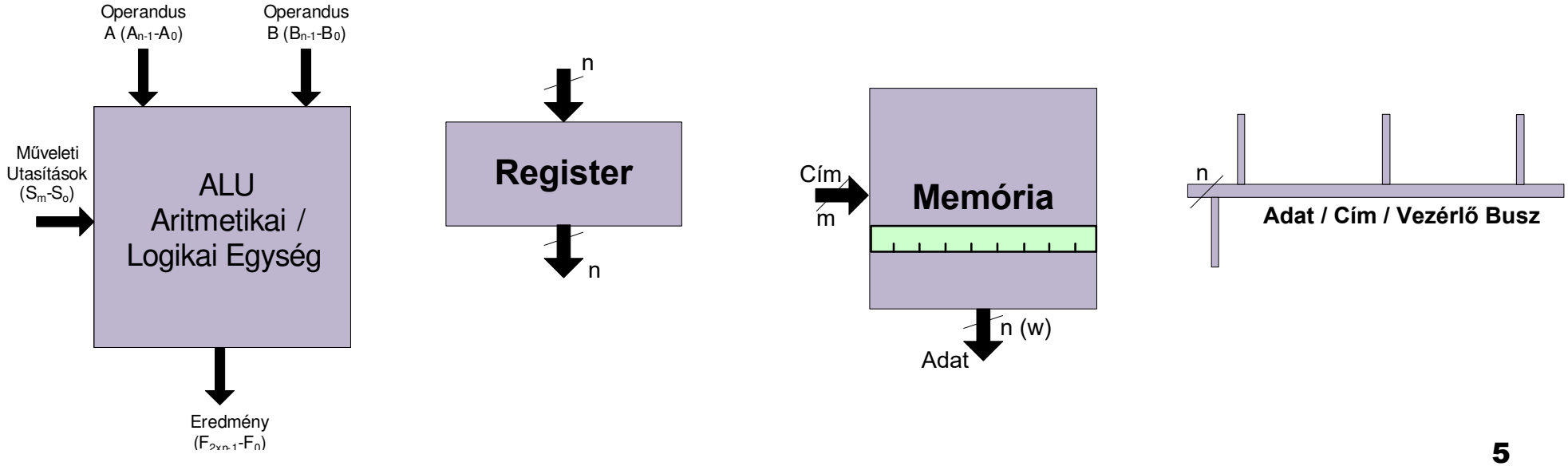
Utasítás végrehajtás folyamata

- Utasítás kódok
  - Programvezérlő utasítások
- Címzési módok
- RISC vs. CISC processzor architektúrák

Alapvető digitális építőelemek (rövid áttekintés)

Legfontosabb digitális építőelemeink:

- ALU
- Memóriák
- Adat / Cím / Vezérlő Buszok
- Regiszterek, De/Multiplexerek, De/Kódoló áramkörök



ALU egység

- Az **ALU** egység két különböző *n*-bites bemenettel (A, B) rendelkezik, és egy *n*-bites** kimenettel (F). A szelektáló (S) jelek segítenek a megfelelő műveletek kiválasztásában. Az ALU egység egy algoritmus utasításainak megfelelően aritmetikai ill. logikai műveleteket hajt végre.
- Eml: funkcionális teljesség, +,-,*,/ és Logikai fgv.
- (Korábban részletesen: chapter_03.pdf)

- ** eredmény valójában *n+1*, vagy *2*n* bites

Memória egységek

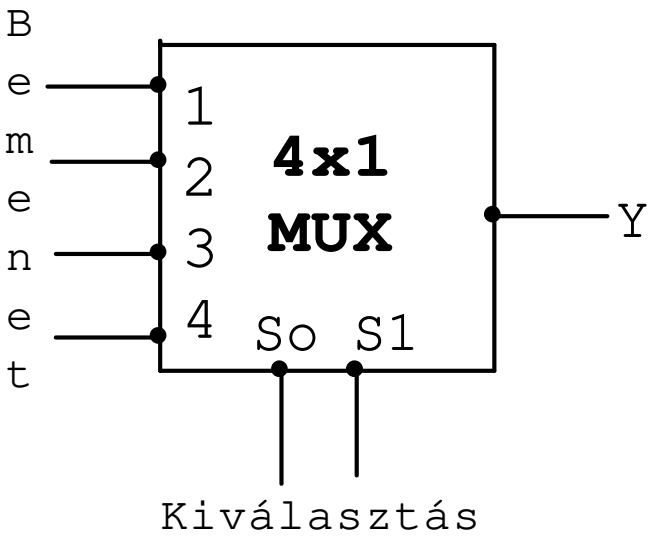
- Az ALU által kezelt / végrehajtott adatok a **memóriában** (tároló rekeszek lineáris tömbjében) tárolódnak el. A memória rekeszei általában olyan szélesek, amilyen széles az adatbusz. Például, legyen *n-bit (w)* széles, és álljon *2^m számú* rekeszből. Ekkor *m* számú címvezetékkel címezhető meg. Az adatbuszon kétirányú (írás/olvasás) kommunikáció is megengedett.
- Memória a *Neumann* architektúrát követi: tehát az utasítások (program/kód) és az adatok egy helyen tárolódnak, nem pedig külön-külön (Harvard architektúra). A programot is adatként tárolja a memória.

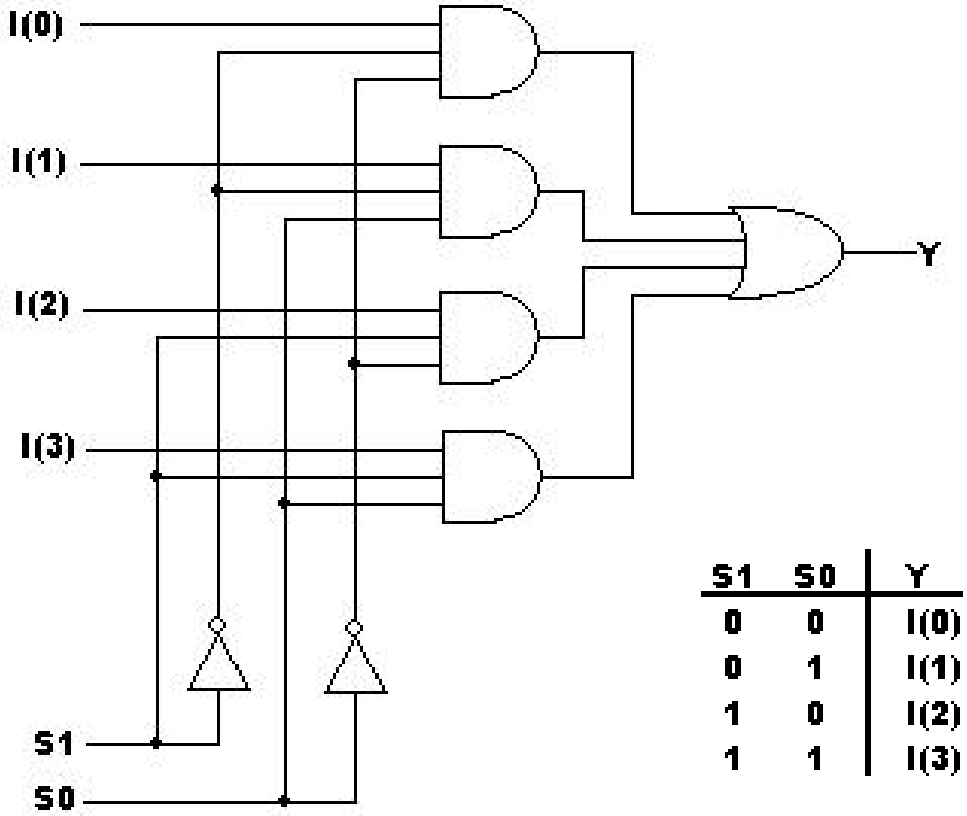
Adatbuszok – adatvonalak

- Másik alap építőelem az **adatbusz** vagy **adatút (datapath)**. Fontos paraméter a szélessége: egy *n* természetes szám.
- Az adatutak pont-pont (p2p) összeköttetéseket jelentenek különböző méretű és sebességű eszközök között.
- A közvetlen kapcsolat nagy sebességet, de egyben rugalmatlanságot is jelent a bővíthetőségben.
- Ezek az adatutak adatbuszokká szervezhetők, amivel különböző jelvezetékek információi foghatók össze.

a.) Multiplexer (MUX)

- N kiválasztó jel → 2^N bemenet, 1 kimenet
- Példa: 4:1 MUX



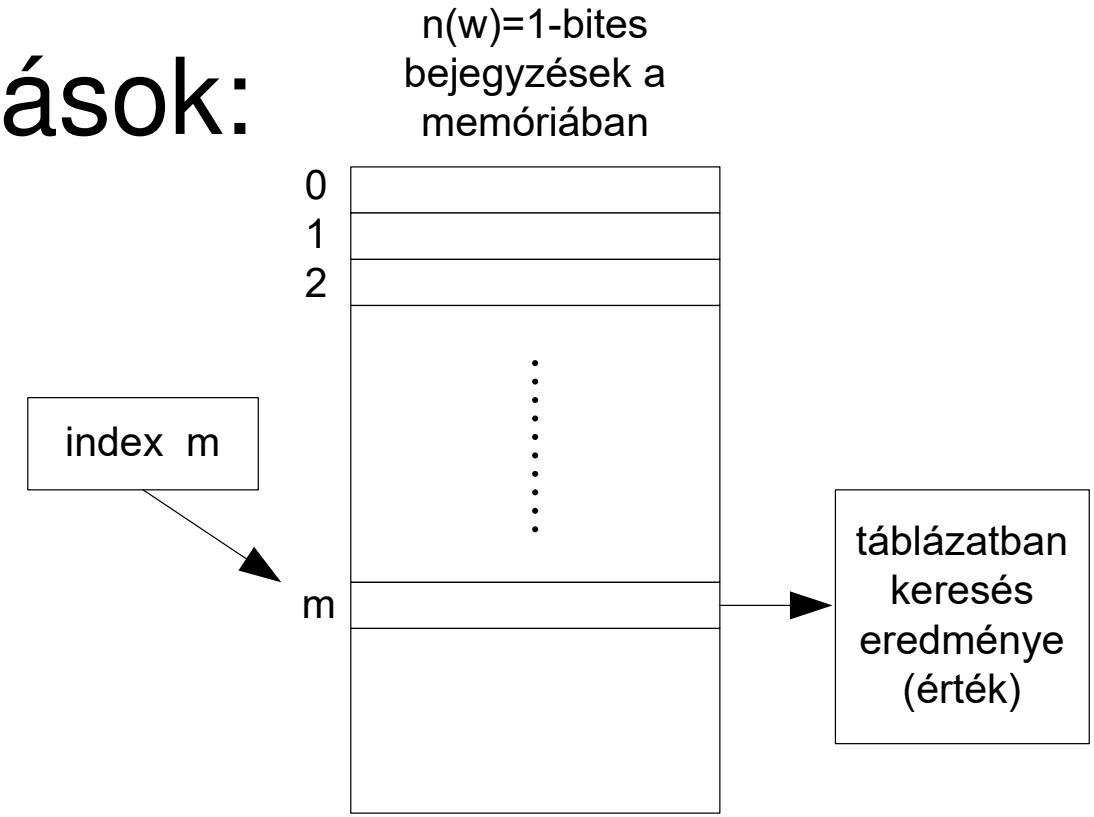


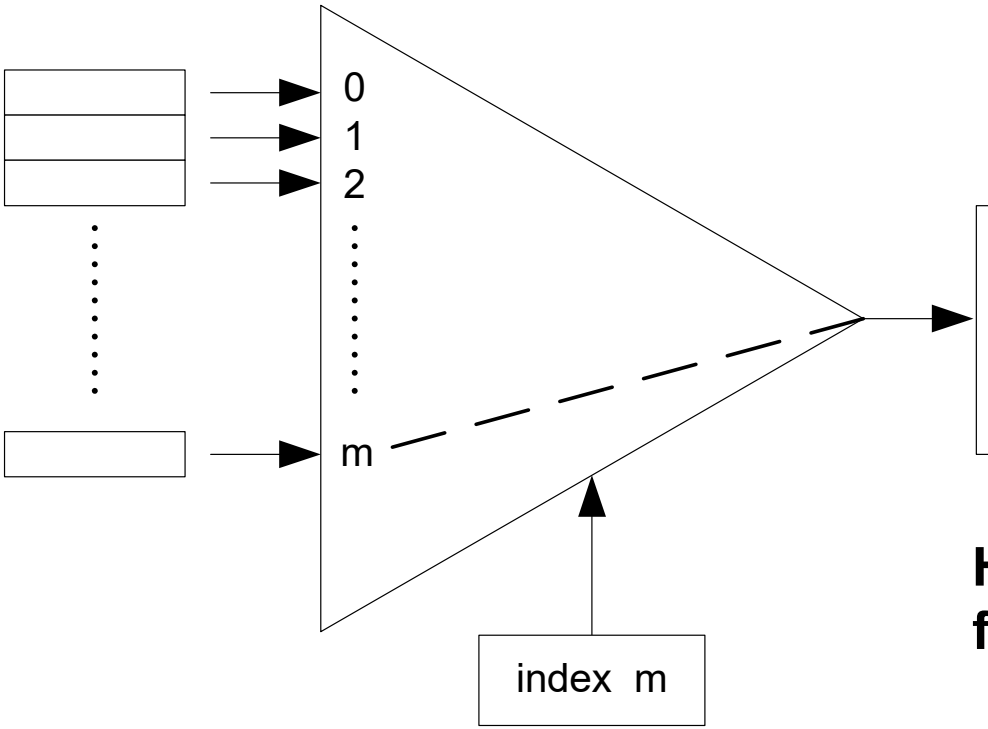
S1	S0	Y
0	0	I(0)
0	1	I(1)
1	0	I(2)
1	1	I(3)

2^N számú bemenet közül választ egyet (Y), mint egy kapcsoló. Rendelkezhet EN bemenettel is.

Pl. LUT megvalósítások:

Szoftveres Look-up-Table

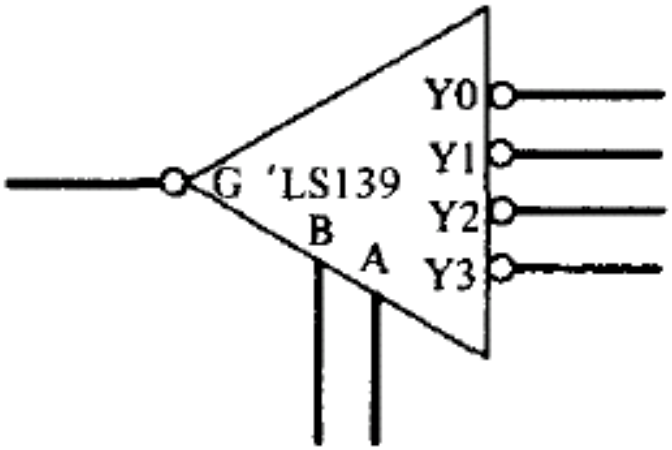




Hardveres Look-up-Table, MUX-ból felépítve (1 bites táblázatkeresés)

b.) Példa - 1:4 Demultiplexer

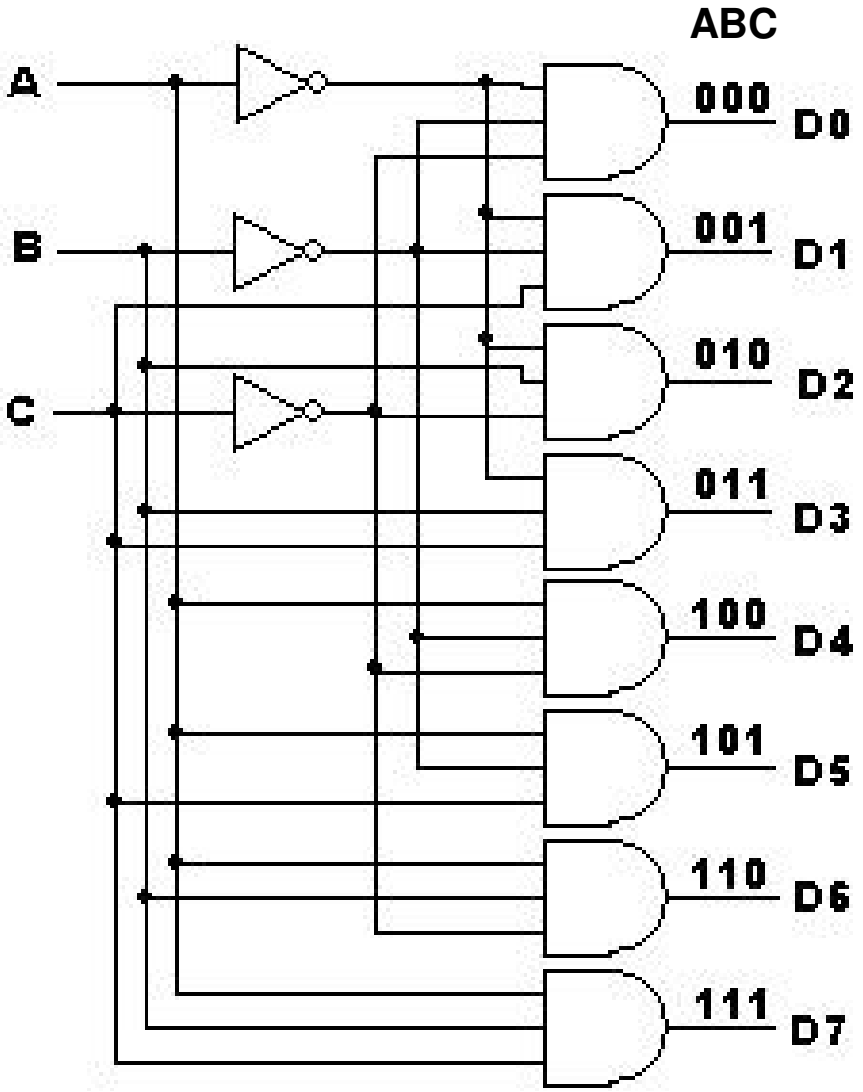
- TTL 74'LS139 duál 1:4 demultiplexer
  - Kereskedelmi forgalomban kapható
  - G: egy bemenetű
  - A,B: routing control jelek (bináris kód)
  - 4-kimenet mindegyike False, egyet kivéve, amelyik a kiválasztott (annak az értéke a bemenettől függően lehet T/F)



T=L !

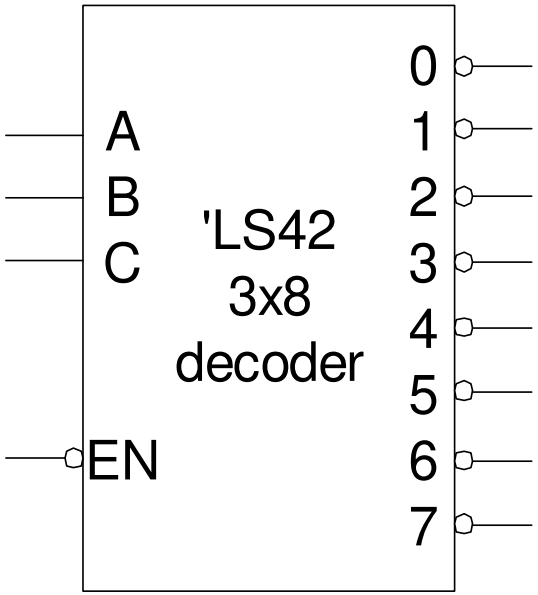
c.) Dekódoló áramkörök

- N bemenet esetén 2^N kimenete van
- Példa: 3x8 dekóder áramkör
  - Példa: Hamming-kódú hibajavító áramkör



TTL’74LS42 dekóder áramkör

- **3→8** dekóder áramkör
  - (A,B,C) 3 bemenet, (1…7) 8 kimenet
- EN: engedélyező jel, (T=L) alacsony aktív

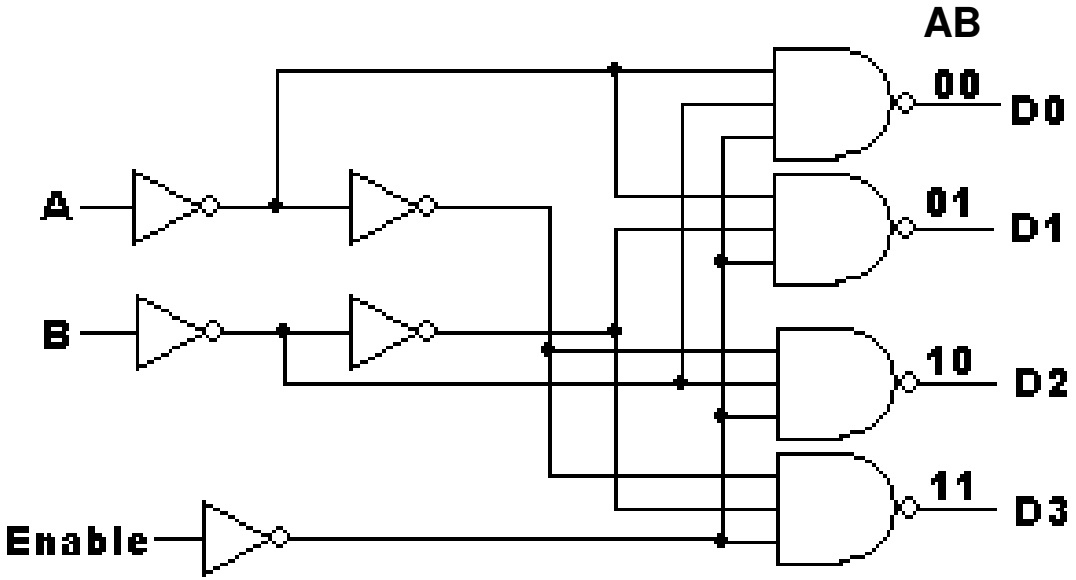


Mixed logic szimbólum

T=L!

Példa: 2x4 Dekódoló áramkör engedélyező bemenettel

- EN: alacsony aktív állapotban működik
- 2 bemenő bit (A,B)
- 4 kimenő bit (D0…D3)



En	A	B	D0	D1	D2	D3
1	x	x	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

d.) Kódoló (encoder) áramkör

- A dekódoló áramkör ellentéte: bemenetek kódolt ábrázolásának egy formája
  - ***Hagyományos encoder***: csak egy bemenete lehet igaz egyszerre
  - ***Priority encoder***: több bemenete is igaz lehet egyszerre, de azok közül a legnagyobb bináris értékű, azaz *prioritású* bemenethez generál kódot! (kód: address, index lehet)
    - I/O, IRQ jelek generálásánál használják leggyakrabban

e.) Komparátor

- Logikai kifejezés – *referencia* kifejezés (bináris számok) *aritmetikai kapcsolatának* megállapítására szolgáló eszköz.
  - Pl: Kettő n-bites szám összehasonlítása
- **compare** = **összehasonlítás**! Az azonosság eldöntéséhez a EQ/XNOR/Coincidence operátort használjuk. Jele: $A.EQ.B = A \odot B$
- n-bites minták esetén:

$$A.EQ.B = (A0 \odot B0) \cdot (A1 \odot B1) \cdot \ldots \cdot (An \odot Bn)$$

Ismétlés: EQ/XNOR/Coincidence operátor

- Logikai egyenlet: $A.EQ.B = A \odot B = A \cdot B + \overline{A} \cdot \overline{B}$
- Referenciabit szerinti megkülönböztetés:
  - ha a referencia bit (B), amihez hasonlítunk **konstans**
  - ha a referencia bit (B) egy **változó** mennyiség
- Példa: ha B referencia konstans -> egyszerűsítése A-nak

$$A.EQ.B = A \quad if \quad B = T$$

$$A.EQ.B = \overline{A} \quad if \quad B = F$$

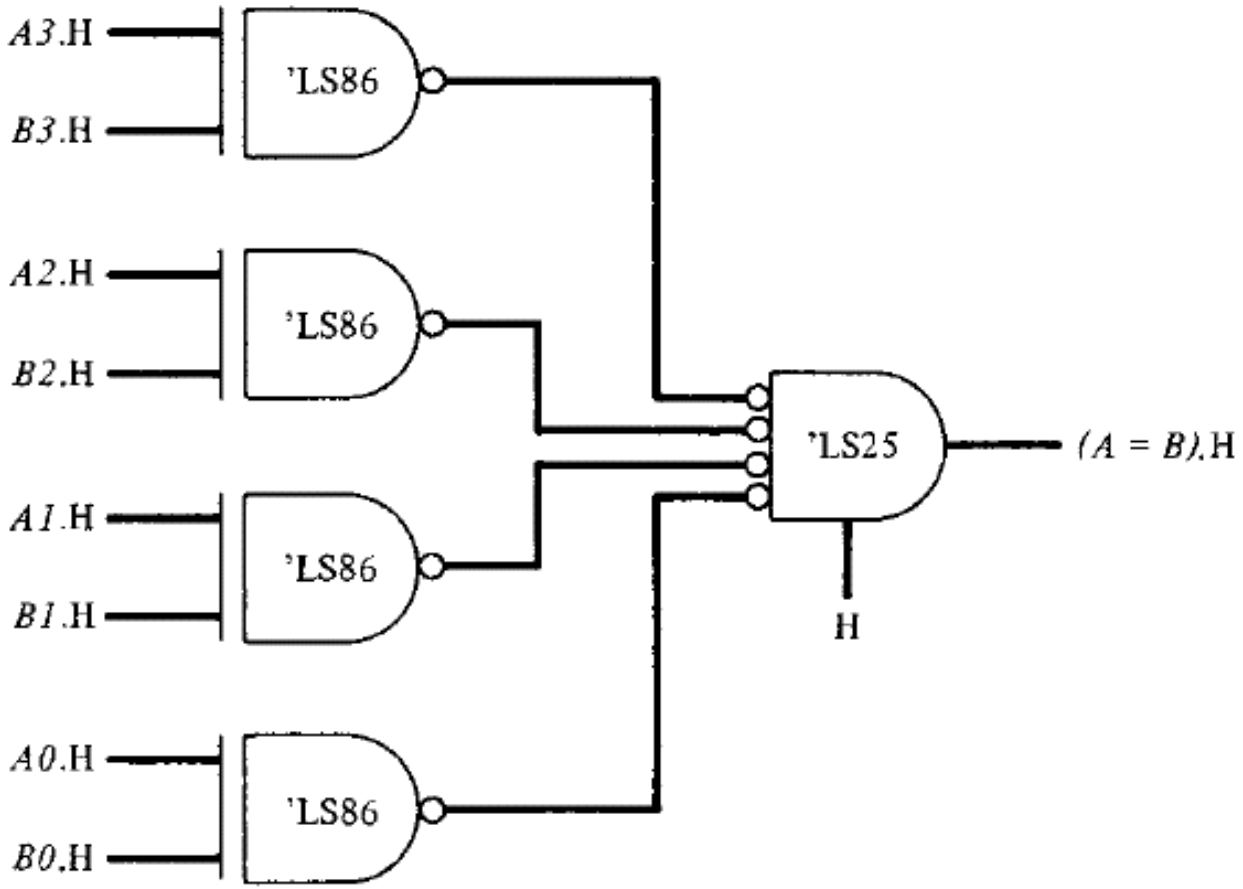
- Példa: legyen B egy 4-bites konstans mennyiség (B:=TFFT), és A tetszőleges, akkor:

$$A.EQ.B = A0 \cdot \overline{A1} \cdot \overline{A2} \cdot A3$$

Példa: 4-bites komparátor

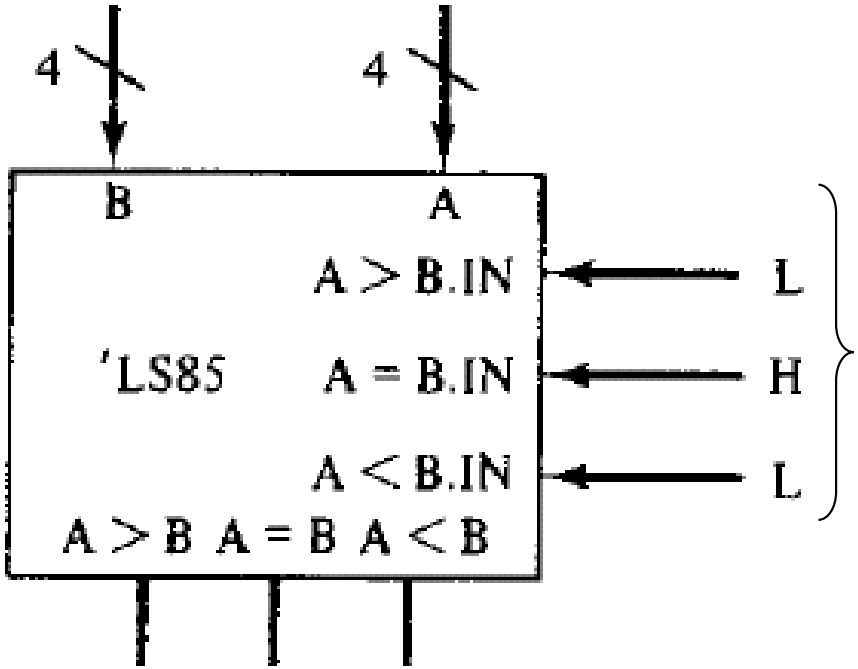
- Mixed-logic kapcsolási rajza, és log.egyenlete:

$$A.EQ.B = (A0 \odot B0) \cdot (A1 \odot B1) \cdot (A2 \odot B2) \cdot (A3 \odot B3)$$



’74LS85 4-bit Magnitude Comparator

- Magnitude comparing (~nagyságrend összehasonlítás):
  - két kifejezés **nagyságának** összehasonlítása (A<B; A=B; A>B stb.) egyszerre



3 állapot (státusz) –bemenet

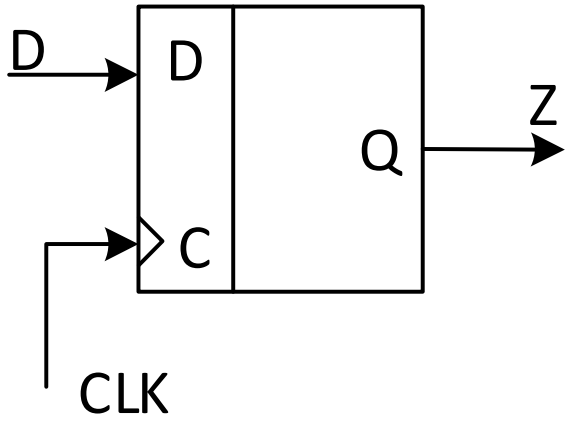
Ha B.IN=H: egyenlőség

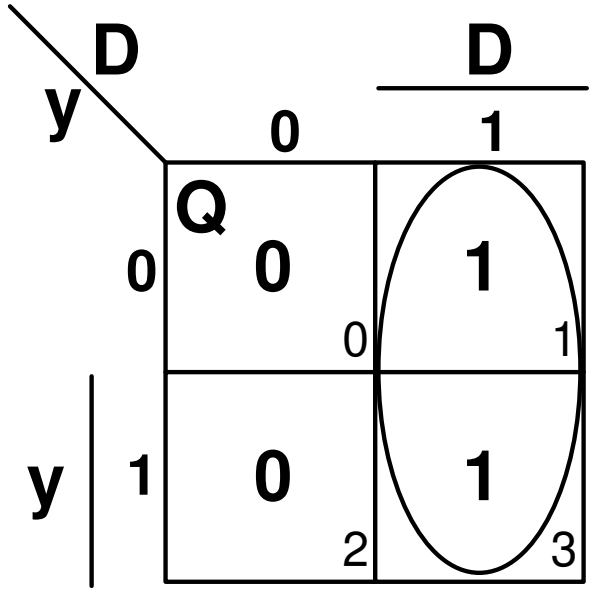
Ismétlés: Regiszterek

- A következő fontos elem a **regiszter**. Olyan szélesnek kell lennie, hogy benne, a buszokról, memóriákból, ALU-ból érkező információ eltárolható legyen. Adott vezérlőjelek hatására a bemenetén lévő adatokat betölti, és ideiglenesen eltárolja. Más vezérlőjelek hatására a kimenetére rakja a tárolt adatokat, vagy például egy vezérlőjel hatására, *lépteti (shift-eli*) a benne lévő adatokat.
- Megvalósítások működési mód szerint:
  a) *Hagyományos reg.*: párhuzamos betöltésű/kiolvasású
  b) *Léptető (Shift regiszter)*: soros betöltésű (kiolvasású)

Ismétlés: D flip-flop

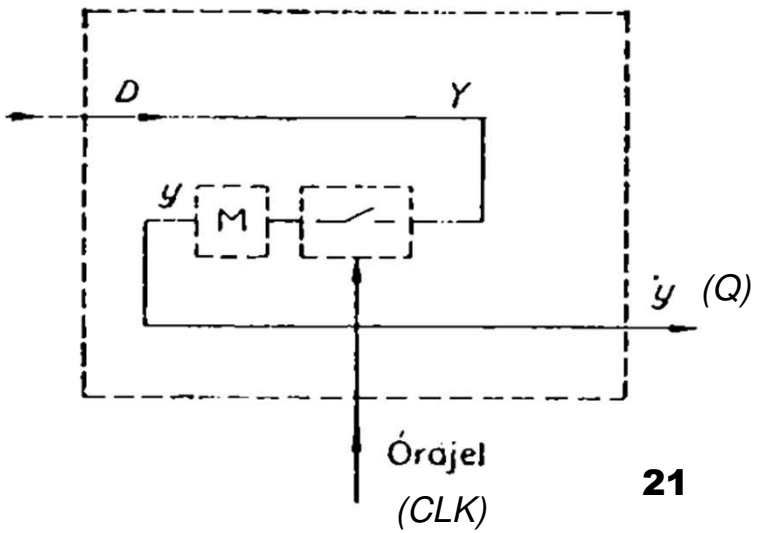
- **D tároló**
  - Csak szinkron módon értelmezhető
- **Működése:**
  - `D='0' ==` D-FF állapota változik (‘0’-t tárol)
  - `D='1' ==` D-FF állapota változik (‘1’-et tárol)
  - Tehát egy órajel ciklus (CLK) ideig tároljuk a bemenetre érkezett értéket, változás a CLK élére történhet



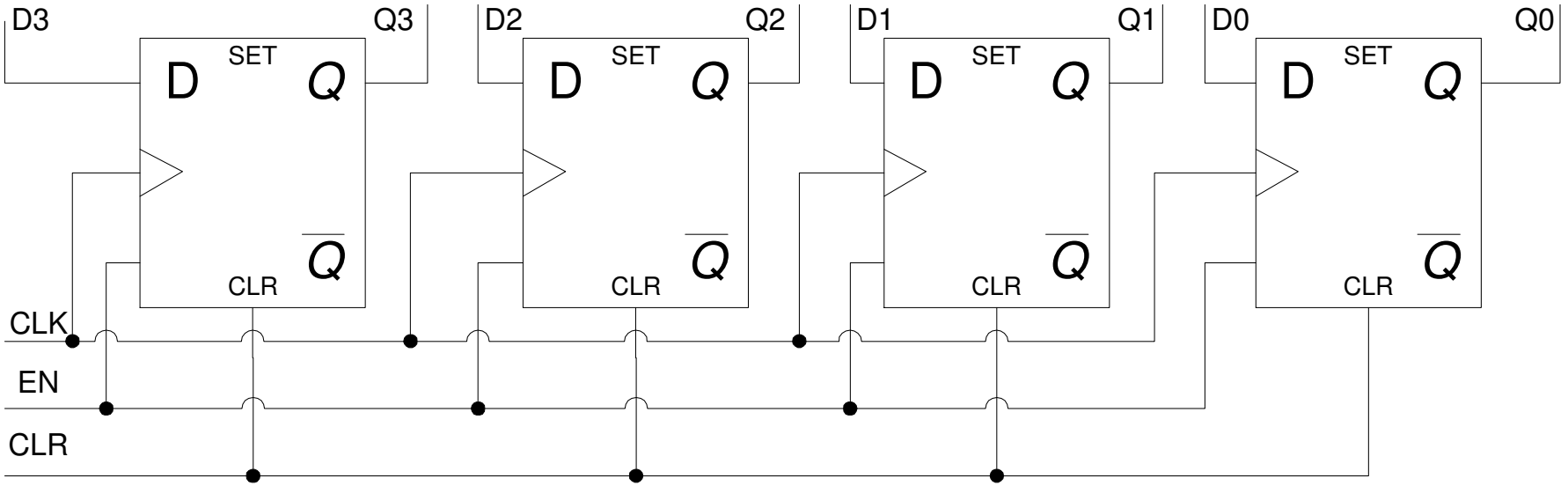


Egyszerűsített DNF alak és elvi logikai rajz:

$$Q = f_y(D, y) = D$$

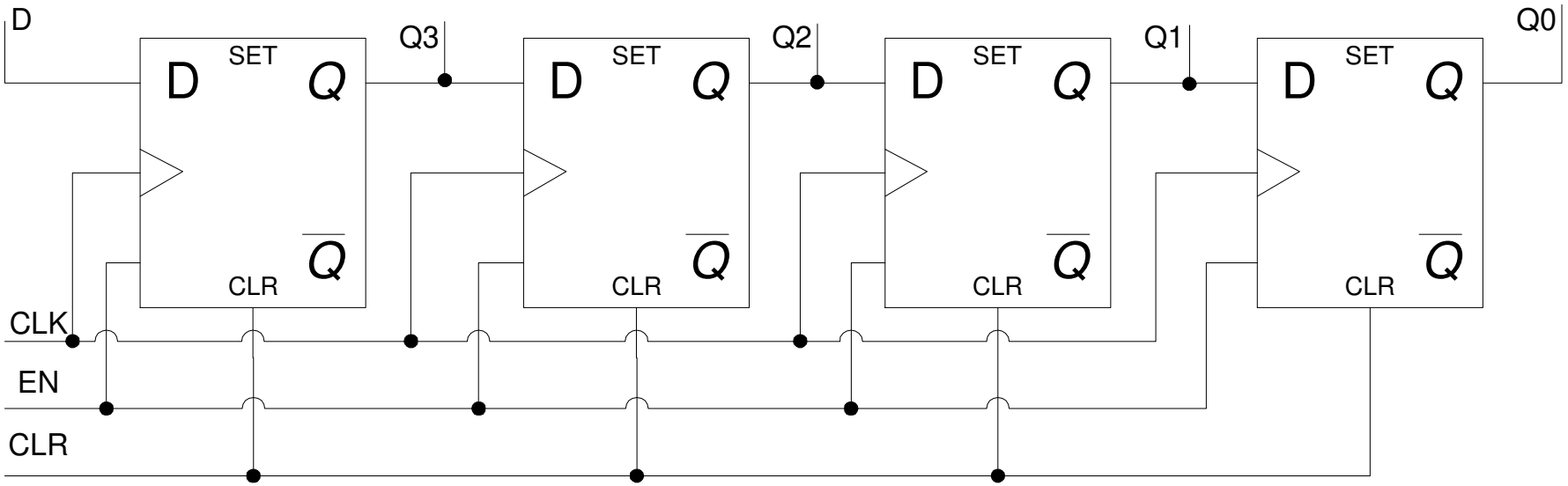


a.) 4-bites Parallel In/ Parallel Out regiszter (D-tárolókból felépítve)



Katalógus adat: SN54/74LS175

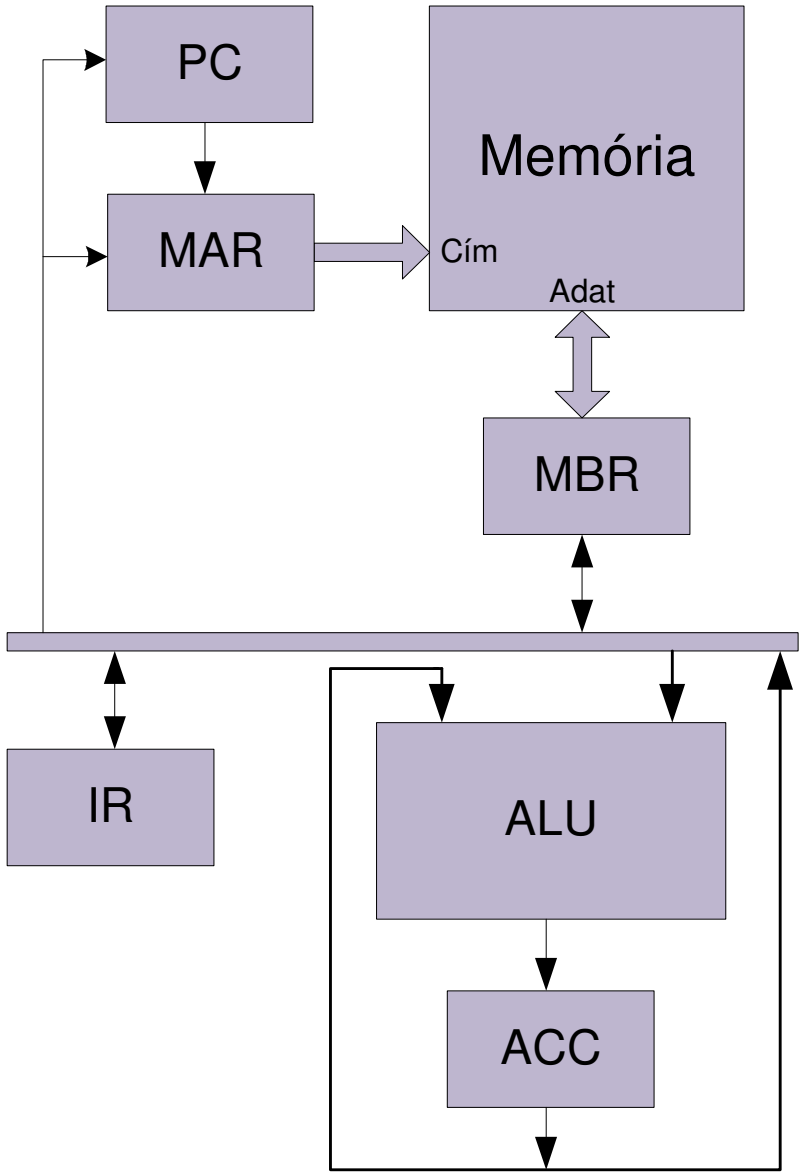
b.) 4-bites Shift/léptető regiszter (Serial in/Paralel Out – D-tárolós)



Katalógus adat : SN54/74LS95

Egyszerű számítógép (egycímű gép) blokkdiagramja

• ***MAR: Memory Address Register*** *(Memória-cím Regiszter)*: információ helyét azonosítja adott memóriacím alapján.
• ***MBR: Memory Buffer Register*** *(Memória Puffer Regiszter)*: tárolja a memóriába bevitt, ill. érkező információt. Egy adott memóriacímen lévő adat kiolvasásakor az ott lévő bejegyzés törlődhet (destruktív memória)
• ***PC: Program Counter*** *(Programszámláló)*: a soron következő (végrehajtandó) utasítás helyét azonosítja. Azon gépeknél, amelyek egy utasítást tárolnak memóriaterületenként, az utasítás végrehajtása után a PC értékét 1-el kell növelni (increment), mint egy számlálót.
• ***IR: Instruction Register*** *(Utasítás Regiszter)*: tárolja az éppen végrehajtás alatt álló utasítást. Engedélyezi a gép vezérlő részeinek, hogy a regiszterek, memóriák, aritmetikai egységek vezérlő vonalait a végrehajtáshoz szükséges működési módba állítsák. Az IR olyan széles, hogy az utasítás műveleti kódja ill. a hozzá tartozó egyéb utasítások ideiglenes másolatai eltárolhatók legyenek.
• ***ACC: Accumulator regiszter*** *(tároló regiszter):* eredmény ideiglenes tárolására használjuk (összes adatkezeléshez tartozó utasítás tárolása).



Neumann architektúra!

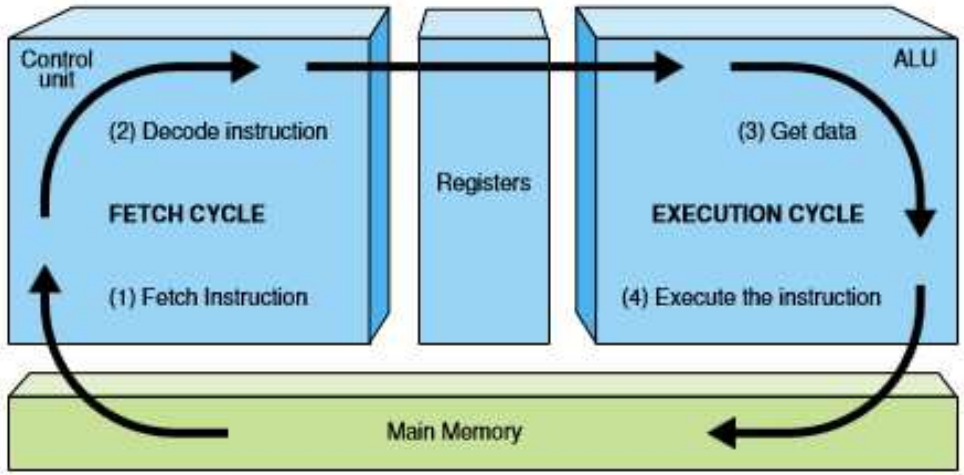
Utasítások kódolása

Utasítás kódok

- A rendszer-tervezéshez szükséges erőforrások a *regiszterek, ALU, memória, adatbuszok* nem elegendőek a végrehajtás egyes fázisainak (tranzakcióknak) ábrázolásánál. Szükség van egy olyan eljárásra, amely leírja ezeket az egyes egységek között végbemenő tranzakciókat.
- Utasítások végrehajtásának leírására szolgáló programnyelv az **assembly**. Az utasítások gyűjteményét - amelyeket a felhasználó/programozó használ az adatkezelésnél - *gépi utasításkészletnek* nevezzük.

FDE mechanizmus

- Egy utasítás végrehajtásának három fő lépését a **Fetch-Decode-Execute** (FDE) mechanizmussal definiálhatjuk:
  - **F - Fetch:** az utasítás betöltődik a memóriából az utasításregiszterbe (regiszter-transzfer művelet)
  - **D - Decode:** utasítás dekódolása (értelmezése), azonosítja az utasítást
  - **E - Execute:** a dekódolt utasítást végrehajtjuk az adatokon, aminek eredménye visszakerül a memóriába
- További lépések lehetnek még (tipikusan):
  - **MEM: Memory operations:** következő utasítás letöltése a memóriából
  - **WB: Memory Write-Back:** eredmény visszaírása



RTL leírás:

- Minden utasítás végrehajtása az **RTL leírás (Regiszter-Transzfer Nyelv)** segítségével írható le. A szükséges adatátviteleket ezzel a nyelvvel specifikáljuk az egyik fő komponenstől a másikig.
- Továbbá megadható az engedélyezett adatátvitelekhez tartozó blokkdiagram (**gráf**) is, az éleken adott irányítással, amelyek az adatátvitel pontos irányát jelölik.
- Az RTL leírások specifikálják a műveletek pontos sorrendjét. Az egyes utasításokhoz megadhatók a *szükséges végrehajtási idők* (pl. `[ns, ps]`–ban), amelyek erősen függenek a felhasznált technológia tulajdonságaitól. Ezek összege fogja megadni a teljes tranzakció időszükségletét.

Néhány alapvető tranzakció specifikációja a következő:

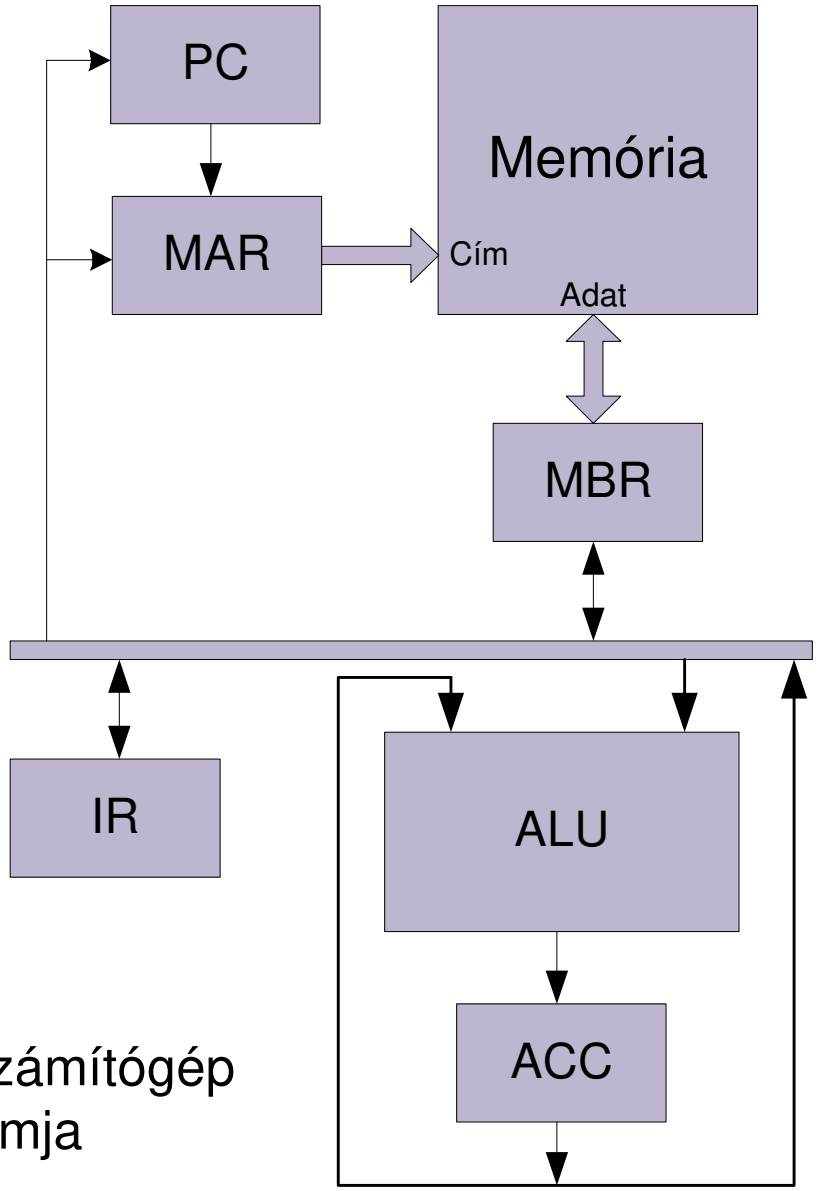
- **PC → MAR** :A Program Számláló tartalma a Memória Cím Regiszterbe töltődik
- **PC+1→ PC** :A PC 1-el inkrementálódik, és PC-be visszatöltődik
- **MBR→IR** :MBR tartalma az IR-be töltődik. Ha az adatbusz megenged többszörös műveletvégzést egyidejűleg, akkor az egyes akciók összekapcsolhatók!
- **M[MAR] → MBR** : MAR címregiszter tartalmával címezzük meg az M memória adott celláját, melynek tartalma az MBR regiszterbe kerül
- **IR <3:0> → ALU** :Az információnak csak egy része, az IR regiszter 3-0 bitje töltődik az ALU-ba
- **REG[2] → MEM[MAR]** :A Regiszter 2. rekesze töltődik a Memória Cím Regiszter adott rekeszébe, a MAR által mutatott címre
- **If (carry==1) then PC-24 → PC** :Feltételes utasítások: Ha átvitel 1, akkor PC 24-el dekrementálódik, és visszatöltődik
- **Else PC+1 → PC** :egyébként 1-el inkrementálódik.

Utasítás formák:

- **Zéró-című (0 című):** (PUSH, POP, ACC) [operátor] (példa: STACK, vagy verem)
- **1-című:** [operátor],[operandus] (Példa: Egyszerű számítógép blokkdiagramja)
- **2-című:** [operátor],[operandus1],[operandus2]
- **3-című:** [operátor],[operandus1],[operandus2], [eredmény]
- …
- **4-című:** [operátor],[operandus1],[operandus2], [eredmény],[következő utasítás]

Egy-című gépek (Egyszerű számítógép)

Példa: DEC PDP-8 számítógépe



Egyszerű számítógép blokkdiagramja

Neumann architektúra!

Egy-című gép

- **Megadása:** [operátor],[operandus]
- A műveletekhez csak 1 operandus szükséges. Ilyen művelet lehet például:
  - 1's vagy 2's komplemens képzés,
  - inkrementálás, törlés, keresés az *ACC*-ben (akkumulátor).
- Az eredmény az ACC-ben tárolódik. (ACC egy olyan speciális regiszter, amelyben az aritmetikai és logikai műveletek eredménye ideiglenesen tárolódik.)
- Két operandus esetén az első operandus ACC-ben tárolt értékét használjuk fel, míg a másik operandust *egyetlen címmel* azonosítjuk!

Példa: Egy-című gép
2's komplemens képzés

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	Elsőként a PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]→MBR	Memóriában lévő utasítás beírása az MBR-be (később visszaírjuk)
PC+I_len→PC	Az utasítás hosszával (I_len) növeli a PC értékét
MBR→IR	Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

$\overline{\textbf{ACC}}$**→ACC**	ACC komplemensét az ACC-be töltjük
ACC+1→ACC	majd ACC-t 1-el inkrementáljuk (eredmény)

Időszükségletek itt még nincsenek feltüntetve!

Példa: 1-című gép (Kivonás SUB_1X)

Mostantól: „X” Operandus címét is a PC-vel azonosítjuk!

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	Elsőként a PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]→MBR	Memóriában lévő utasítás beírása az MBR-be (később visszaírjuk)
PC+I_len→PC	Az utasítás hosszával (I_len) növeli a PC értékét
MBR→IR	Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

PC→MAR	PC-vel a következő címre mutatunk
PC+X_len →PC	X operandus címének hosszával növeljük a PC-t
M[MAR]→MBR	Ezt címet az MBR-be tesszük
MBR→MAR	Ez a cím lesz az X operandus címe
M[MAR]→MBR	Címen lévő értéket az MBR-be töltjük
ACC – MBR→ACC	ACC-ből kivonjuk az X-et, és ACC-be töltjük

Időszükségletek itt még nincsenek feltüntetve!

Példa: 1-című gép (kivonás SUB_1X)

Időszükségletek feltüntetésével! T_{MEM}=30ns, T_{ALU}=10ns, T_{REG}=5ns

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	[5ns] Elsőként a PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]→MBR	[30ns] Memóriában lévő utasítás beírása az MBR-be (később visszaírjuk)
PC+I_len→PC	[5ns] Az utasítás hosszával (I_len) növeli a PC értékét
MBR→IR	[5ns] Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

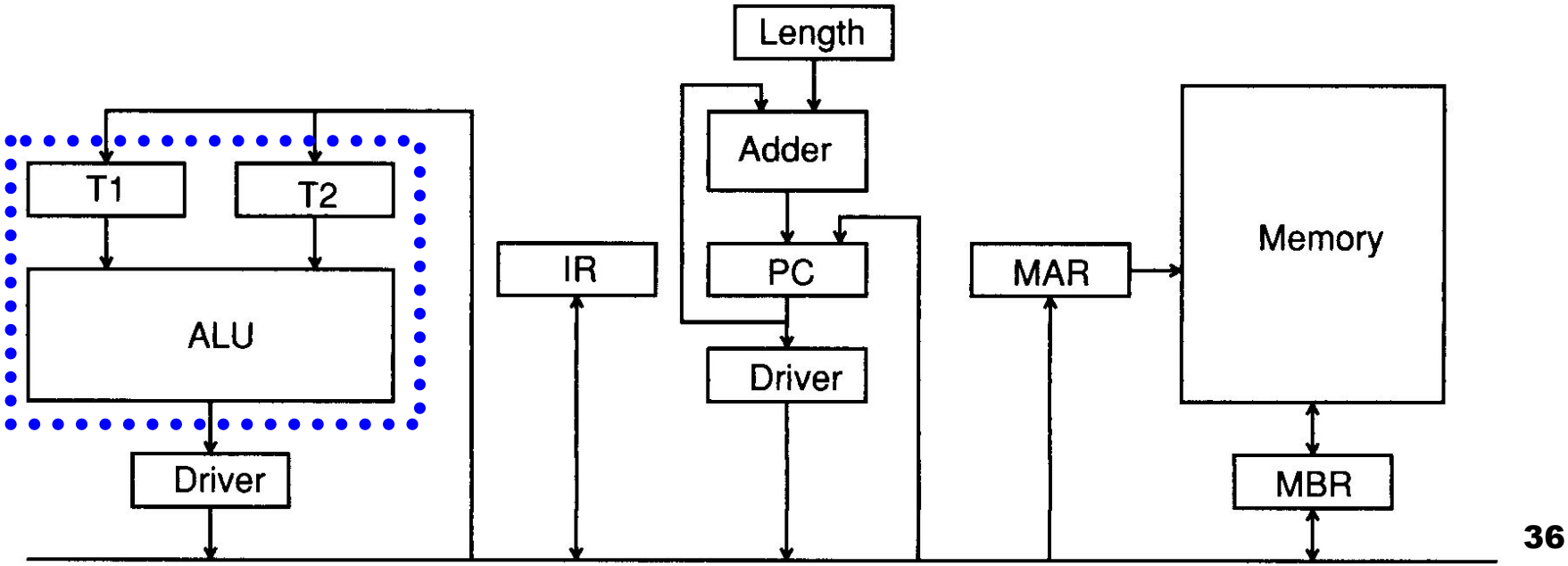
Direkt címzést használunk itt!

PC→MAR	[5ns] PC-vel a következő címre mutatunk
M[MAR]→MBR	[30ns] Ezt címet az MBR-be tesszük
MBR→MAR	[5ns] Ez a cím lesz az X operandus címe
M[MAR]→MBR	[30ns] Címen lévő értéket az MBR-be töltjük
PC+X_len →PC	[5ns] X operandus címének hosszával növeljük a PC-t
ACC – MBR→ACC	[10+5ns] ACC-ből kivonjuk az X-et, és ACC-be töltjük

Σ 135ns

b.) Kettő-, és több-című gépek (regiszter nélküli változat)

- Egy utasítással több operandust / operátort lehet megadni,
- Kevesebb utasítás-sorral, összetett módon írhatók le az RTL nyelven a folyamatok, (az egycímű gépekkel ellentétben)
- A többcímű utasítások meghatározzák, mind a *forrás*, mind a *cél*információt. A célinformáció helyét az utoljára megcímzett operandus adja meg! [operátor],[operandus1],[operandus2]…



Jelölés: kettő-, és többcímű gép

- Jelölés: **ADD$_2$ X, Y** **két-című** utasítás (*műv, op1, op2*). Az X cím által azonosított helyen tárolt értéket hozzáadjuk az Y cím által azonosított helyen lévő értékhez, és az összeadás eredményét az Y címmel azonosított helyen tároljuk el.
- Jelölés: **ADD$_3$ X,Y,Z** **három-című** utasítás (*műv, op1, op2, eredmény*): hasonló az előzőhöz, csak az összeadás eredménye egy új helyen, a Z cím által azonosított helyen tárolódik el.
- Fontos megjegyezni hogy ebben az esetben („*regiszter nélküliség*”) a T1, ill. T2 regiszter nem az utasítás-készlet architektúra része! (Ezért nem keverendő össze a később említésre kerülő *regiszteres* címzéssel!)
  - Ebben az esetben T1, T2-t „csak” az ALU részeként, nem pedig a rendszer gyorsítását szolgáló elkülönített regiszter bankként használjuk.

Példa: Összeadás két-című géppel ADD$_2$(X,Y)

Időszükségletek feltüntetésével!

T_{MEM}=30ns, T_{ALU}=10ns, T_{REG}=5ns

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	[5ns] PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]→MBR	[30ns] Memóriában lévő utasítás beírása az MBR-be
PC+I_len→PC	[5ns] Az utasítás hosszával (I_len) növeli a PC értékét
MBR→IR	[5ns] Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

PC→MAR	[5ns] PC-vel a következő (**X**) címre mutatunk
PC+X_Alen →PC	[5ns] X operandus címének hosszával növeljük a PC-t
M[MAR]→MBR	[30ns] Ezt az *X* címet az MBR-be írjuk
MBR→MAR	[5ns] Ez a cím lesz az X operandus címe
M[MAR]→MBR	[30ns] X címen lévő értéket az MBR-be töltjük
MBR→T1	[5ns] **X értékét** T1-be töltjük
PC→MAR	[5ns] PC-vel a következő (**Y**) címre mutatunk
PC+Y_Alen →PC	[5ns] Y operandus címének hosszával növeljük a PC-t
M[MAR]→MBR	[30ns] Ezt a Y címet az MBR-be írjuk
MBR→MAR	[5ns] Ez a cím lesz az Y operandus címe
M[MAR]→MBR	[30ns] Y Címen lévő értéket az MBR-be töltjük
MBR→T2	[5ns] **Y értékét** T2-be töltjük
T1 + T2→MBR	[10+5ns] ADD2 művelet elvégzése, MBR-be töltjük
MBR→M[MAR]	[30ns] Eredményt a MAR-ban tároljuk el (ahol Y volt)

Σ 250ns

Direkt címzést használunk itt!

Példa: Összeadás három-című géppel ADD$_3$(X,Y,Z)

Időszükségletek feltüntetésével!

T_{MEM}=30ns, T_{ALU}=10ns, T_{REG}=5ns

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	[5ns] PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]→MBR	[30ns] Memóriában lévő utasítás beírása az MBR-be
PC+I_len→PC	[5ns] Az utasítás hosszával (I_len) növeli a PC értékét
MBR→IR	[5ns] Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

PC→MAR	[5ns] PC-vel a következő (**X**) címre mutatunk
PC+X_Alen →PC	[5ns] X operandus címének hosszával növeljük a PC-t
M[MAR]→MBR	[30ns] Ezt az *X* címet az MBR-be írjuk
MBR→MAR	[5ns] Ez a cím lesz az X operandus címe
M[MAR]→MBR	[30ns] X címen lévő értéket az MBR-be töltjük
MBR→T1	[5ns] **X értékét** T1-be töltjük
PC→MAR	[5ns] PC-vel a következő (**Y**) címre mutatunk
PC+Y_Alen →PC	[5ns] Y operandus címének hosszával növeljük a PC-t
M[MAR]→MBR	[30ns] Ezt a Y címet az MBR-be írjuk
MBR→MAR	[5ns] Ez a cím lesz az Y operandus címe
M[MAR]→MBR	[30ns] Y Címen lévő értéket az MBR-be töltjük
MBR→T2	[5ns] **Y értékét** T2-be töltjük
PC→MAR	[5ns] PC-vel a következő (**Z**) címre mutatunk
PC+Z_Alen →PC	[5ns] Z operandus címének hosszával növeljük a PC-t
M[MAR]→MBR	[30ns] a Z eredmény címét az MBR-be írjuk
MBR→MAR	[5ns] majd a MAR-ba töltjük
T1 + T2→MBR	[10+5ns] ADD2 művelet elvégzése, MBR-be töltjük
MBR→M[MAR]	[30ns] Eredményt a memóriában tároljuk el (ahol Z volt)

Σ 295ns

Direkt címzést használunk itt!

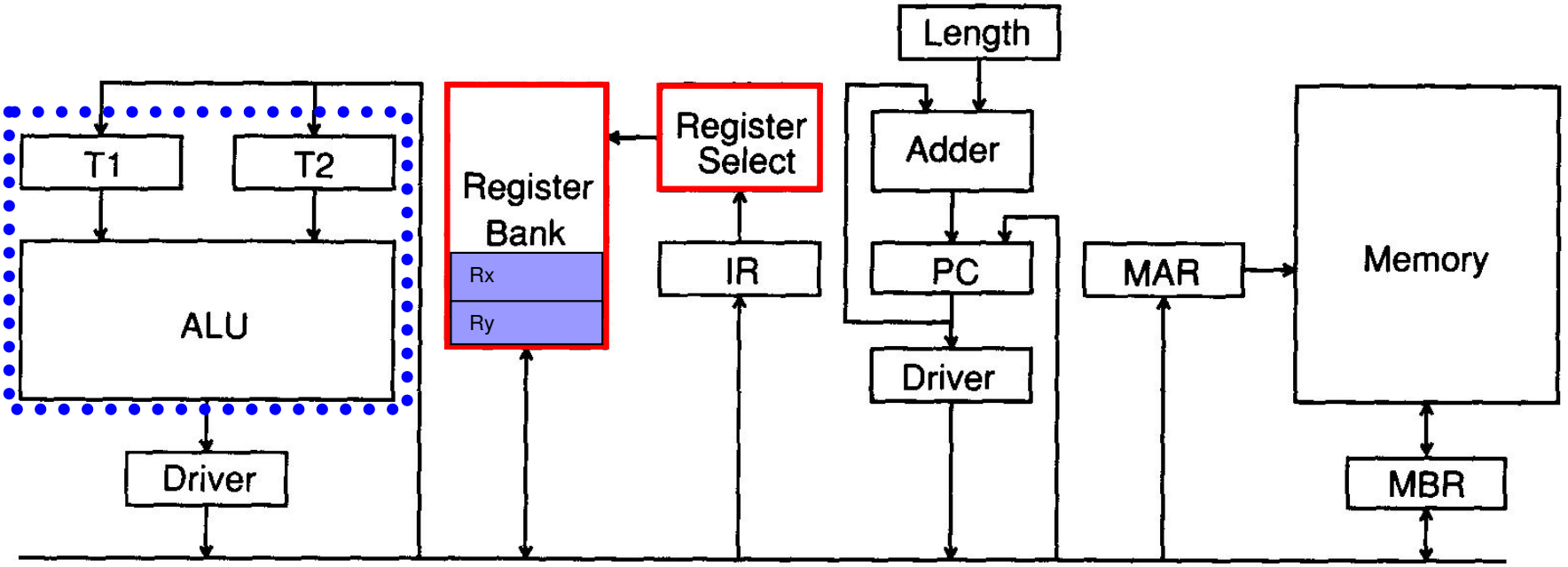
Komplex műveletek: MŰV$_2$ vs. MŰV$_3$

- Az **MŰV$_3$** végrehajtásánál (ahogy az RTL leírásból is látszott), egy sor több időt vesz igénybe az utasítások F-D-E fázisánál, mint az MŰV$_2$ esetén, mivel egyel több címre kell hivatkozni.
- Azonban, az **MŰV$_3$** jelentősége a komplexebb műveletek elvégzésekor mutatkozik meg: tömörebb forma, kevesebb utasítás sor!
- Példa: Legyen X = Y * Z + W * V (oldjuk meg MŰV$_2$-vel, és MŰV$_3$-al)

MŰV$_2$	**MŰV$_3$**
MOVE Y to X	MUL$_3$ Y,Z,T
MUL$_2$ Z,X	MUL$_3$ W,V,Y
MOVE W to Y	ADD$_3$ T,Y,X
MUL$_2$ V,Y	
ADD$_2$ Y,X	

c.) Kettő-, és több-című gépek (Regiszteres változat)

- Egy utasítással több operandust / operátort lehet megadni,
- Kevesebb utasítás-sorral, összetett módon írhatók le az RTL nyelven a folyamatok, (az egycímű gépekkel szemben)
- Az általános célú regiszterek (Reg. Bank) használata csökkenti a végrehajtási időt, mivel a lassú memória-intenzív műveletek helyett gyorsabb regisztermű veleteket használnak. (A regiszterbank 2^N számú regisztert tartalmazhat.)



Példa 1: Összeadás kétcímű géppel $ADD_2(R_X,R_Y)$

Időszükségletek feltüntetésével!

T_{MEM}=30ns, T_{ALU}=10ns, T_{REG}=5ns

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	[5ns] PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]→MBR	[30ns] Memóriában lévő utasítás beírása az MBR-be
PC+I_len→PC	[5ns] Az utasítás hosszával (I_len) növeli a PC értékét
MBR→IR	[5ns] Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

RX→T1	[5ns] **RX értékét** T1-be töltjük
RY→T2	[5ns] **RY értékét** T2-be töltjük
T1 + T2→RY	[10+5ns] ADD2 művelet elvégzése, **RY**-ba töltjük

Σ 70ns

Regiszteres, direkt-címzést használunk itt!

Példa 2: Összeadás kétcímű géppel $ADD_3(R_X,R_Y,R_Z)$

Időszükségletek feltüntetésével!

T_{MEM}=30ns, T_{ALU}=10ns, T_{REG}=5ns

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	[5ns] PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]→MBR	[30ns] Memóriában lévő utasítás beírása az MBR-be
PC+I_len→PC	[5ns] Az utasítás hosszával (I_len) növeli a PC értékét
MBR→IR	[5ns] Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

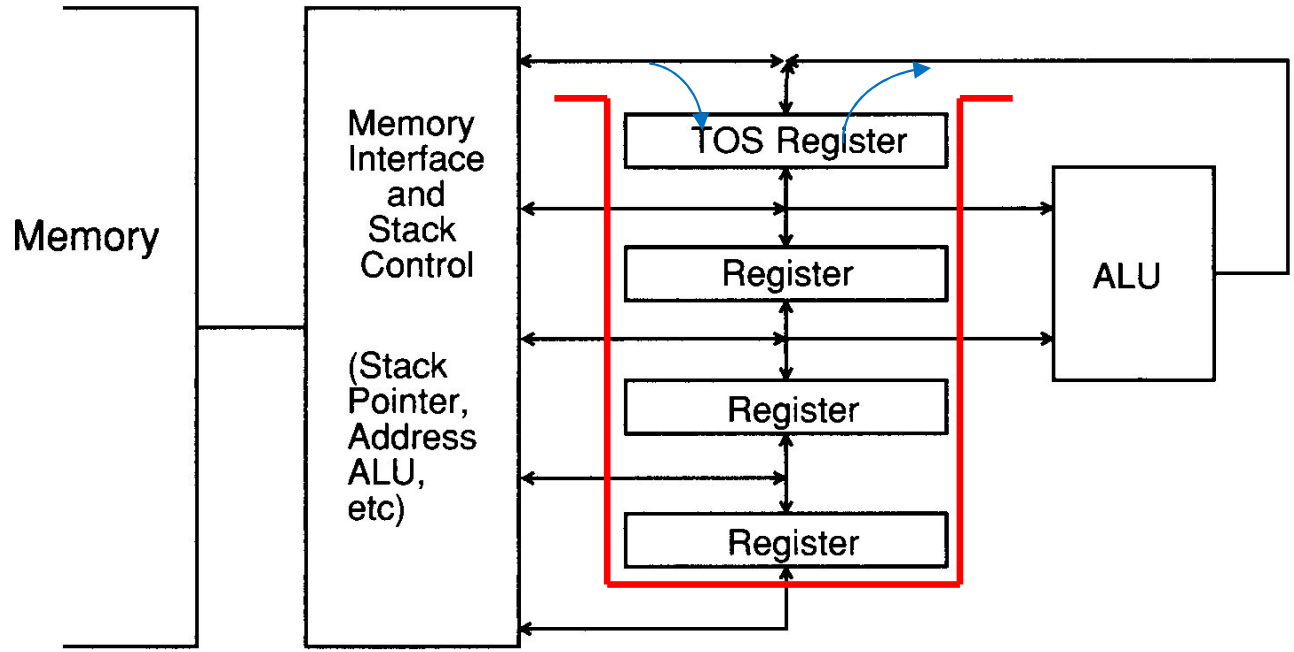
RX→T1	[5ns] **RX értékét** T1-be töltjük
RY→T2	[5ns] **RY értékét** T2-be töltjük
T1 + T2→RZ	[10+5ns] ADD2 művelet elvégzése, **RZ**-be töltjük

Σ 70ns

Regiszteres, direkt-címzést használunk itt!

d.) Zéró-, vagy 0-című gépek (Stack)

- Egy vermet (Stack) használunk: LIFO-típusú tároló, amelyből az utoljára betett adatot vesszük ki elsőként. A Stack a memóriában található egy elkülönített részen. MŰV$_0$ (). Alkalmazása:
  - Függvény hívás, visszatérés (CALL-RETURN eljárások)
  - Különböző aritmetikai kifejezések hajthatók végre elég hatékonyan stack használatával: a szükséges operandusokat a stack egy-egy rekeszében tároljuk. A megfelelő operandusokat (mindig a felső kettő regiszterből) vesszük ki, elvégezzük rajtuk a műveletet, majd az eredményt a verem tetejére tesszük vissza. Azért nevezzük **zéró címűnek**, mivel az operandusok azonosítására szolgáló utasításhoz nem használunk címeket.
  - Egy HW-orientált stack rendszert felépítése a következő:



Példa: Zéró-, vagy 0-című gép

Legyen **F=A+[B*C + D*(E / F)]** aritmetikai kifejezés, F-et akarjuk kiszámolni verem segítségével és eltárolni az eredményt. A következő műveletek szükségesek: **PUSH, POP**, **ADD$_2$, DIV$_2$, MULT$_2$**

Fontos: minden elvégzett művelet egy szinttel csökkenti a verem mélységét!

1. **módszer** kiértékelésénél az aritmetikai kifejezés elejétől haladunk, és amint lehetséges a verem tetején lévő két értéken végrehajtjuk a soron következő műveletet, az eredményt, pedig a verem tetejére pakoljuk. A veremben max. 5 értéket tárolunk el, (`mélysége 5 lesz`) ezért lassabb, mint a második módszer.
2. **módszernél** az aritmetikai kifejezést hátulról előrefelé haladva értékeljük ki. Itt is elvégezzük a soron következő műveletet, és az eredményt a verem tetejére rakjuk. De ez gyorsabb módszer, mivel a veremben max. csak 3 értéket tárolunk el (`mélysége 3`).

1. módszer: (arit. kif. elejétől haladva)	2. módszer: (arit. kif. végétől visszafelé haladva)
PUSH A	PUSH E
PUSH B	PUSH F
PUSH C	DIV [E/F]
MULT [B*C]	PUSH D
PUSH D	MULT [D*(E/F)]
PUSH E	PUSH C
PUSH F	PUSH B
DIV [E/F]	MULT [B*C]
MULT [D*(E/F)]	ADD [B*C+D*(E/F)]
ADD [B*C+D*(E/F)]	PUSH A
ADD [A+(B*C+D*(E/F))]	ADD [A+(B*C+D*(E/F))]
POP F	POP F

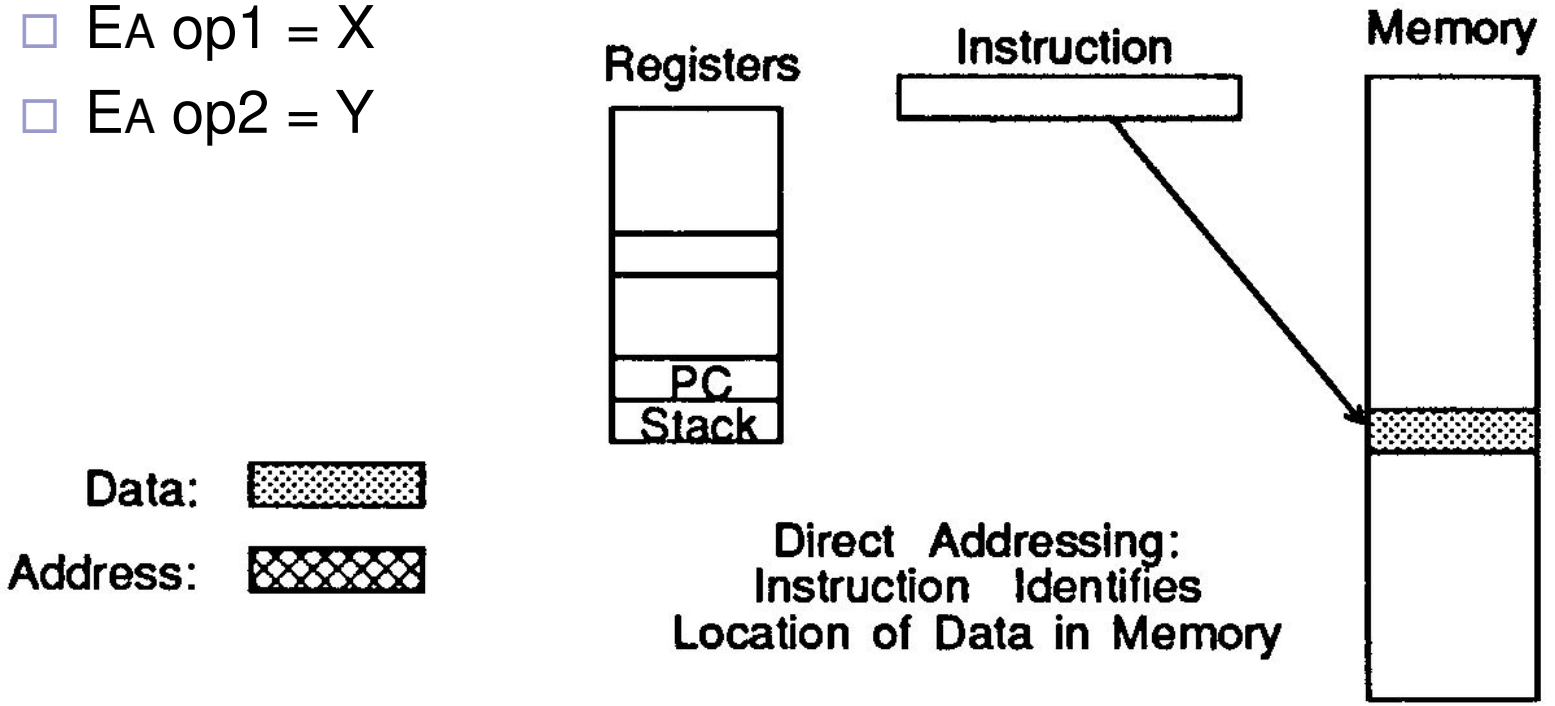
Operandus „címzési módok”

Operandus „címzési módok”

- Utasítás végrehajtásakor a kívánt operandust el szeretnénk érni, címével hivatkozhatunk a pontos helyére, azonosítjuk őt.
- Többféle címzési mód is létezik:
  - közvetlen (directed),
  - közvetett (indirected),
  - indexelt (indexed),
  - regiszteres megvalósítású (register relative).
- Ezek kombinációja igen sokféle, általánosan legalább 10-féle azonosítási mód ismert.
- Jelölés: E_A= Effektív (valódi) címe egy operandusnak

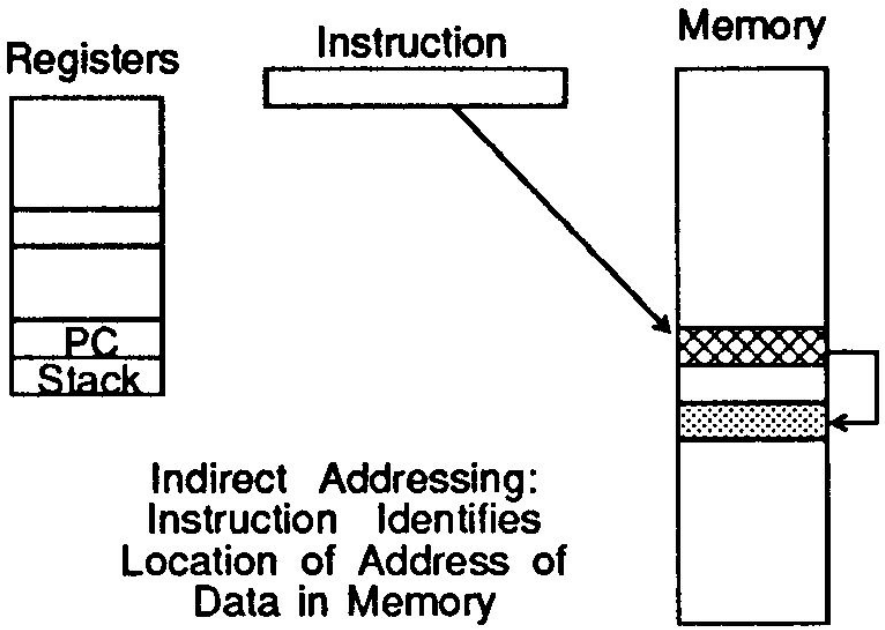
1. Direkt címzés (X)

- Az utasítás egyértelműen, közvetlenül azonosítja az operandus helyét a memóriában. (effektív cím E_A= valódi címén tárolt érték) Jel: E_A=A.
- **Jel: ADD$_2$ X,Y** (X-ben tárolt op1 értéket hozzáadjuk az Y-ban tárolt op2 értékhez, az eredmény az Y-ban lesz.)
  - E_A op1 = X
  - E_A op2 = Y



2. Indirekt címzés (*X)

- Az utasítás közvetett módon, (nem közvetlenül az operandus értékére), hanem az operandus **helyére** mutat egy **cím** segítségével a memóriában. Ez a cím a helyet azonosítja. Ez sokkal hatékonyabb megvalósítás.
- Jel: **ADD$_2$ *X,*Y (*: indirekció)**
  - E_A op1 = MEM[X]
  - E_A op2 = MEM[Y]
- Ezt különböző gyártók többféleképpen jelölik. Általában az indirekt címzési módot (*)-al jelölik:
- Példa: **ADD$_2$ *X,*Y** (az első op1 értékének címe az X-ben található, a második op2 értékének címe az Y-ban lesz, és az eredmény is az Y-ban tárolódik el.) Az 1.), 2.), 3.), 4.), közül ez a *leglassabb* megvalósítás, de az indirekt címzés a *memóriatömb* elemeinek elérhetőségét biztosítja!



Direkt-indirekt kombináció is lehetséges:

PI: ADD$_2$ X, *Y

Példa: Összeadás két-című géppel ADD$_2$(*X,*Y)

Időszükségletek feltüntetésével!

T_{MEM}=30ns, T_{ALU}=10ns, T_{REG}=5ns

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	[5ns] PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]→MBR	[30ns] Memóriában lévő utasítás beírása az MBR-be
PC+I_len→PC	[5ns] Az utasítás hosszával (I_len) növeli a PC értékét
MBR→IR	[5ns] Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

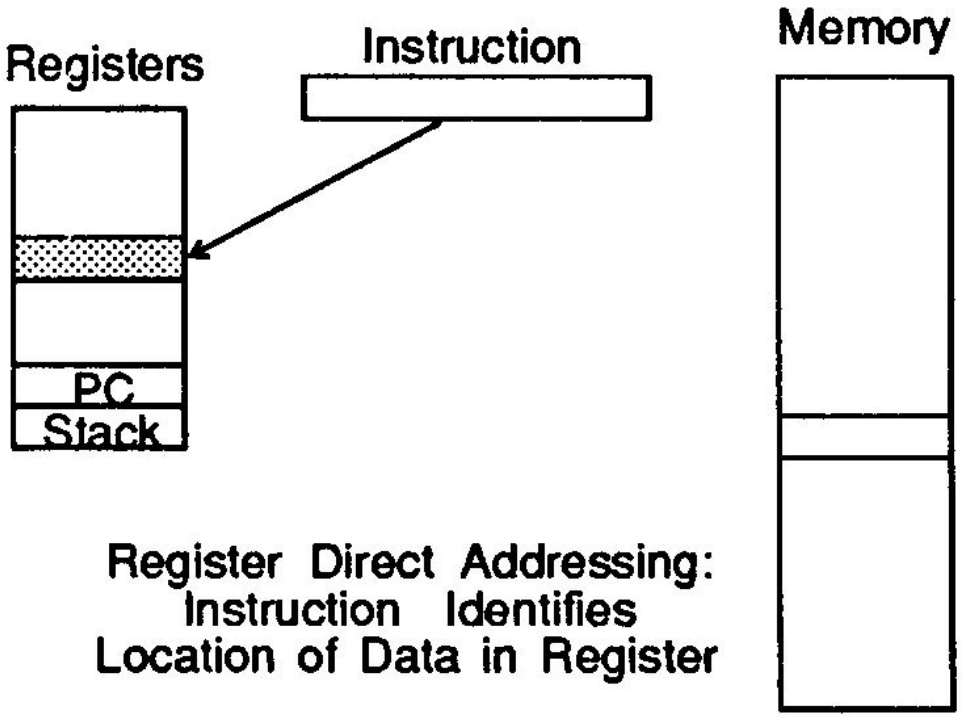
PC→MAR	[5ns] PC-vel a következő (**X**) *címének címére* mutatunk
PC+X_Alen →PC	[5ns] X operandus címének hosszával növeljük a PC-t
M[MAR]→MBR	[30ns] *X* címének címét az MBR-be írjuk
MBR→MAR	[5ns] Ezt a címet a MAR-ba töltjük
M[MAR]→MBR	[30ns] X *címét* megkapjuk az MBR-ben
MBR→MAR	[5ns] X címét a MAR-ba töltjük
M[MAR]→MBR	[30ns] X címén lévő *értékét* megkapjuk MBR-ben
MBR→T1	[5ns] **X értékét** T1-be töltjük
PC→MAR	[5ns] PC-vel a következő (**Y**) *címének címére* mutatunk
PC+Y_Alen →PC	[5ns] Y operandus címének hosszával növeljük a PC-t
M[MAR]→MBR	[30ns] Y címének címét az MBR-be írjuk
MBR→MAR	[5ns] Ezt a címet a MAR-ba töltjük
M[MAR]→MBR	[30ns] Y *címét* megkapjuk az MBR-ben
MBR→MAR	[5ns] Y címét a MAR-ba töltjük
M[MAR]→MBR	[30ns] Y címén lévő *értékét* megkapjuk MBR-ben
MBR→T2	[5ns] **Y értékét** T2-be töltjük
T1 + T2→MBR	[10+5ns] ADD2 művelet elvégzése, MBR-be töltjük
MBR→M[MAR]	[30ns] Eredményt a memóriában tároljuk el (ahol Y volt)

Indirekt címzést használunk itt!

Σ 320ns

3. Regiszteres direkt címzés (R_X)

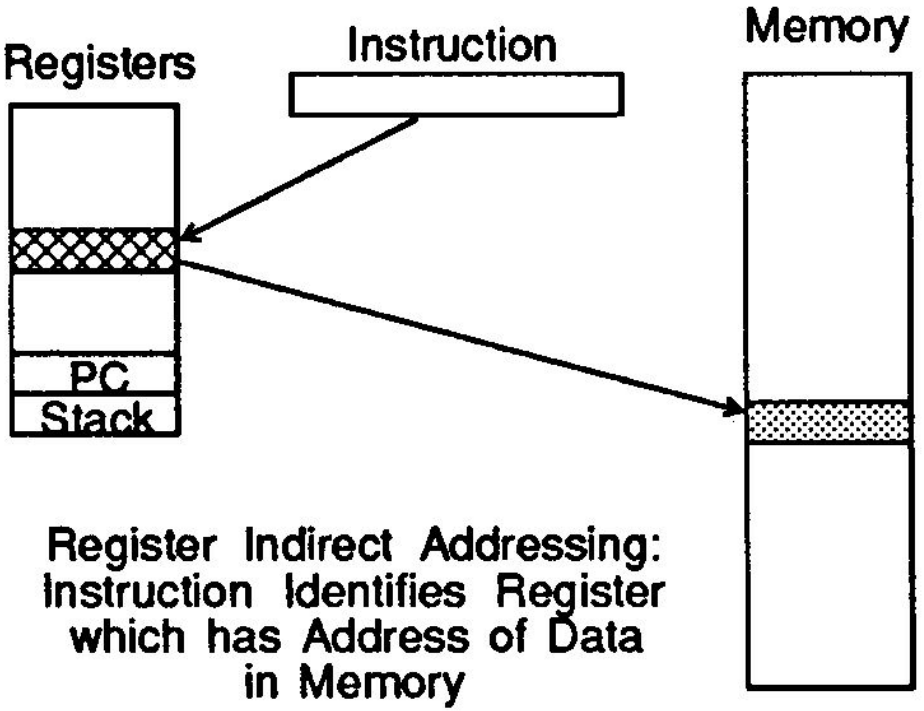
- Hasonló, mint a direkt címzés, de sokkal gyorsabb, mivel a memória intenzív-műveletek helyett a köztes eredményeket a gyors regiszterekben tárolja, és csak a számítási eredményt tölti át a memóriába. Az 1), 2), 3), 4) közül ez a *leggyorsabb* módszer.



Register Direct Addressing:
Instruction Identifies
Location of Data in Register

4. Regiszteres indirekt címzés (*R_X)

- Hasonló, mint az indirekt címzés, de sokkal gyorsabb, mivel a memória-intenzív műveletek helyett a köztes eredményeket a gyors regiszterekben tárolja, és csak a végén tölti át a memóriába. A 3.) regiszteres módszer után ez a második leggyorsabb.



Register Indirect Addressing:
Instruction Identifies Register
which has Address of Data
in Memory

Példa: Összeadás kétcímű géppel ADD$_2$(*R_X,*R_Y)

Időszükségletek feltüntetésével!

T_{MEM}=30ns, T_{ALU}=10ns, T_{REG}=5ns

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	[5ns] PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]→MBR	[30ns] Memóriában lévő utasítás beírása az MBR-be
PC+I_len→PC	[5ns] Az utasítás hosszával (I_len) növeli a PC értékét
MBR→IR	[5ns] Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

Regiszteres indirekt-címzést használunk itt!

RX→MAR	[5ns] **RX címét** a MAR-ba töltjük
M[MAR]→MBR	[30ns] Kinyerjük az **RX** címén lévő **értéket**, amit MBR-be töltünk
MBR→T1	[5ns] **RX értékét** T1-be töltjük
RY→MAR	[5ns] **RY címét** a MAR-ba töltjük
M[MAR]→MBR	[30ns] Kinyerjük az **RY** címén lévő **értéket**, amit MBR-be töltünk
MBR→T2	[5ns] **RY értékét** T2-be töltjük
T1 + T2→MBR	[10+5ns] ADD2 művelet elvégzése, MBR-be töltjük
MBR→ M[MAR]	[30ns] eredményt a memóriában RY operandus helyén tároljuk el

Σ 170ns

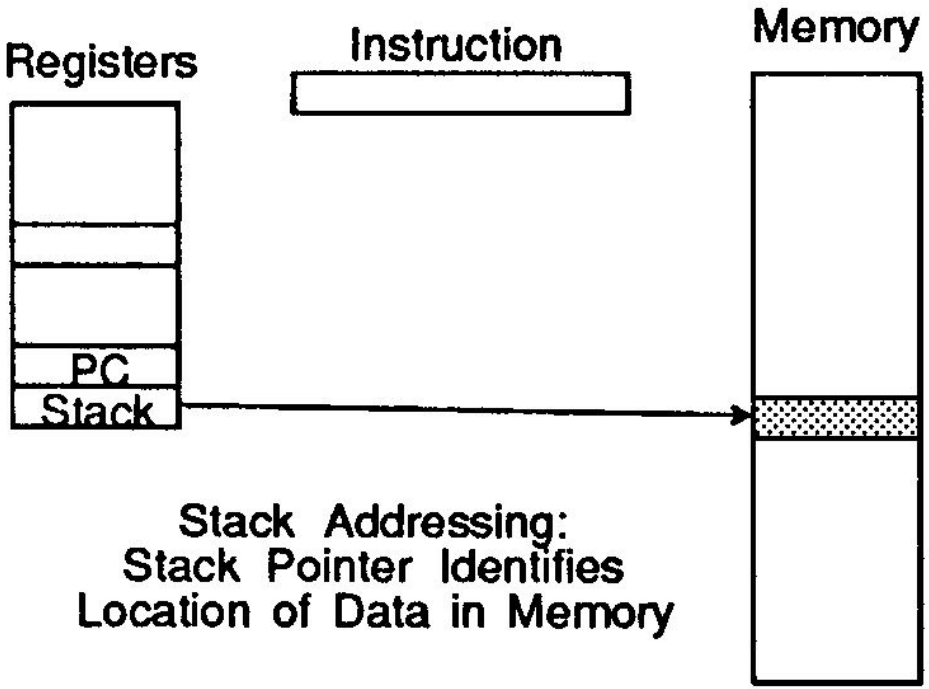
Összehasonlító táblázat – I.

- Az 1.) – 4.) címzési módok időszükségleteinek összehasonlító táblázata:

Címzési módszer	Memória hivatkozások száma	Fetch (ns)	Execute (ns)	Total Time (ns)
Direkt	6	45	205	250
Indirekt	8	45	275	320
Regiszteres direkt	1	45	25	70
Regiszteres indirekt	4	45	125	170

5. Verem (Stack) címzés

- **Verem (STACK) címzés, vagy regiszteres indirekt autoincrement címzési mód:** *indirekt* módszerrel az operandus memóriában elfoglalt helyét a címével azonosítjuk, és akár az összes memóriatömbben lévő elem megcímezhető. *Autoincrement* is, mivel a címeket automatikusan növeli. Ezt (+) jellel jelöljük: ***Rx+**
- Ezt a mechanizmust használjuk a verem esetében. A Stack-et a memóriában foglaljuk le. A stackben lévő információra a stack pointerrel (SP-mutatóval) hivatkozunk. A stack egy *LIFO tároló*: amit utoljára tettünk be, tehát ami a verem tetején van, azt vehetjük ki legelőször.
- A stack pointer címe jelzi a TOS verem tetejét, ahol a hivatkozott információ található, ill. címmel azonosítható a következő elérhető hely. A stack (az ábra szerint) lefelé növekszik a memóriában.

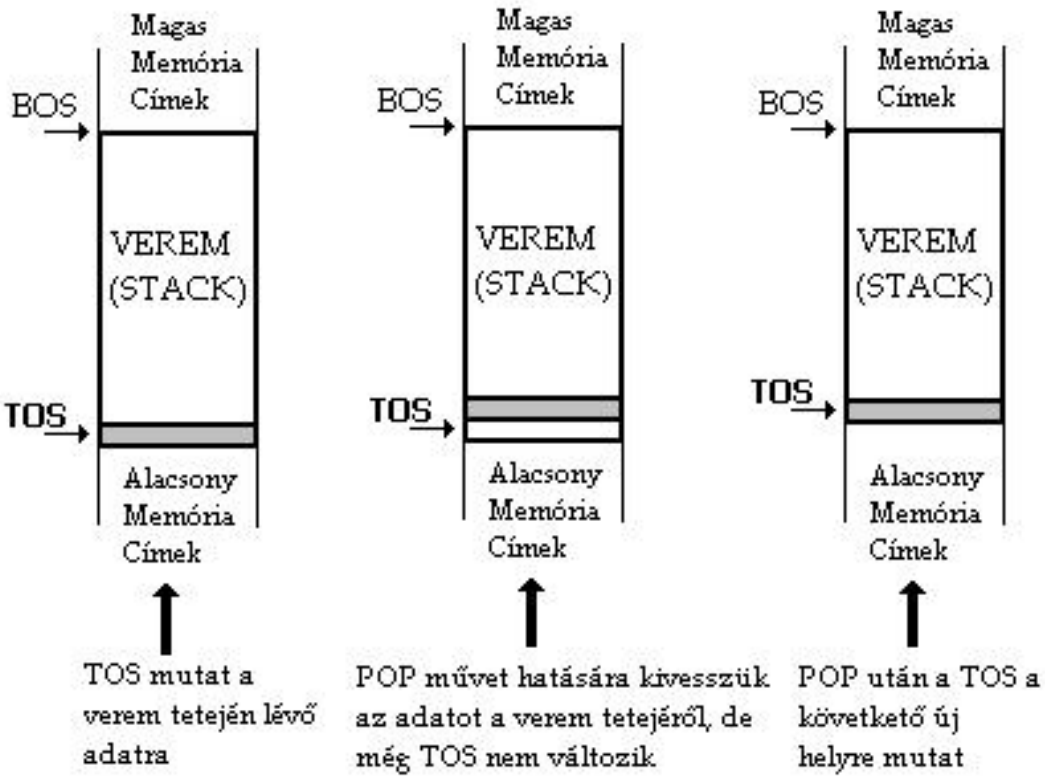


Verem – PUSH, POP műveletek

- **POP művelet** kivesszük a stack tetején lévő adatot (TOS), és egy Rx regiszterbe rakjuk. Ezután a stack pointer automatikusan `inkrementálja` a címet, amivel a következő elemet azonosítja a verem tetején.
  Jel: **MOVE *R (stack pointer)+, Rx**

- **PUSH művelet:** a stack pointer automatikusan `dekrementálja` a címet, amivel a verem tetején lévő elemet azonosítja. Majd ezután berakjuk az Rx regiszterben lévő elemet a stack tetejére, a pointer által mutatott címre.
  Jel: **MOVE Rx, *R (stack pointer)–**

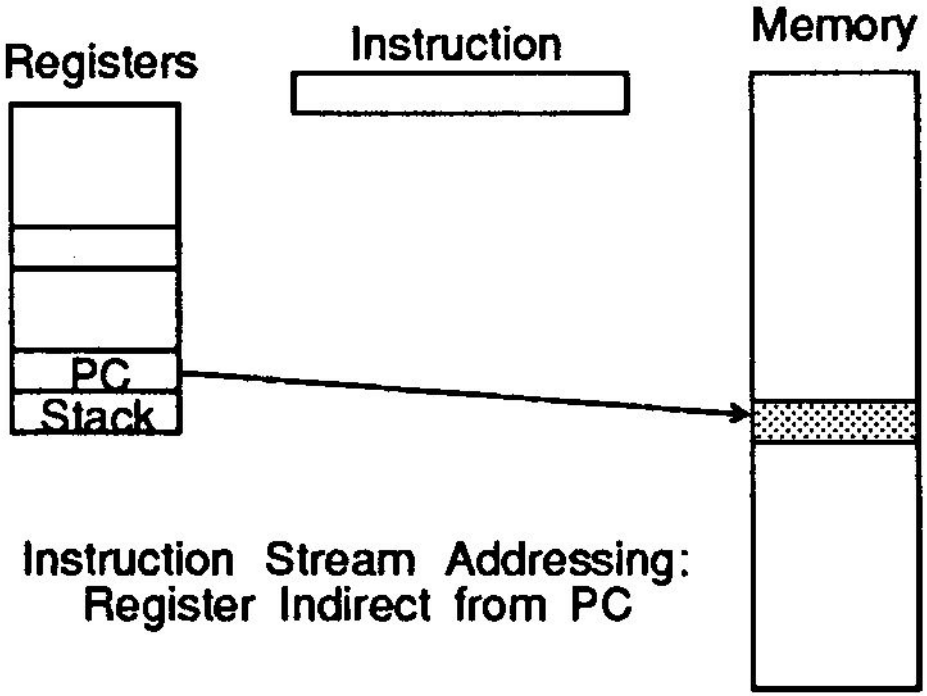
Példa: POP műveletre



Figyelem! Magas memória cím a verem alját (BOS) jelöli.

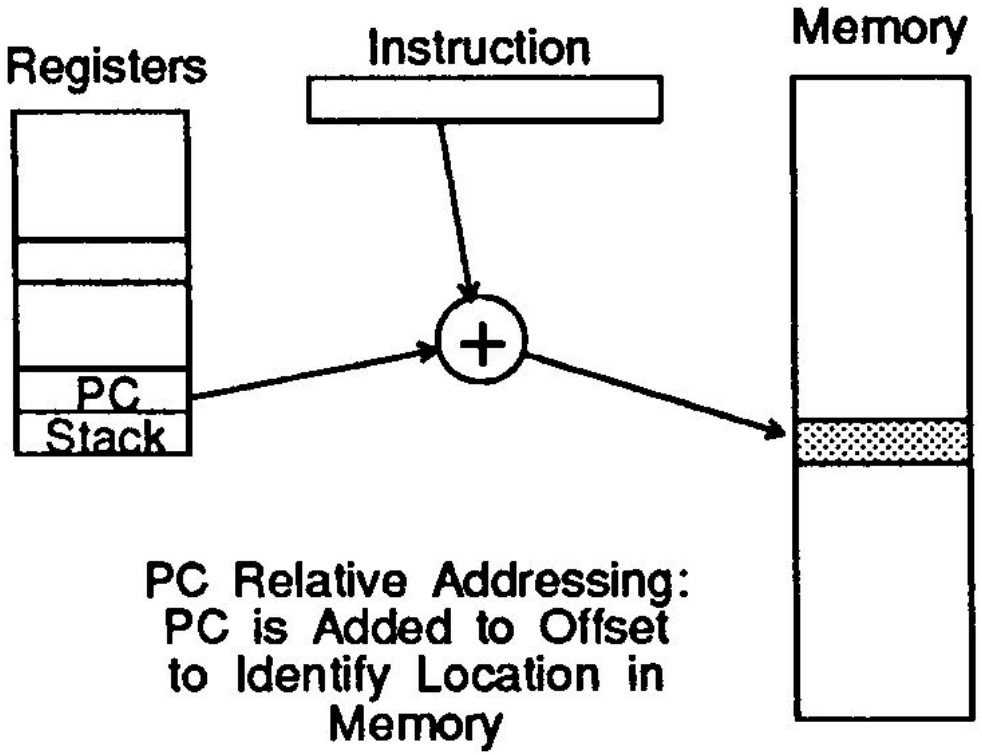
6. Instruction Stream címzés

- A PC közvetlenül azonosítja a memóriában lévő adatot és címet. Az utasítás végrehajtásakor visszakapjuk az utasításfolyamból magát az utasítást, amelyet a *PC* azonosít. Hívják még *azonnali módszernek* (`imm` X) is, mivel az adatok és címek azonnal a rendelkezésünkre állnak. Konstansok, előredefiniált címek szerepelhetnek az utasítás-folyamban.



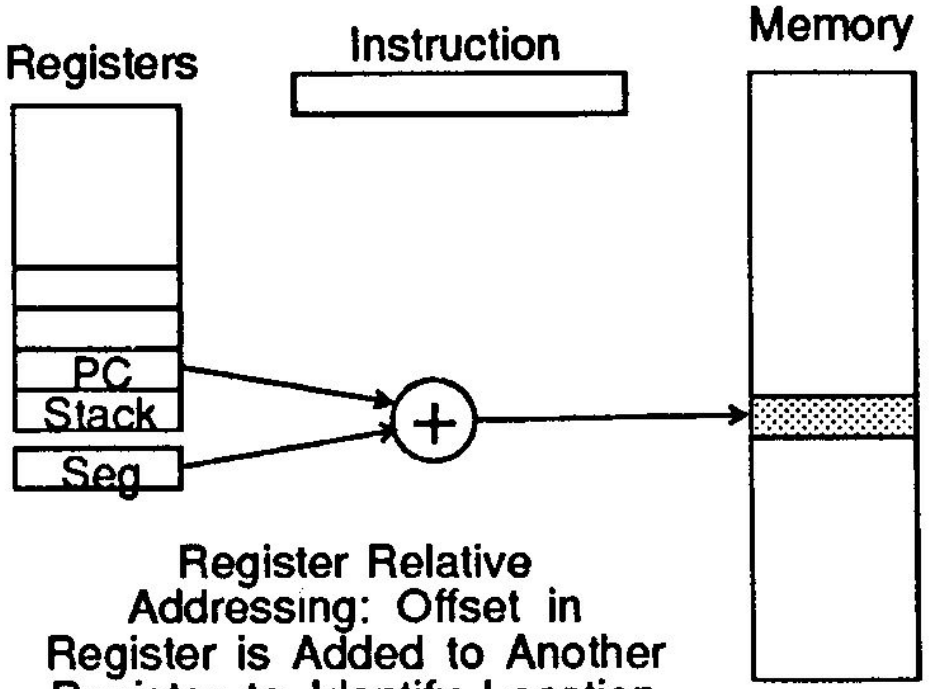
7. PC-relatív címzés

- Memóriabeli adat címét a regiszteren belüli PC értéke, és az utasítás eltolási (offset) értéke együttesen azonosítja.
  - Effektív cím = Regiszteren belüli PC értéke + Eltolás (offset) értéke



8. Regiszter relatív címzés

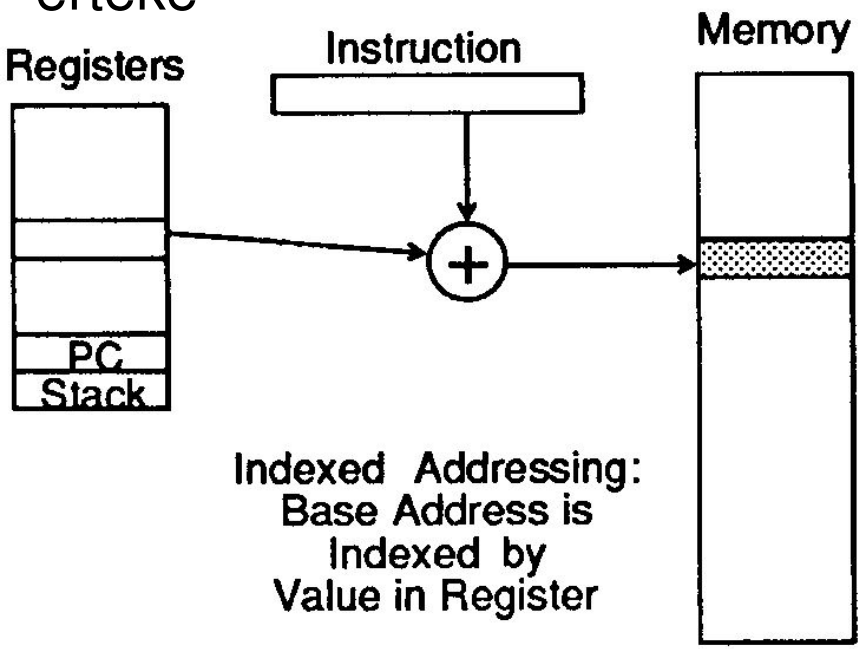
- A PC-regiszter eltolási értékéhez hozzáadódik *egy, vagy több* másik, külső Szegmens-regiszter értéke. Tehát ez abban különbözik a PC-relatív címzéstől, hogy itt a címzés két különböző regiszter segítségével történik.
  - Effektív cím = Regiszternek a PC eltolási értéke (offset) + Szegmens regiszter értéke



Register Relative Addressing: Offset in Register is Added to Another Register to Identify Location in Memory

9. Indexelt címzési mód

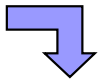
- A memóriában lévő operandus helyét legalább két érték összegéből kapjuk meg. Tehát a tényleges címet az *indexelt bázisértékből*, és az általános célú regiszter értékéből kapjuk meg. Ezt módszert használják adatstruktúrák indexelt tárolásánál. (Pl: tömböknél)
  - Effektív cím= utasításfolyam bázis értéke + általános célú regiszter értéke



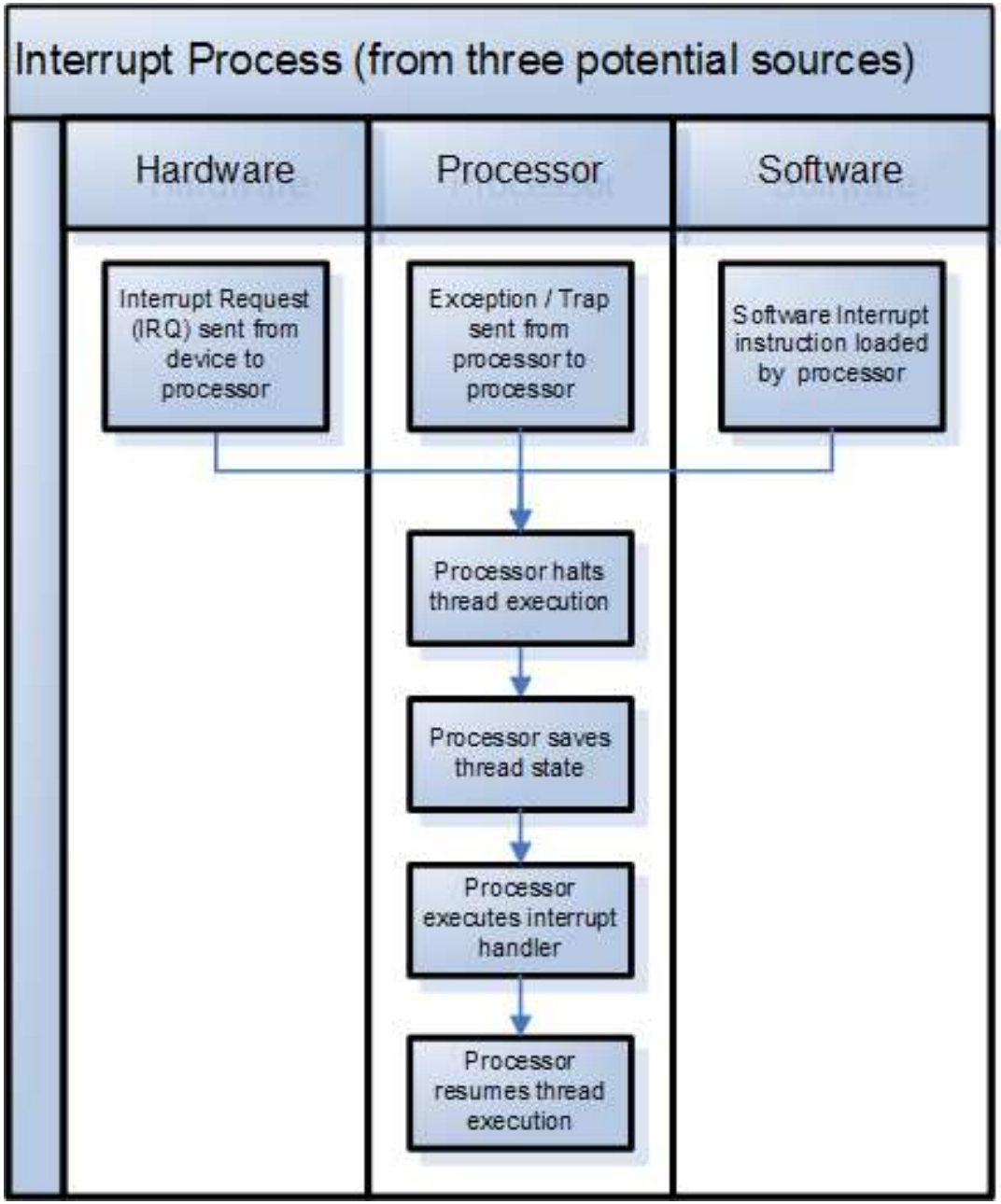

```
int main(void){
      int i;
      ptr = &my_array[0];     /* point our ptr pointer
          to the first  element of the my_array */
      printf("\n\n");
      for (i = 0; i < 6; i++)    {
      printf("my_array[%d] = %d   ",i,my_array[i]);
          /*ver.A */
      printf("ptr + %d = %d\n",i, *(ptr + i) );
          /*ver.B */
      }
      return 0;
}
```

Programvezérlési (program-control) módszerek

- RTL szinten: CALL-RETURN (szubrutin)

- I/O vezérlés:
  - memory-mapped I/O
  - DMA: Direct Memory Access (lásd. chapter6.pdf)
- Megszakítás (interrupt) [IRQ]
  - HW: nem-maszkolható interrupt (NMI) = nem / maszkolható (ignorable)
  - CPU
  - SW: 
- Trap (csapda): programvezérelt megszakítás
  - =Exception: kivétel (pl. 0-val osztás)

Megszakítások



Megszakítás (interrupt)

- HW: külső eszköz
- CPU: multiprocesszoros, társprocesszoros rendszerben másik CPU-tól érkező
- SW: speciális utasítás, vagy program végrehajtása végén küldi (exception is lehet)

Minden megszakításhoz saját „lekezelő” handler tartozik

Bővebben: Operációs rendszerek órán!

RISC és CISC processzorok utasításkészletei

Utasítás készletek

- Fontos paraméter: utasítások száma (intruction set),
- Kezdetben egyszerű felépítésű gépek: korai generációk,
- Egyszerű utasítások, és gépi nyelv,
- Azonban a komplex problémákat kívántak megoldani (magasabb szintű leírással) → **„Szemantikus rés”**
  - Megoldás: compiler
- CISC, RISC mikro-architektúrák

RISC processzorok jellemzői (1):

- Például: Motorola 88000 RISC rendszere, vagy Berkeley RISC-I rendszere, Alpha, SPARC, MicroChip PIC, ARM, DSP sorozatok, IBM PowerPC stb.
- **RISC:** Reduced Instruction Set Computer (Csökkentett utasításkészletű számítógép):
- Csak a kívánt alkalmazásra jellemző utasítástípusokat tartalmaz, az utasításkészlet összetettségének csökkentése végett kihagytak olyan utasításokat, amelyeket a program amúgy sem használ, ezáltal nő a sebesség.
- *Minimális utasításkészletet és címzési módot* (csak amit gyakran használ), gyors HW elemeket, optimalizált SW használ.
- Azonban, hogy a programozási nyelvek komplex függvényei leírhatók legyenek (ahogyan az a CISC-nél működik) szubrutinokra, és hosszabb utasítássorozatokra (sok egyszerű utasítás) van szükség.
- Hogyan tudjuk a rendszer erőforrásait hatékonyan kihasználni? Gyorsabb működés érhető el (MIPS), egyszerűbb architektúra megvalósítására kell törekedni.
- *Azonos hosszúságú utasításformátum* (korlátozott utasításformátum miatt a tárolt programú gépeknél az *F-D-E* folyamatban a dekódolás minimális idejű lesz (nullának feltételezzük), amely során azonosítani kell a végrehajtandó utasítást)

RISC processzorok jellemzői (2):

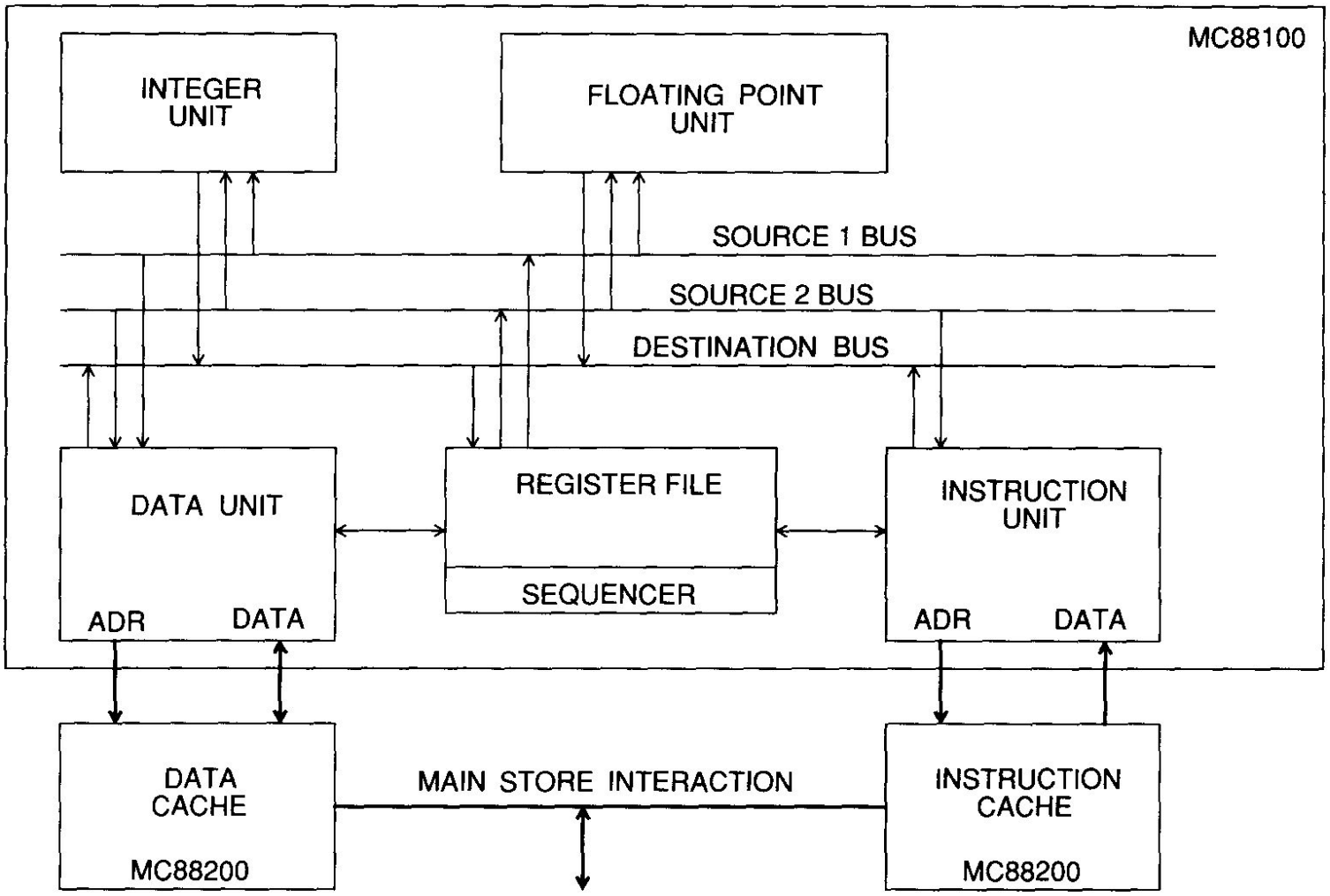
- *Huzalozott (hard-wired) utasításdekódolás* (a hardveres dekódolás megvalósításához kombinációs logikát használ, azonban a mai memóriaalapú mikro-kódú gépeknél ez lassabb).
- *Egyszeres ciklusvégrehajtás*: (minden egyes ciklusban egy utasítást hajt végre, ha ezt sikerülne elérni optimális lenne az erőforrás kihasználás - VLSI technológiafüggő. Egy lebegőpontos művelet rendkívül kis idő alatt végrehajtható. Hátránya, hogy vannak bizonyos műveletek, amelyeket egy ciklus alatt nem kapunk meg: pl. a memóriában lévő érték inkrementálásakor az értéket előbb ki kell venni, frissíteni, majd visszaírni a memóriába).
- *LOAD/STORE memóriaszervezés*: 2 művelet – tölt és tárol (regiszter <-> memória). Regiszterre azért van szükség, mivel a betöltött adatot sokkal gyorsabban tudjuk kiolvasni, mint a memóriából. Az aritmetikai/logikai utasítások a regiszterekben tárolódnak. A regiszterek gyorsabbak, mint a memória-intenzív műveletek.
- További architektúra technikák: **pipe-line** (utasítás feldolgozás párhuzamosítása), többszörös adatvonalak, nagyszámú gyors regiszterek alkalmazásával.

RISC: „Pipe-line” technika

- **Utasítás szintű párhuzamosítás**: A soros feldolgozással ellentétben, egy feladat egymástól független részei (fázisai) a rendszer különböző pontjain egyszerre, egy időben hajtódnak végre, ezáltal növekszik a sebesség. Az operandusokat gyors regiszterekben tároljuk. Azonos hosszúságú utasítások gyors F-D-E eljárása. Egy ciklusban egyszerre történik különböző utasításrészek Fetch-Decode-Execute fázisok feldolgozása (gyors fetch és dekódolás).
- Többszörös adatvonalak párhuzamos végrehajtást engednek meg (hardveres párhuzamosítás). Tehát egy órajelciklus alatt több utasítást tudnak feldolgozni. (pl. SourceI-1,2, Destination adatbuszok a Motorola 88000 rendszerben.)

	1 fázis	**2 fázis**	**3 fázis**	…	…
1.utasítás	F	D	E	F	D
2.utasítás	-	F	D	E	F
3.utasítás	-	-	F	D	E

Motorola 88000 RISC rendszere



Hardveres párhuzamosítás:

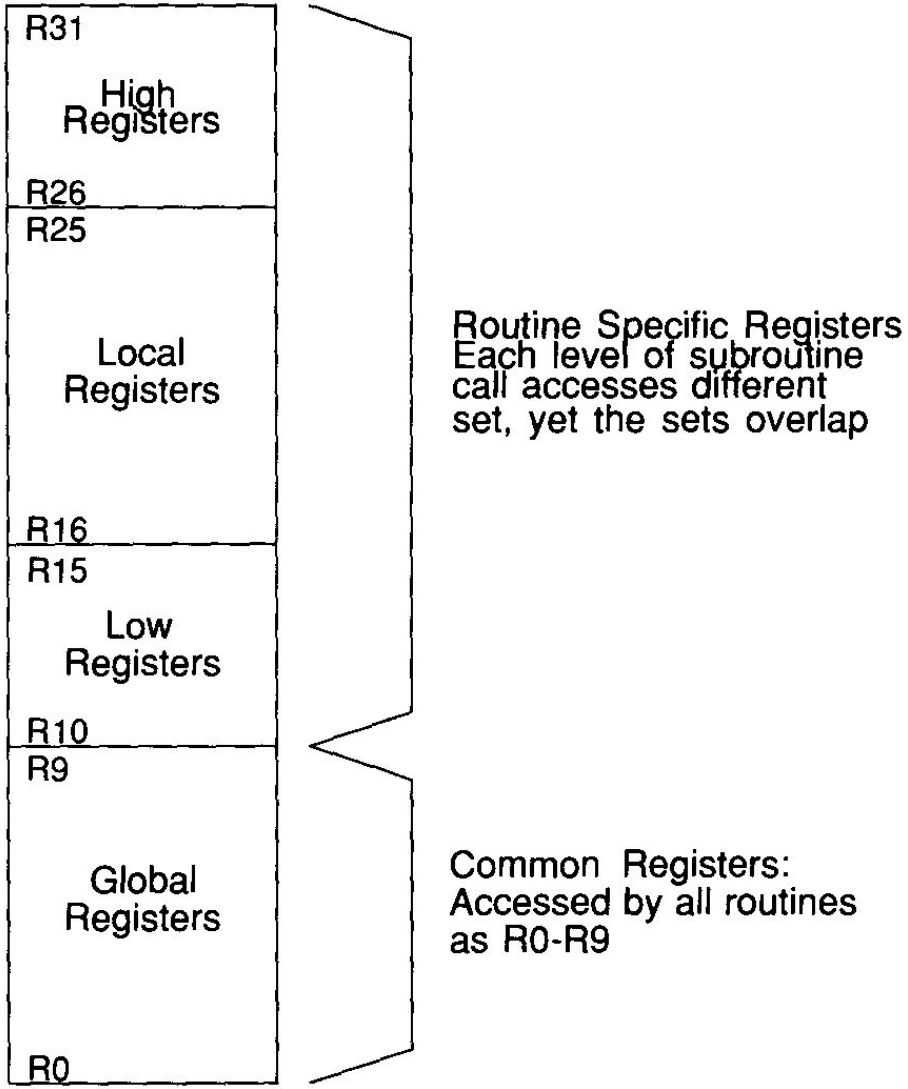
- Buszok
- Cache
- külön Data / Instruction Unit

(Harvard arch!)

- Közös MEM (Neumann arch!)

Berkeley RISC-I rendszere: Regiszter használat

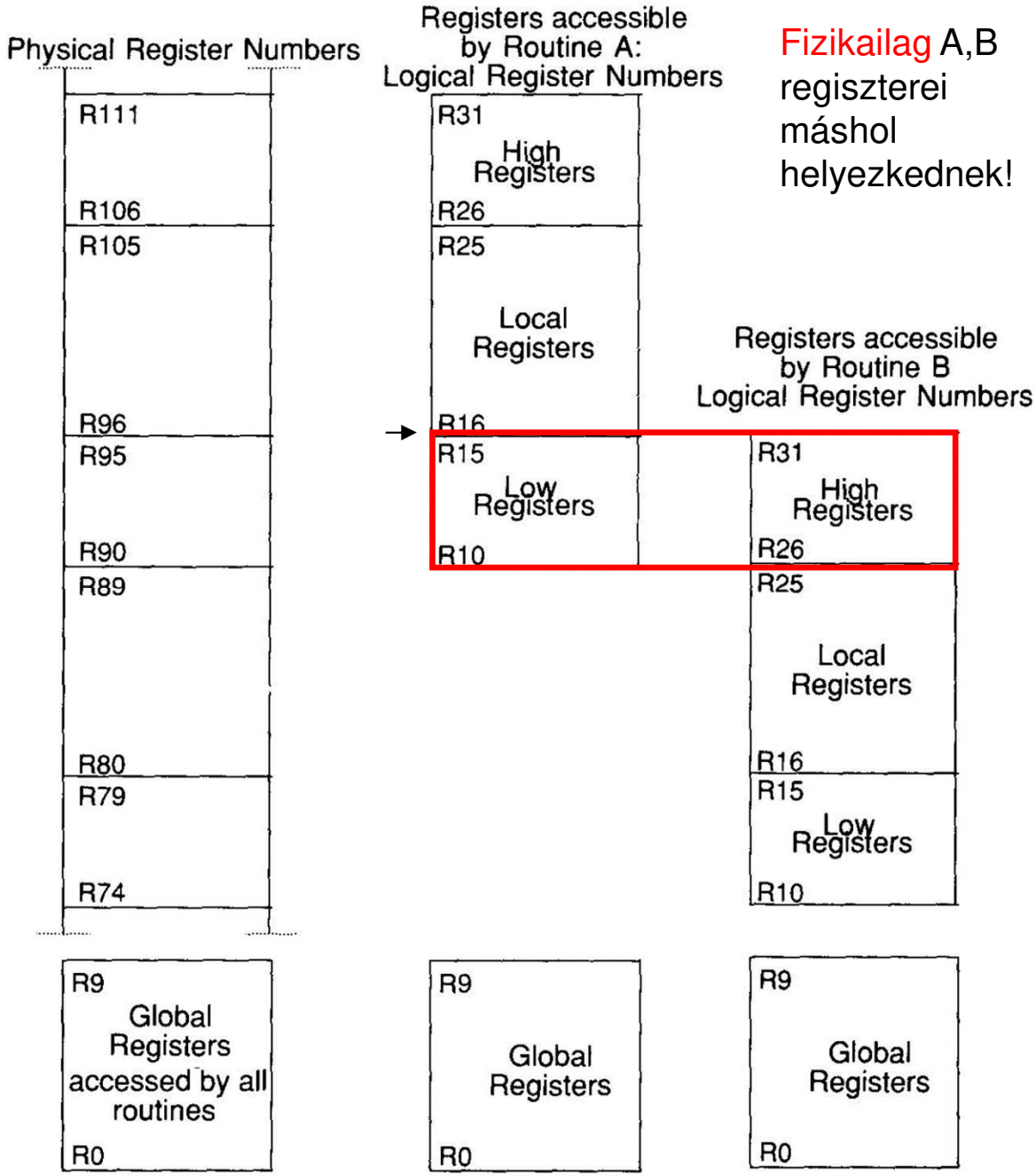
- Regiszter ablak: a szubrutin híváshoz (call) / visszatéréshez (ret) szükséges processzor-időt kívánták minimalizálni nagy számú regiszter használatával.
- Regiszteregeknek csak egy kis része érhető el („ablak”). Egy pointer mutat az ablakra, amely azonosítja a benne található aktuális regisztereket. Ha a szubrutinok között „átlapolódás” van az ablakban, akkor történhet paraméter átadás.



„Regiszter ablakozási„ technika

- Paraméter átadás „ablakozással”: a globális regisztereken keresztül történik, amelyet mindkét (A,B) szubrutin elérhet.

- 5 bit → 32 regiszter A(R0-R31) címezhető meg B(R0-R31)
- *közös (globális) regiszterek*: R0-R9, minden szubrutin által elérhetők
- *rutin specifikus regiszterek*: R10-R31 mely további három részből áll (a regiszterek között történhet átlapolódás!)
  - Alacsony-szintű regiszterek: R10-R15
  - Lokális regiszterek: R16-R25
  - Magas-szintű regiszterek: R26-R31
- Ez az eljárás mindaddig jól működik, ameddig a paraméterek száma kisebb a regiszterek méreténél, mivel nem igényel memória-intenzív Stack műveletet.



Fizikailag A,B regiszterei máshol helyezkednek!

CISC Processzorok jellemzői

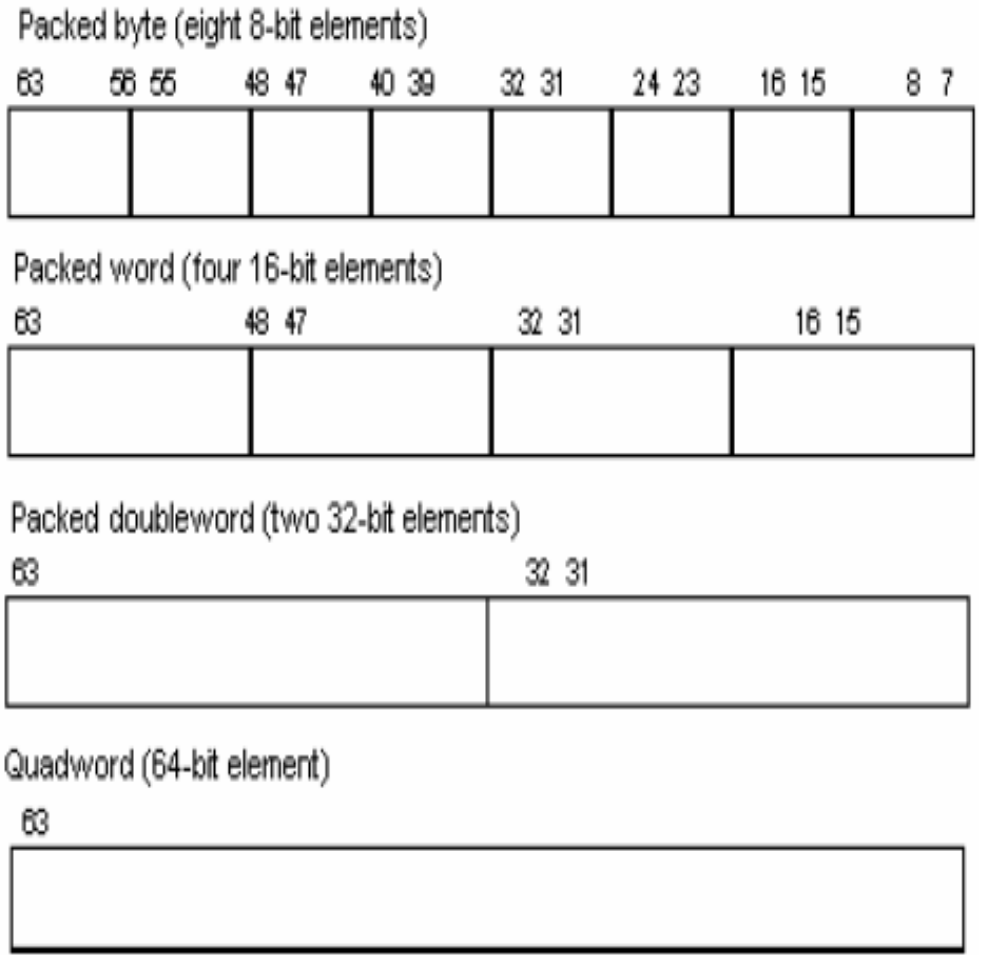
- **CISC:** *Complex Instruction Set Computer*
- *Nagyszámú utasítás-típust*, és címzési módot tartalmaz, egy utasítással több elemi feladatot végre tud hajtani. Változó méretű utasításformátum miatt a dekódolónak először azonosítania kell az utasítás hosszát, az utasításfolyamból kinyerni a szükséges információt, és csak ezután tudja végrehajtani a feladatát.
- A korai gépeknek egyszerű volt a felépítése, de bonyolult a nyelvezete. Összetett problémákat kívántak vele megoldani, a gépi kódnál magasabb szintű nyelven. *Szemantikus* rés= a gépi nyelv és felhasználó nyelve közötti különbség. Ennek áthidalására új nyelvek születtek: Fortran, Lisp, Pascal, C, amelyek bonyolultabb problémákat is egyszerűen képesek voltak kezelni. Komplexebb gépek születtek, amelyek gyorsak, sokoldalúak voltak.
- *Compiler = Fordító:* a bemenetén a probléma felhasználói nyelven van leírva, míg a kimenetén a megoldást gépi nyelvre fordítja le.
- Megfigyelték, hogy a processzor munkája során a rendelkezésre álló utasításoknak csak egy részét használja (pl. 20%-os használat, az idő 80%-ában).
- Ugyanaz a komplex program, függvény *kevesebb elemi utasítássorozattal* is megvalósítható. Memória, vagy regiszter alapú technikát használ.

CISC Processzorok jellemzői (2)

- Közvetlen memória-elérés (DMA) és összetett/bonyolult műveletek jellemzők rá.
- Mikro-programozott (mikrokódos) vezérlési mód:
  - a CISC processzor esetén a fordító (compiler) a programot egyszerűbb szintre fordítja, majd ezután a mikroprogram (ami meglehetősen összetett lehet) veszi át a vezérlést – mikroutasítások sorozata a mikrokódos memóriában.
- Példák:
  - System/360, VAX, DEC PDP-11/VAX rendszerei, Motorola 68000 család, és AMDx86-32/64 és Intel x86-32/64 CPUs

Pl. MMX kiterjesztés

- MMX: Multi-Media Extension (Intel Pentium sorozat 1996) –
  - SIMD: Single Instruction / Multiple Data alapú ***integer!*** stream data feldolgozásra (jelfeldolgozás)
  - 8 db MM0..7 regiszter (8 bit/reg)
  - Regiszterek adatait 4 különböző formátumban lehet tárolni (packet)
  - 57 MMX utasítás, 6 fő műveleti osztályban:
    - ADD
    - SUBTRACT
    - MULTIPLY
    - MULTIPLY THEN ADD (MAC – FIR)
    - COMPARISON
    - LOGICAL
      - AND, NAND, OR, XOR stb.



Pl. SSE, SSE2 kiterjesztés

Eredeti nevén **KNI**: ***K**atmai **N**ew **I**nstructions* (*első* Intel Pentium III-nál, 1999)

- SSE: Streaming SIMD extension (lebegőpontos és fixpontos adat folyamra) // Intel, AMD
- 32-bites módban 8 db, de már 128-bites regiszter csomag
- SSE-1:
  - 128-bit packed IEEE *single-precision* floating-point operations (~70 utasítás).
  - 2 clock cycles
- SSE-2:
  - 128-bit packed IEEE *double-precision* SIMD floating-point (~144 utasítás)
  - 128-bit packed integer SIMD operations
    - support 8, 16, 32, and 64-bit operands
  - 2 clock cycles
- MA: SSE-4.2v.