



# Digitális Rendszerek és Számítógép Architektúrák

4. előadás: Utasítás végrehajtás folyamata:  
címezési módok, RISC-CISC processzorok

Előadó: Dr. Vörösházi Zsolt  
[voroshazi.zsolt@virt.uni-pannon.hu](mailto:voroshazi.zsolt@virt.uni-pannon.hu)

# Jegyzetek, segédanyagok:

- Könyvfejezetek:

- <http://www.virt.uni-pannon.hu>

- Oktatás → Tantárgyak → Digitális Rendszerek és Számítógép Architektúrák (nappali)

- ([chapter04.pdf](#))

- Fóliák, óravázlatok .ppt (.pdf)

- Frissítésük folyamatosan



# Utasítás végrehajtás folyamata

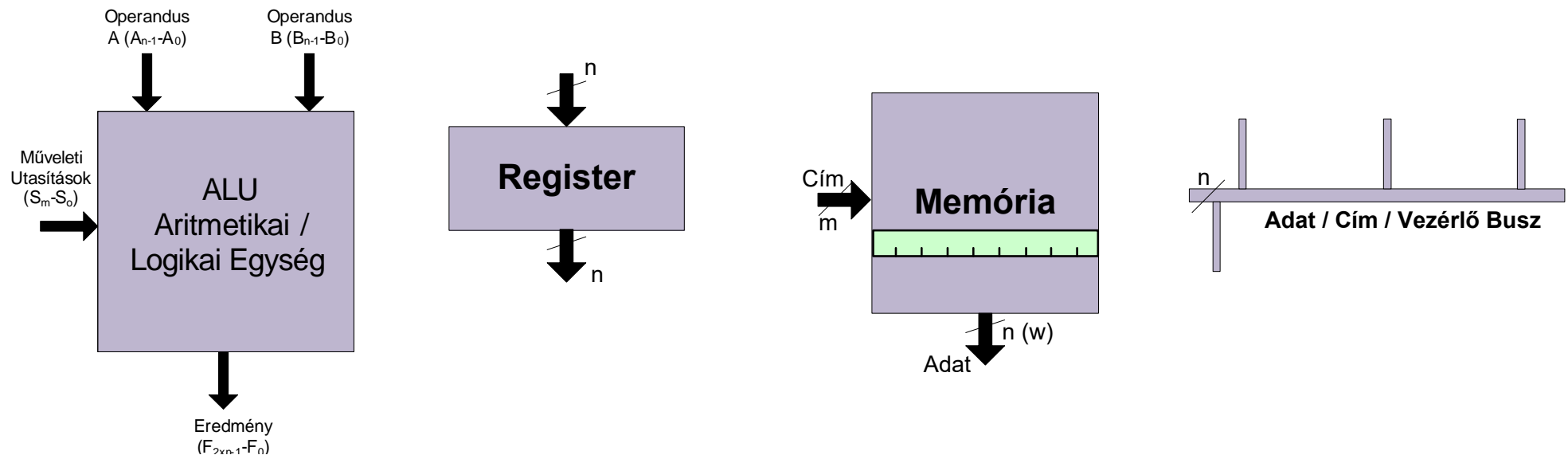
- Utasítás kódok
  - Programvezérlő utasítások
- Címzési módok
- RISC vs. CISC processzor architektúrák



# Alapvető digitális építőelemek

# Legfontosabb digitális építőelemeink:

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



# ALU egység

- Az **ALU** egység két különböző  $n$ -bites bemeneti résszel (A, B) rendelkezik, és egy  $n$ -bites\*\* kimeneti vonallal (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 címvezetékekkel címezhető meg. Az adatbuszon kétirányú (írás/olvasás) kommunikáció is megengedett. Memória a *von 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

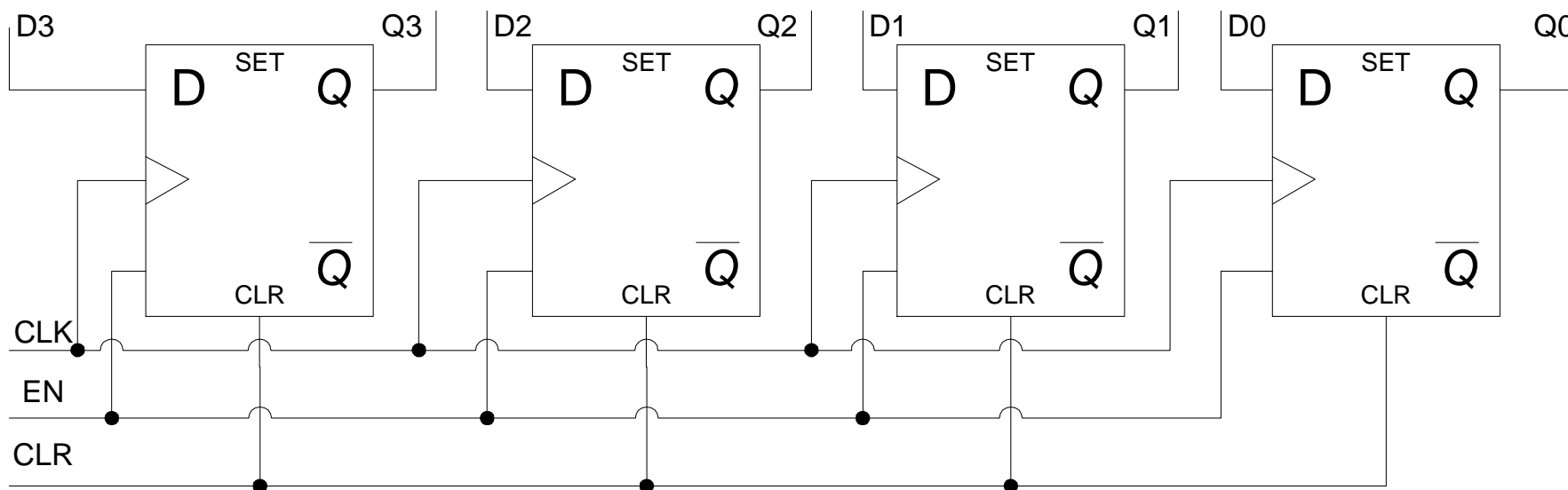
- Másik alap építőelem az **adatbusz** vagy **adatút (datapath)**. Fontos paraméter a szélessége: egy  $n$  természetes szám.
- Az adatutak pont-pont összeköttetéseket jelentenek különböző méretű és sebességű eszközök között.
- A közvetlen kapcsolat nagy sebességet, de egyben rugalmatlanságot is jelent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



# Regiszterek

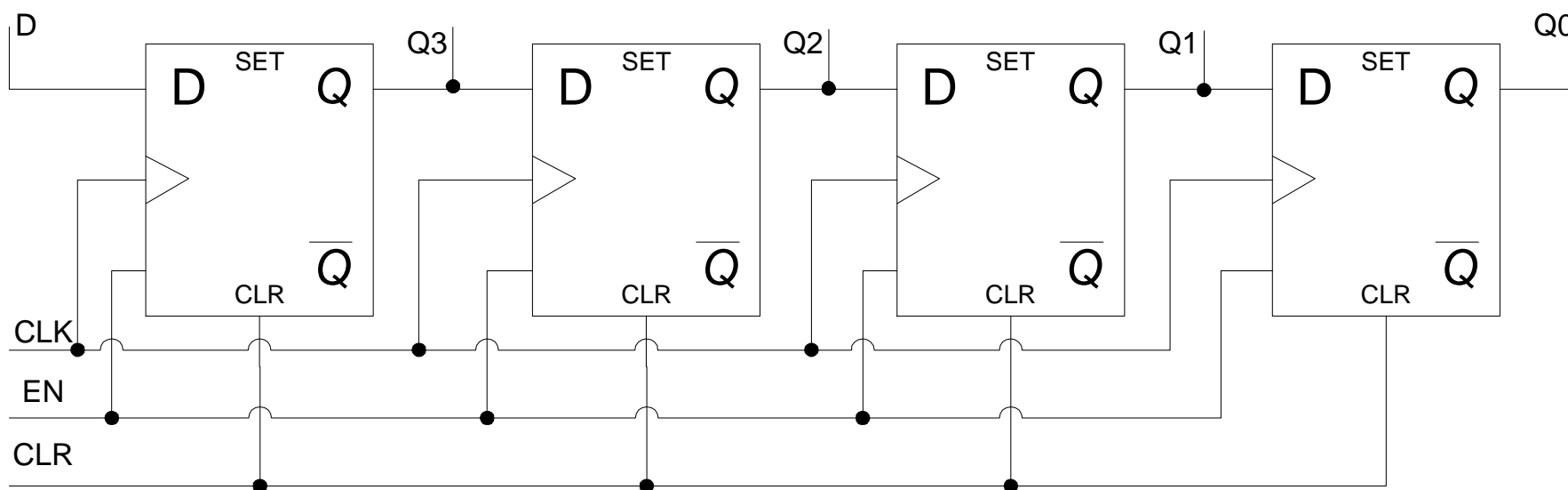
- A következő fontos elem a **regiszter**. Olyan szélesnek kell lennie, hogy benne, a buszokról, memóriákból, ALU-ból érkező információ eltárolható legyen. Adott vezérlőjelek hatására a bemenetén lévő adatokat betölti, és ideiglenesen eltárolja. Más vezérlőjelek hatására a kimenetére rakja a tárolt adatokat, vagy például egy vezérlőjel hatására, *lépteti (shift-el)* a benne lévő adatokat.

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



Katalógus adat: [SN54/74LS175](#)

# 4-bites Shift/léptető regiszter (Serial in/Parallel Out – D-tárolós)



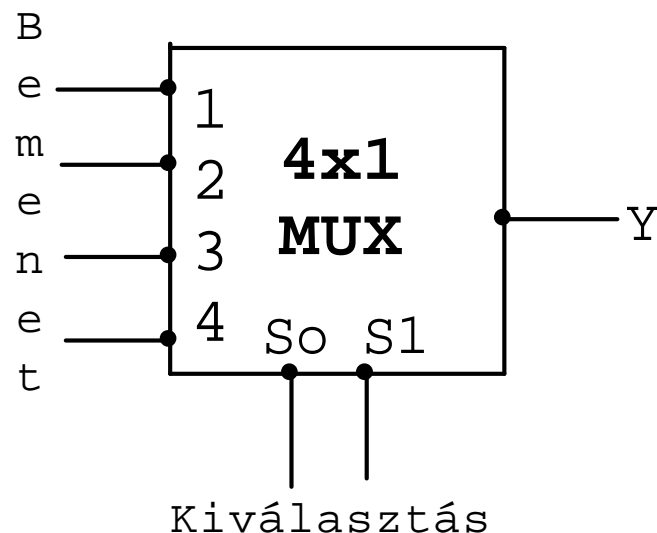
Katalógus adat : SN54/74LS95

# További digitális építőelemek

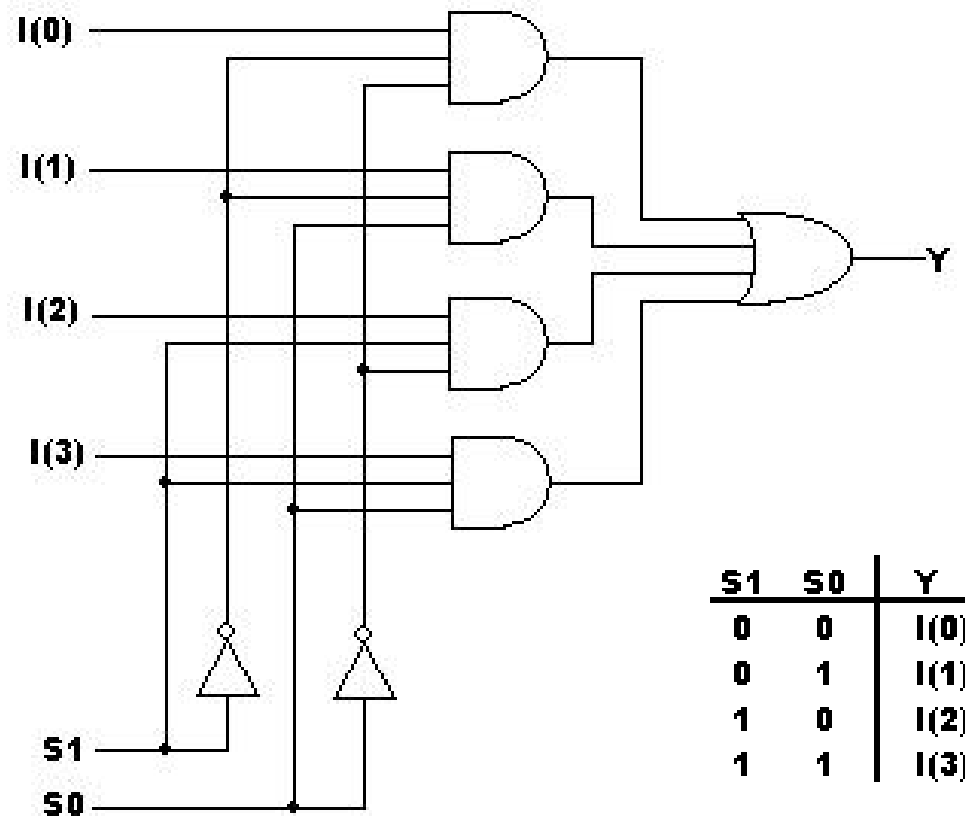
- a.) MUX
- b.) DEMUX
- c.) Kódoló (encoder)
- d.) Dekódoló (decoder)
- e.) Komparátor (Comparator)

# a.) Multiplexer (MUX)

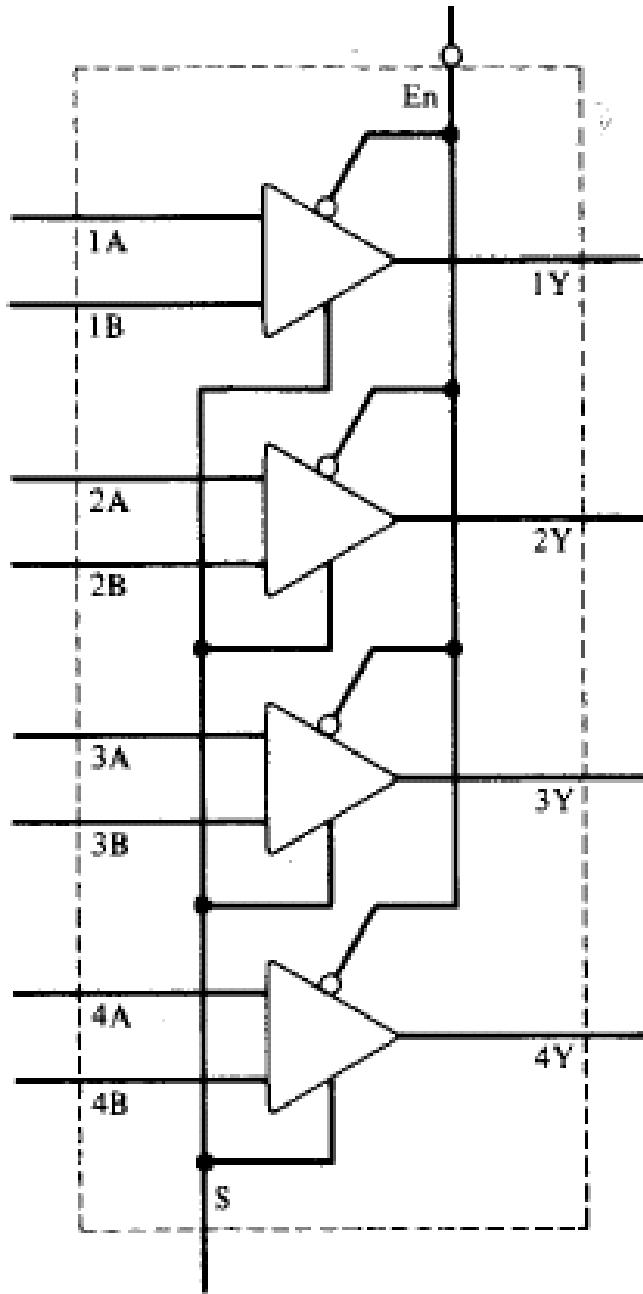
- N kiválasztó jel  $\rightarrow 2^N$  bemenet, 1 kimenet
- Példa: 4:1 MUX



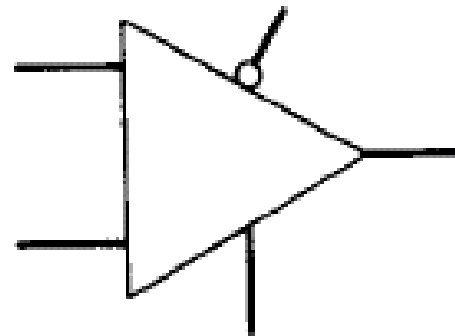
$2^N$  számú bemenet közül választ egyet (Y), mint egy kapcsoló.  
Rendelkezhetsz EN bemenettel is.



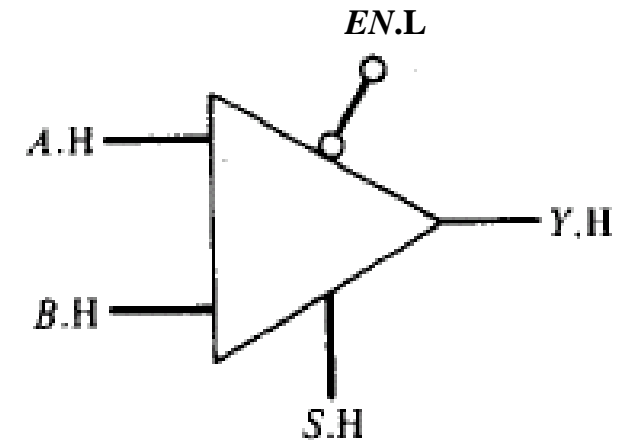
# TTL'74LS157 pl. 2:1 MUX



- Quad (4-bites) 2-bemenetű 2:1 MUX-ból áll.
- ← Közös S, EN jelek.
- 4 db Y(1,2,3,4) kimenet



2:1 MUX szimbóluma

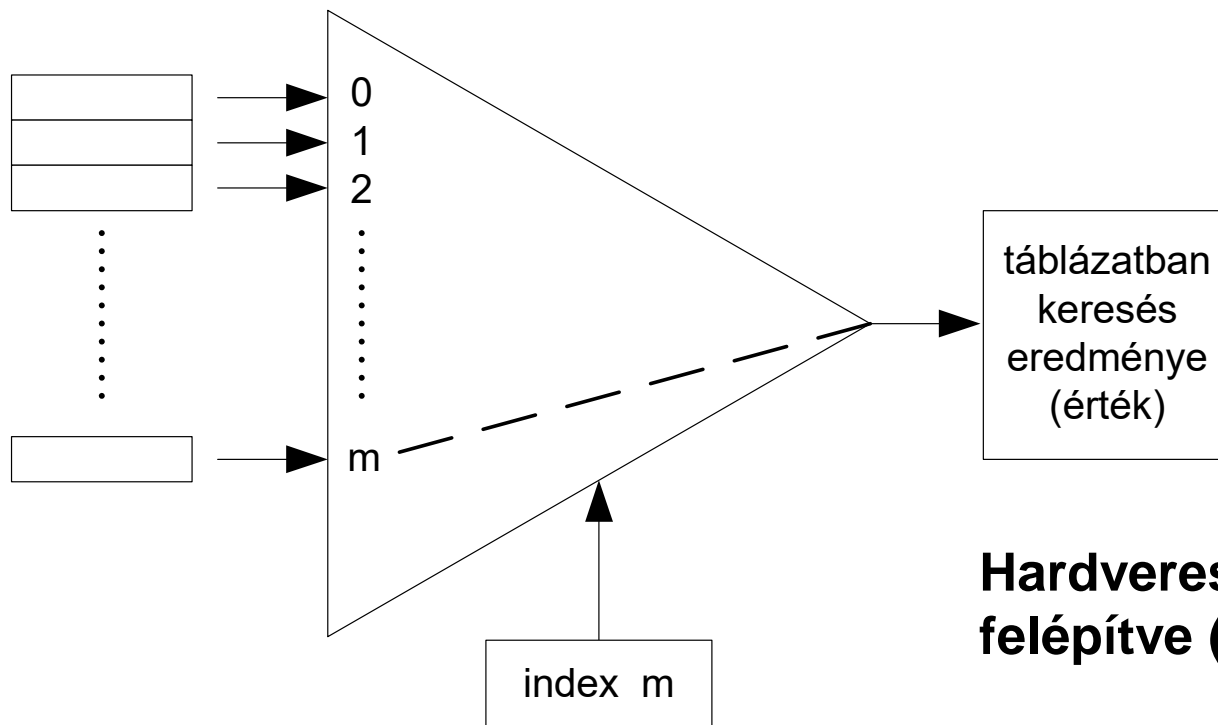


2:1 MUX áramköri  
szignál jelölésekkel

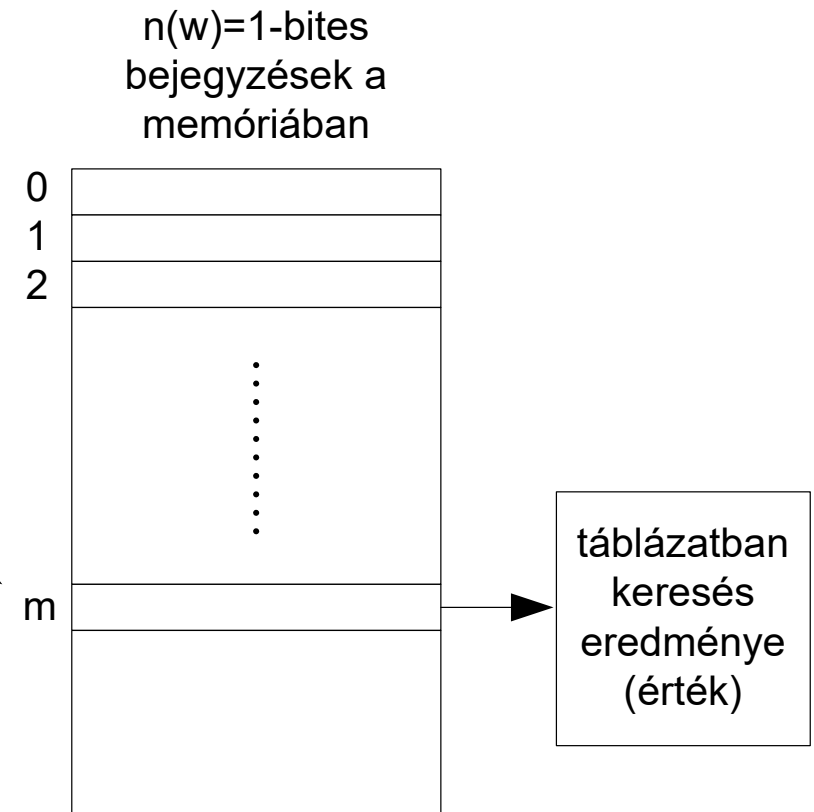
$$Y = EN \cdot (A \cdot \bar{S} + B \cdot S)$$

# PI. LUT megvalósítások:

## Szoftveres Look-up-Table

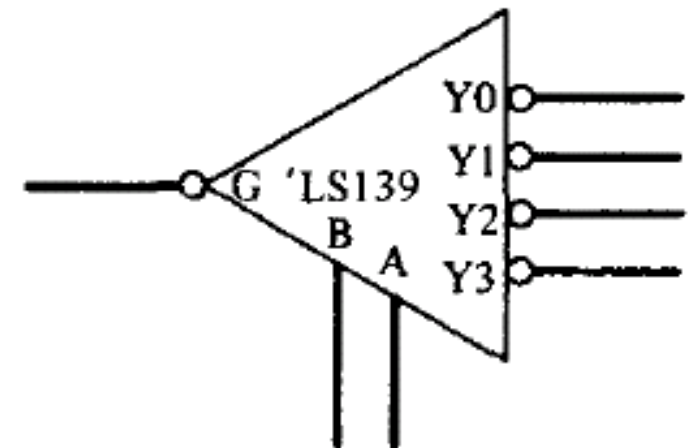


**Hardveres Look-up-Table, MUX-ból felépítve (1 bites táblázatkeresés)**



## 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 !**

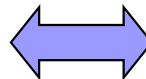


# Példa - 1:4 Demultiplexer (folyt)

## ■ Kanonikus táblázat

Demultiplexer logika						
G	B	A	Y0	Y1	Y2	Y3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

T=L !



## Feszültség-logikai tábl.

74'LS139 feszültség-logika						
G.L	B.H	A.H	Y0.L	Y1.L	Y2.L	Y3.L
H	x	x	H	H	H	H
L	H	H	L	H	H	H
L	H	L	H	L	H	H
L	L	H	H	H	L	H
L	L	L	H	H	H	L

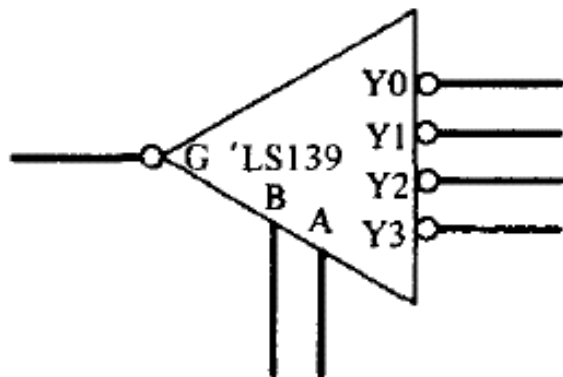
## ■ Demultiplexer logikai egyenletei:

$$Y0 = \overline{B} \cdot \overline{A} \cdot G$$

$$Y1 = \overline{B} \cdot A \cdot G$$

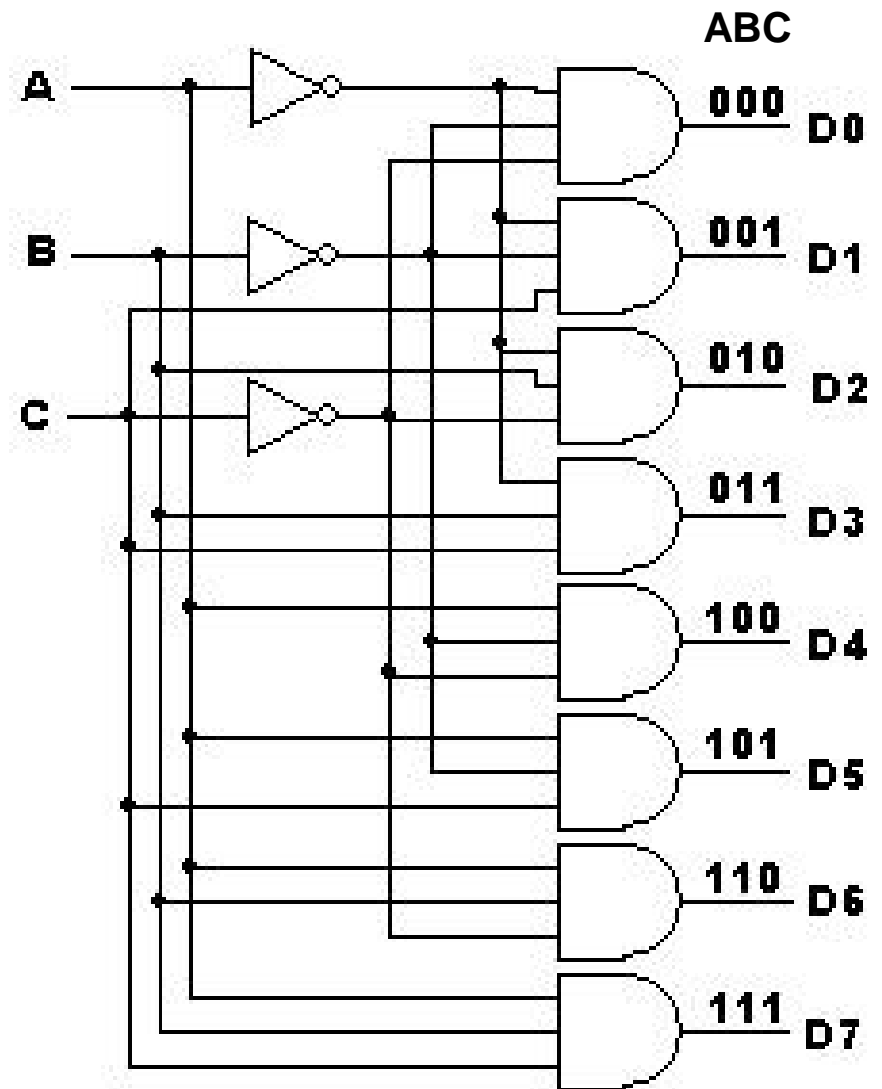
$$Y2 = B \cdot \overline{A} \cdot G$$

$$Y3 = B \cdot A \cdot G$$



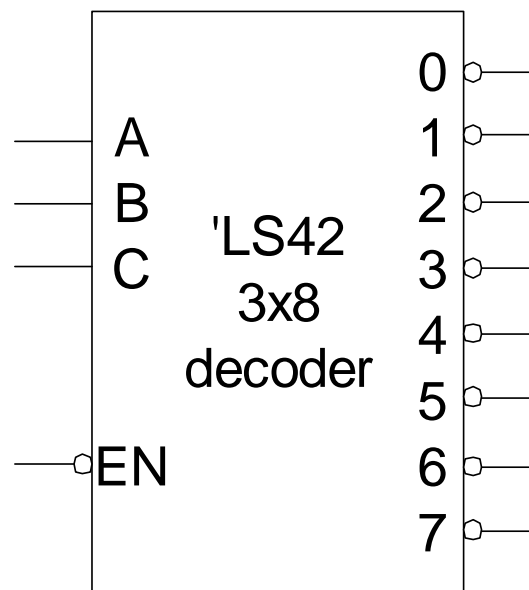
## c.) Dekódoló áramkörök

- N bemenet esetén  $2^N$  kimenete van
- N bemenetből mindig csak egy aktív logikai értékű
- Példa: 3→8 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**

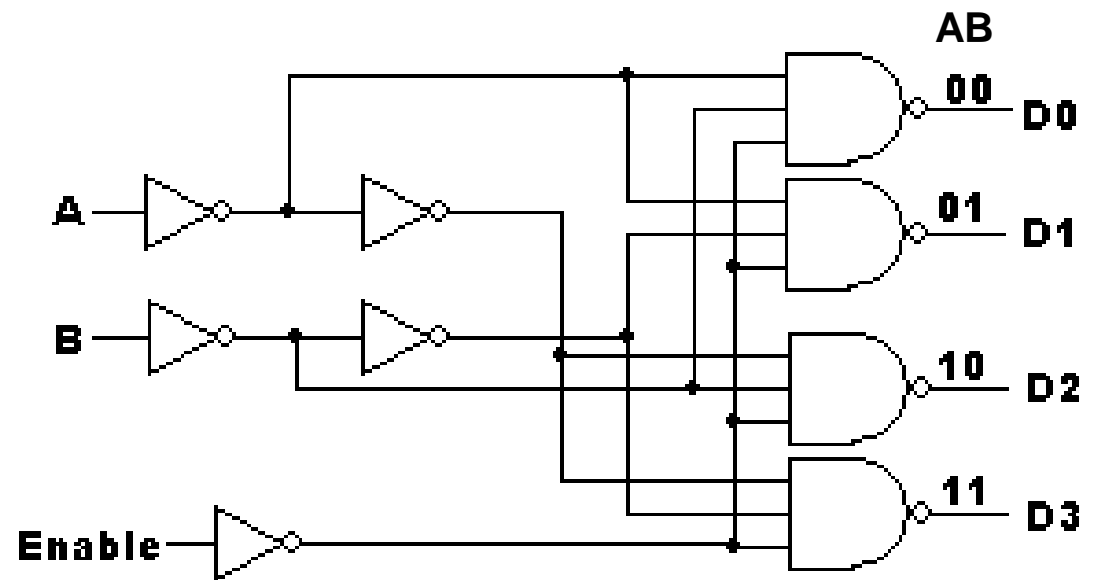


Mixed logic  
szimbólum

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 aktív logikai értékű egyszerre
  - **Priority encoder:** több bemenete is lehet aktív logikai értékű egyszerre, de azok közül ált. a legnagyobb bináris értékű, azaz prioritású bemenethez generál kódot! (pl. kód: address, index is lehet)
    - I/O, IRQ jelek generálásánál használják leggyakrabban

# Probléma: Priority encoder esetén

- Mi van akkor, ha még sincs igaz bemenete (mindegyik hamis)? Két megoldás van:
  - 1.) *módszer*: Input vonalak megszámozása 1-től (D1) kezdődően, és a 0 kimeneti kód (itt FF) jelenti, hogy mind „hamis” volt. (Továbbá: X – don't care)
  - 2.) *módszer*: input vonalak megszámozása 0-tól (D0) kezdődően, és egy külön vezérlőjelet (W) biztosítani arra, hogy nincs „igaz” bemenet. (Továbbá: X – don't care)

Method 1				
D3	D2	D1	B	A
F	F	F	F	F
F	F	T	F	T
F	T	X	T	F
T	X	X	T	T

Method 2						
D3	D2	D1	D0	B	A	W
F	F	F	F	-	-	T
F	F	F	T	F	F	F
F	F	T	X	F	T	F
F	T	X	X	T	F	F
T	X	X	X	T	T	F

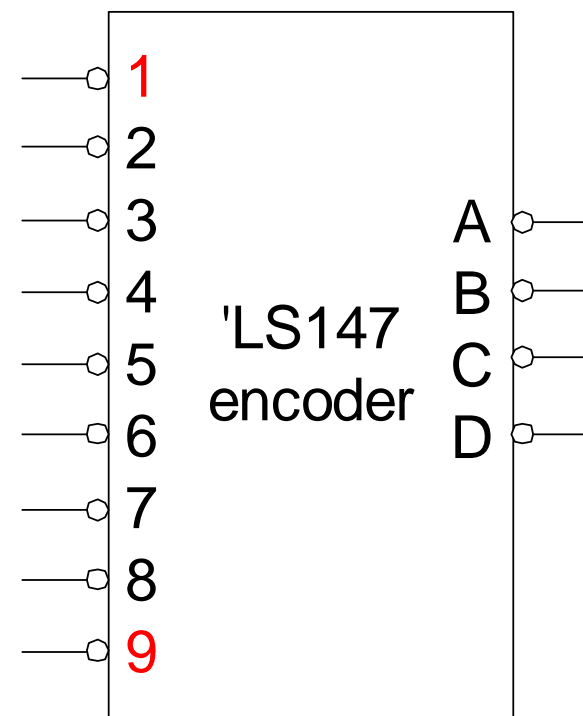
# TTL'74LS147 Priority encoder - kódoló áramkör

- 10-input, 4-output encoder

- „0” nincs jelölve: amikor az összes bemenet False (lefoglalt)

- Alkalmazás:

- Cím, indexgenerálás
    - LUT választás



Mixed logic  
szimbólum

T=L!

## e.) Komparátor

- Logikai kifejezés – *referencia* kifejezés (bináris számok) *aritmetikai kapcsolatának* megállapítására szolgáló eszköz.

- PI: 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:

$$A.EQ.B = (A_0 \odot B_0) \cdot (A_1 \odot B_1) \cdot \dots \cdot (A_n \odot B_n)$$



# Ismétlés: EQ/XNOR/Coincidence operátor

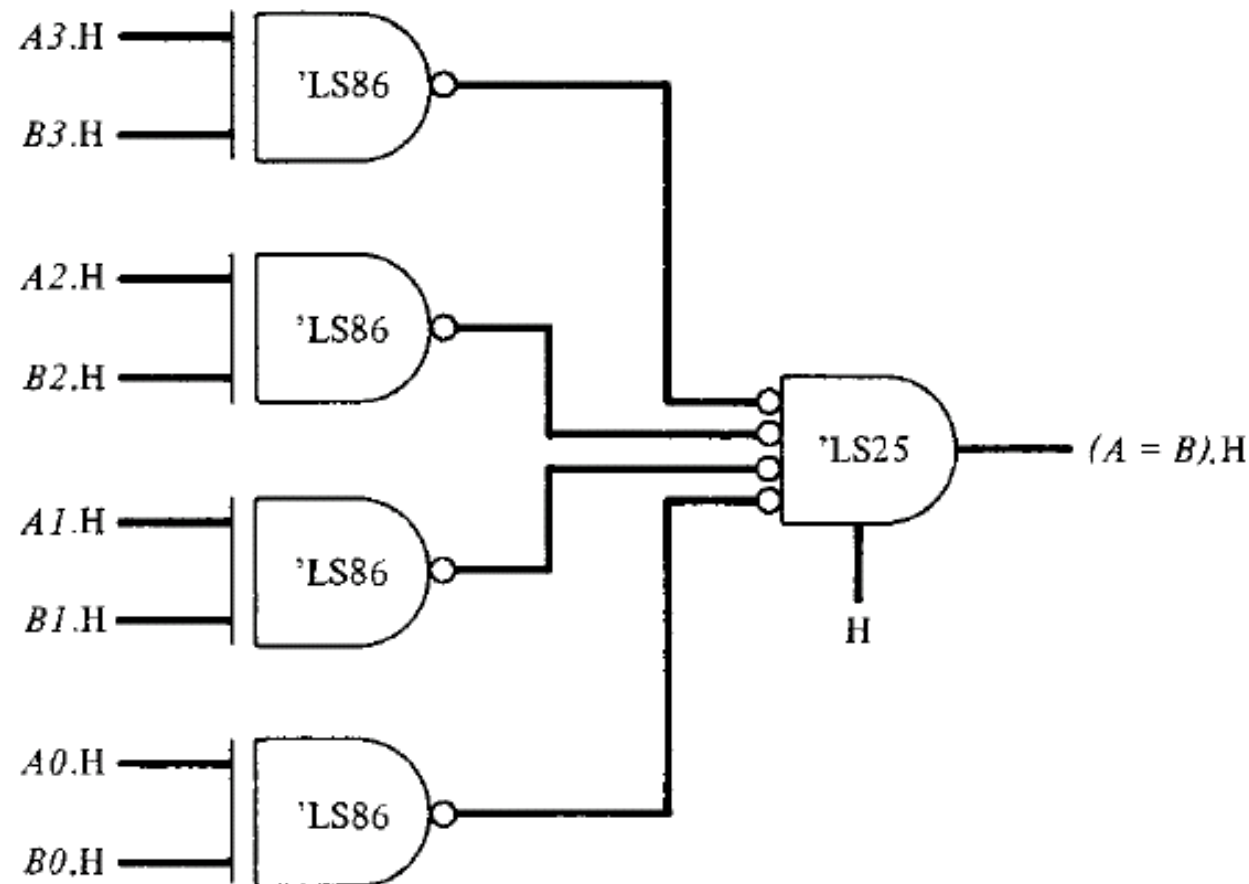
- Logikai egyenlet:  $A.EQ.B = A \odot B = A \cdot B + \overline{A} \cdot \overline{B}$
- Referenciabit szerinti megkülönböztetés:
  - ha a referencia bit (B), amihez hasonlítunk **konstans**
  - ha a referencia bit (B) egy **változó** mennyiség
- Példa: ha B referencia konstans -> egyszerűsítése A-nak
$$A.EQ.B = A \text{ if } B = T$$
$$A.EQ.B = \overline{A} \text{ if } B = F$$
- Példa: legyen B egy 4-bites konstans mennyiség (B=TFFT), és A tetszőleges, akkor:

$$A.EQ.B = A_0 \cdot \overline{A_1} \cdot \overline{A_2} \cdot A_3$$

# Példa: 4-bites komparátor

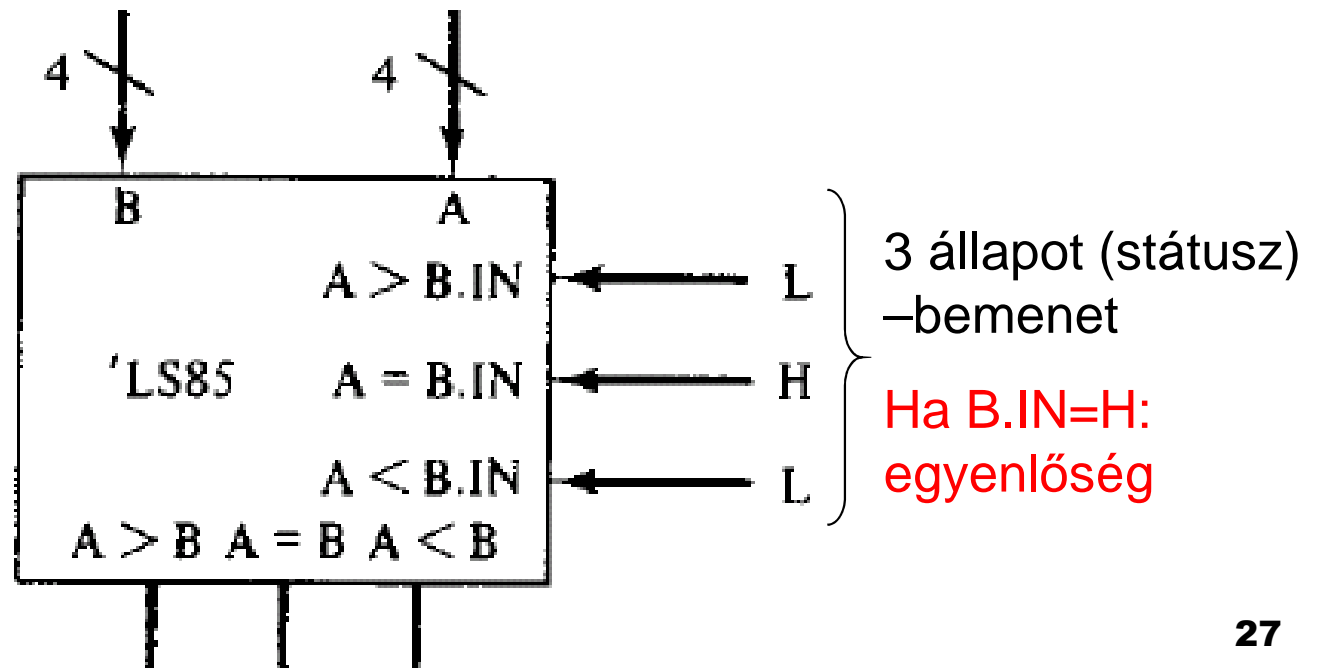
- Mixed-logic kapcsolási rajza, és log.egyenlete:

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



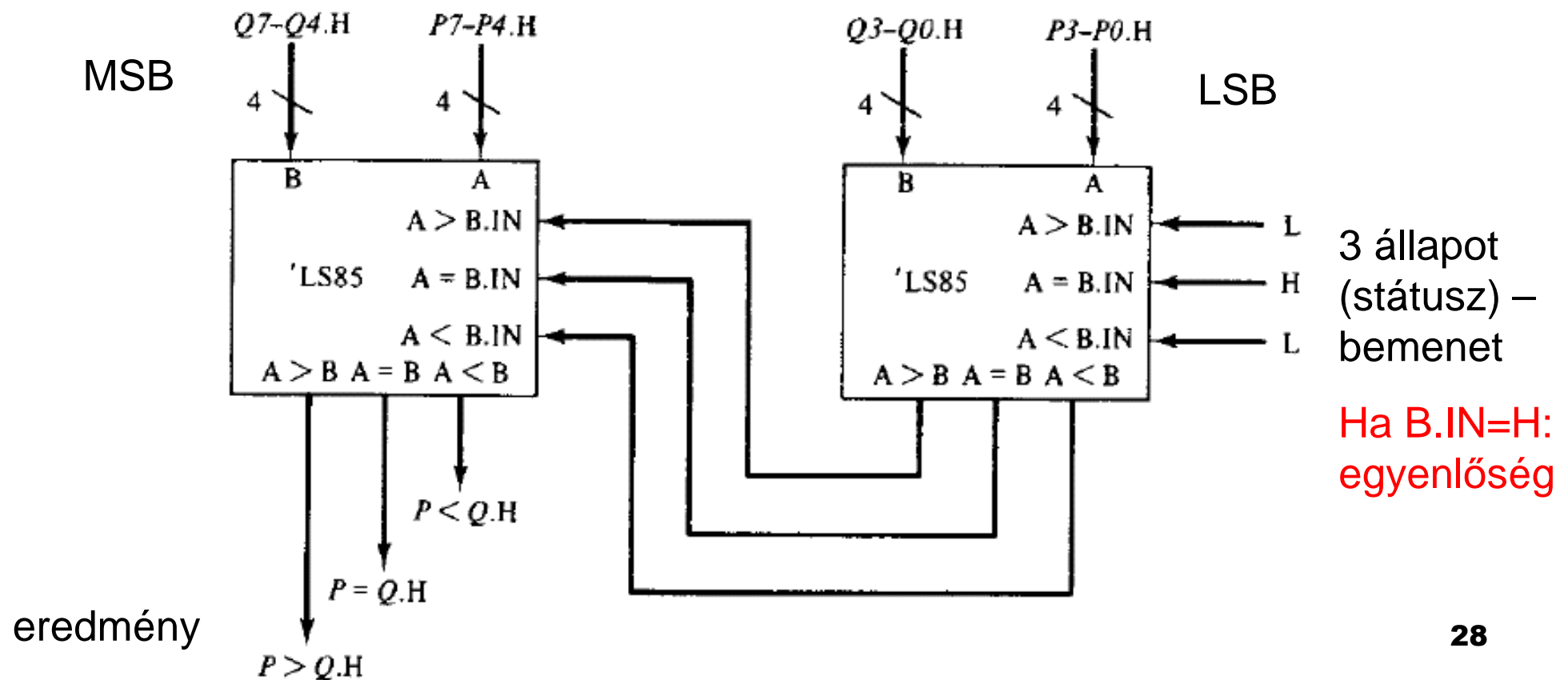
# '74LS85 4-bit Magnitude Comparator

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



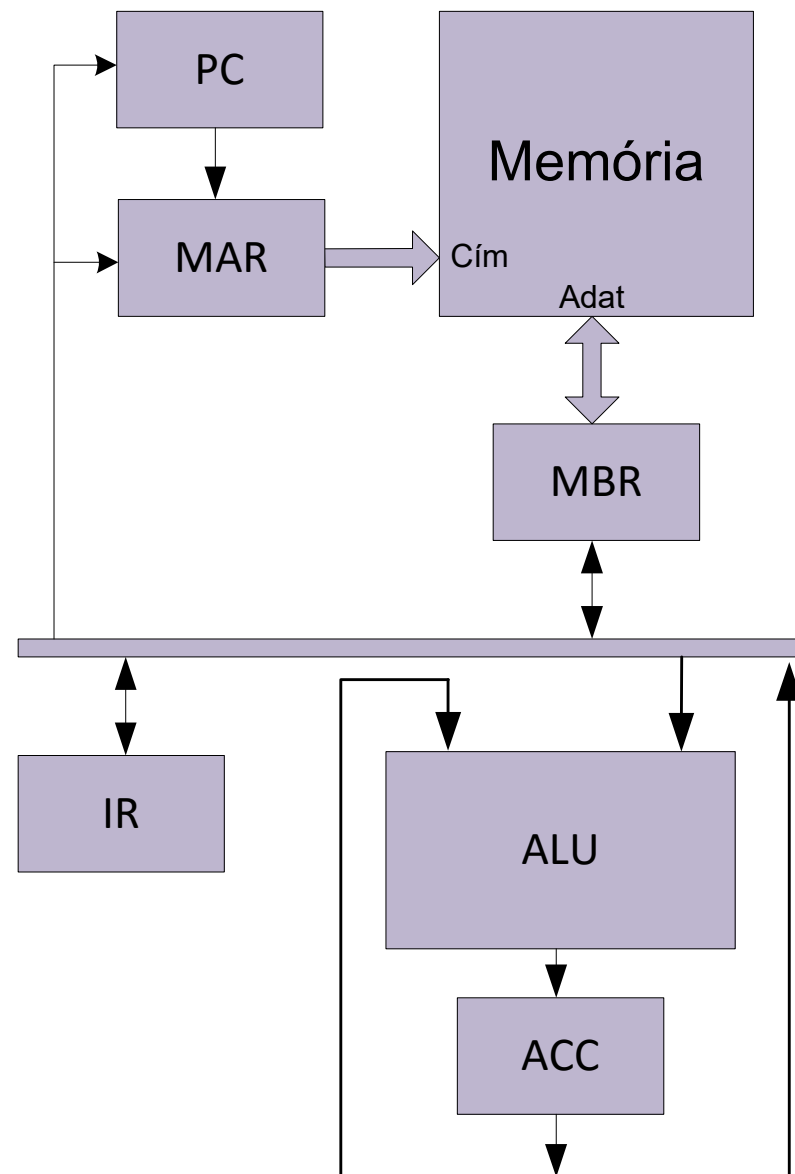
# Példa: 8-bites Magnitude Comparator – Pl. egyenlőség esetén

- Kettő 4-bites '74LS85 Magnitude komparátor sorbakötéséből („cascading”) kapjuk a 8 bites (P,Q) értékek összehasonlítását



# 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).





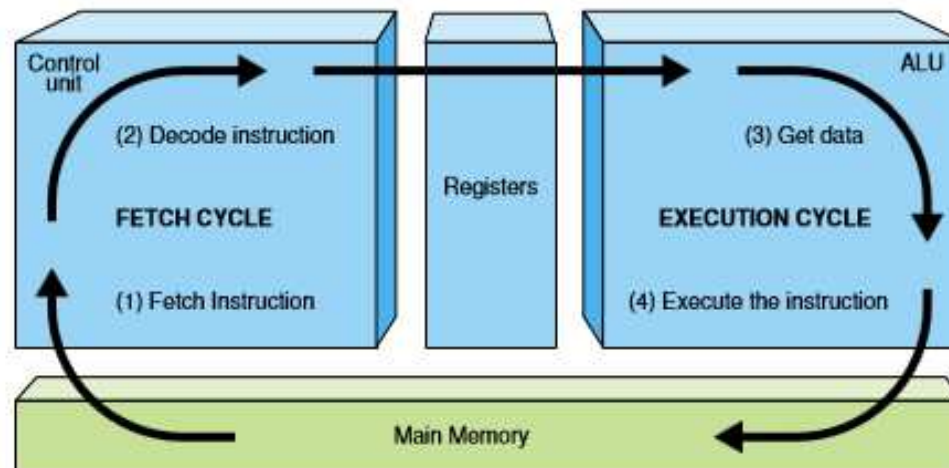
# Utasítások kódolása

# 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 adatátvitelhez 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
- **M[MAR] → MBR**: MAR címregiszter tartalmával címezzük meg az M memória adott celláját, melynek tartalma az MBR regiszterbe kerül
- **MBR → IR**: MBR tartalma az IR-be töltődik. Ha az adatbusz megenged többszörös műveletvégzést egyidejűleg, akkor az egyes akciók összekapcsolhatók!
- **IR <3:0> → ALU**: Az információnak csak egy része, az IR regiszter 3-0 bitje töltődik az ALU-ba
- **REG[2] → MEM[MAR]**: Általános célú regiszter 2. rekesze töltődik a Memória Cím Regiszter adott rekeszébe, a MAR által mutatott címre mutat az operatív memóriában
- **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, ALU műv.)  
[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]
- ...

# Példa: ADD utasítás RTL leírása

*Fetch: (regiszterek feltöltése, utasításhívások):*

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

*Decode: (a dekódolást általában 0 idejűnek feltételezzük)*

*Execute:(végrehajtás)*

<b>IR &lt;addr&gt;</b> → <b>MAR</b>	operandus címét a MAR-ba töltjük
<b>M[MAR]</b> → <b>MBR</b>	ezt az értéket kell az ACC-vel összeadni
<b>ACC+MBR</b> → <b>ACC</b>	összeadás (eredmény az ACC-ben)

Időszükségletek itt még nincsenek feltüntetve!

# ADD3 assembly utasításokkal

■  $z = x + y$

Megfelel: ADD3 R<sub>1</sub>, R<sub>2</sub>, R<sub>3</sub>  
(3-című utasításnak)

LD        x, \$R<sub>1</sub>

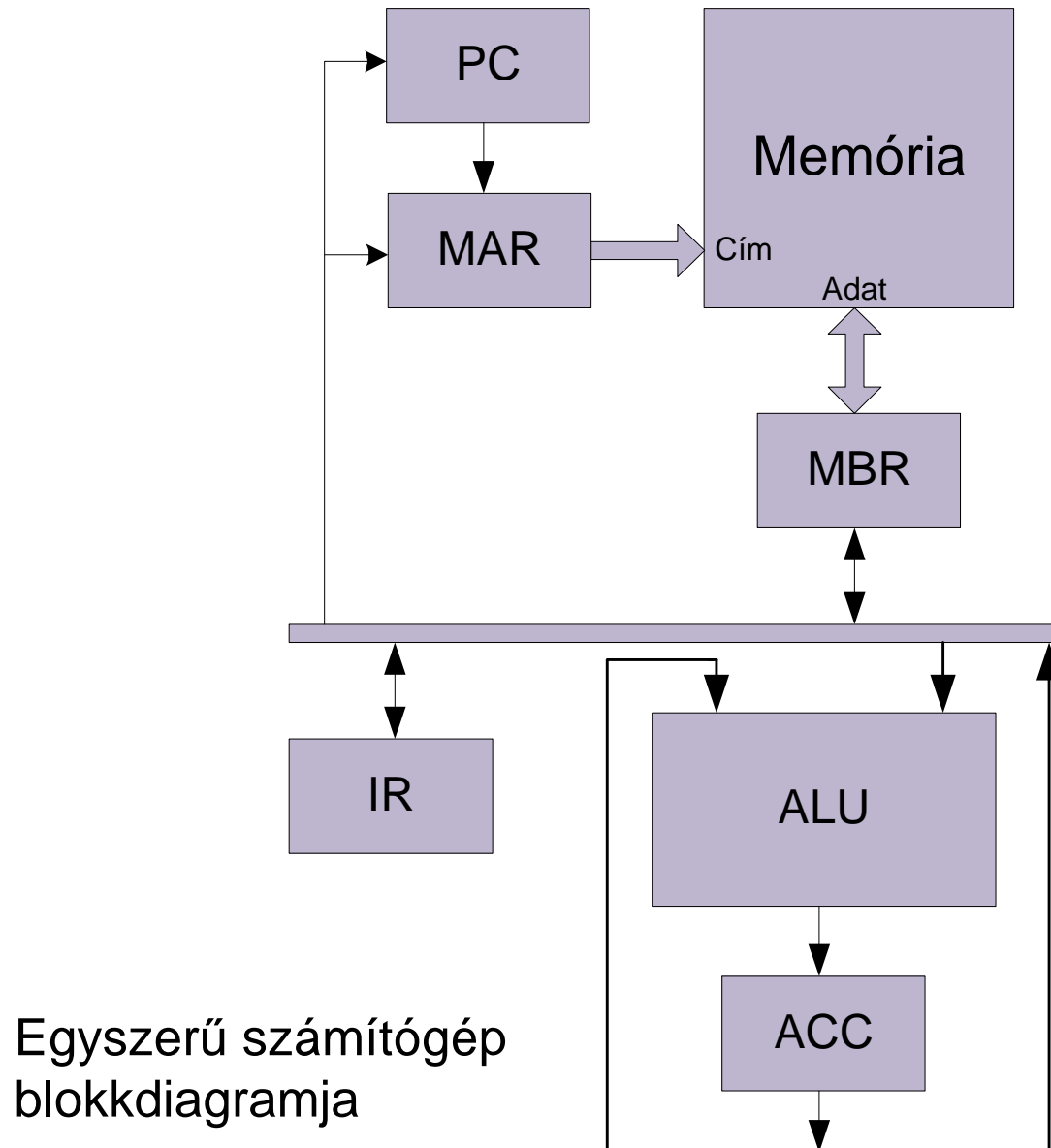
LD        y, \$R<sub>2</sub>

ADD       \$R<sub>1</sub>, \$R<sub>2</sub>, \$R<sub>3</sub>

ST        \$R<sub>3</sub>, z

# Egy-című gépek (Egyszerű számítógép)

Példa: DEC PDP-8  
számítógépe



**Neumann architektúra!**

# Egy-című gép

- **Megadása:** [operátor],[[operandus](#)]
- A műveletekhez csak 1 operandus szükséges. Ilyen művelet lehet például: 1's vagy 2's komplementus 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, és a másik operandust *egyetlen címmel* azonosítjuk.

# Példa: Egy-című gép

## 2's komplementens 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)*

**$\overline{ACC}$ →ACC**

ACC komplementensét 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 X) egy operandusra

*Fetch: (regiszterek feltöltése, utasításhívások):*

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

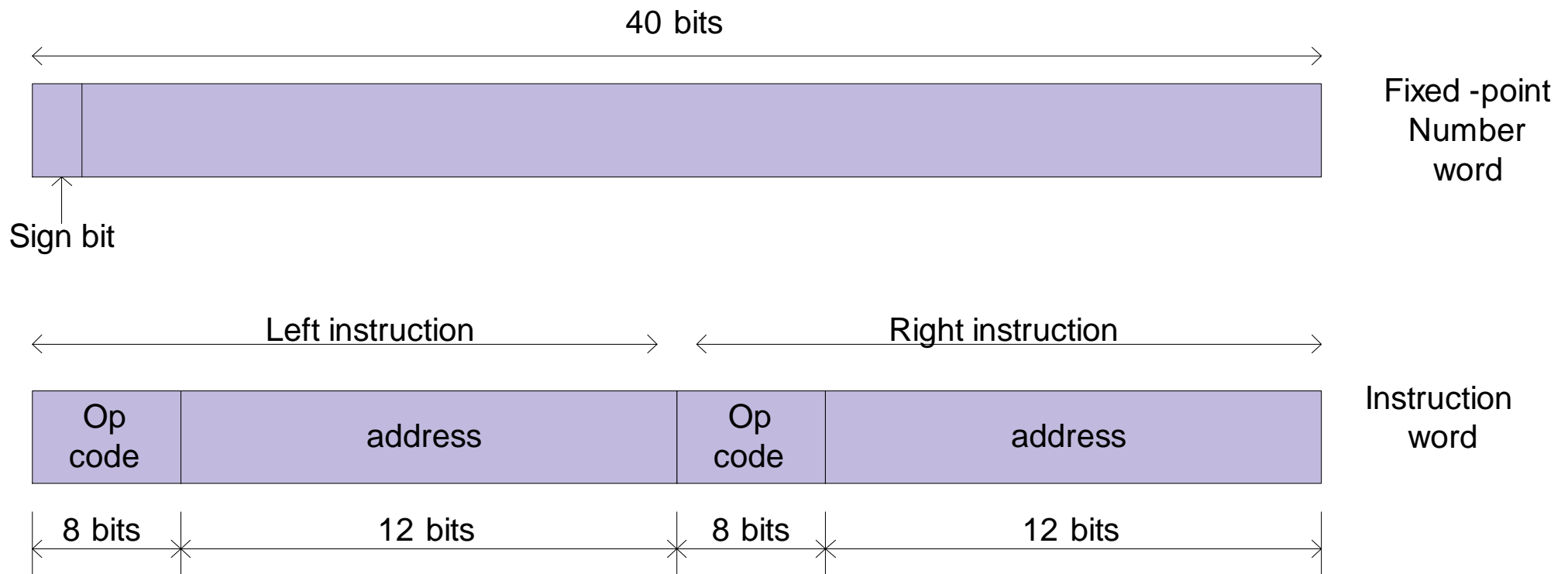
*Decode: (a dekódolást általában 0 idejűnek feltételezzük)*

*Execute: (végrehajtás)*

<b>X→MAR</b>	X operandus címét a <i>közvetlenül</i> a MAR-ba töltjük
<b>M[MAR]→MBR</b>	X címén lévő értéket az MBR-be tesszük
<b>ACC – MBR→ACC</b>	ACC-ből kivonjuk az X-et, és ACC-be töltjük

Időszükségletek itt még nincsenek feltüntetve!

# Példa: IAS adat és utasításformátum (Neumann – 1947)



40 bites adat-szóhosszúság

2x20 bites utasításhossz: 8-bit -> 256 művelet, 12 bit-> 4096  
memóriacím érhető el

# Példa: 1-című gép (Kivonás SUBX)

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

# Példa: 1-című gép (kivonás SUBX)

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

*Fetch: (regiszterek feltöltése, utasításhívások):*

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

*Decode: (a dekódolást általában 0 idejűnek feltételezzük)*

*Execute: (végrehajtás)*

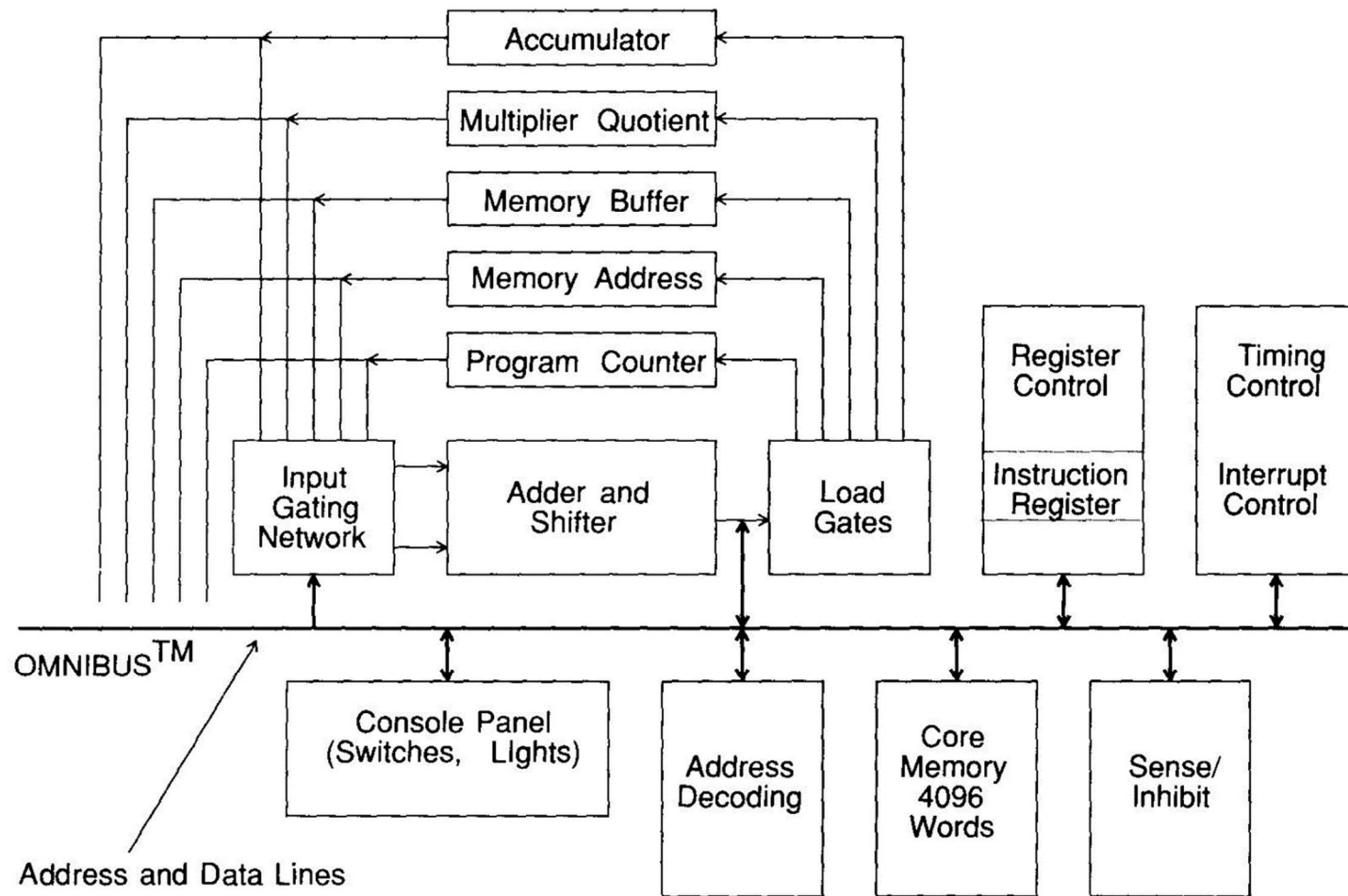
<b>PC→MAR</b>	[5ns] PC-vel a következő címre mutatunk
<b>M[MAR]→MBR</b>	[30ns] Ezt címet az MBR-be tesszük
<b>MBR→MAR</b>	[5ns] Ez a cím lesz az X operandus címe
<b>M[MAR]→MBR</b>	[30ns] Címen lévő értéket az MBR-be töltjük
<b>PC+X_len →PC</b>	[5ns] X operandus címének hosszával növeljük a PC-t
<b>ACC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

---

**Σ 135ns**

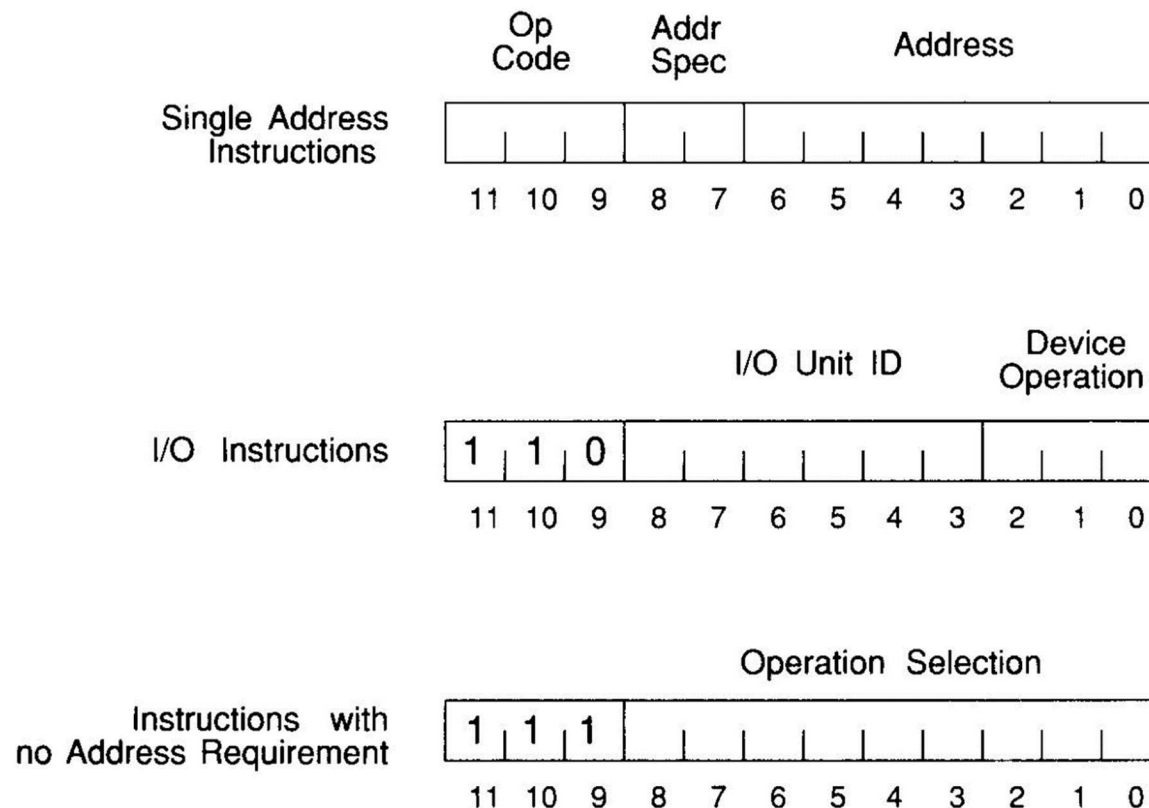
# Példa: DEC PDP 8 (egycímű gép)

- 12-bites szóhossz: 3-bites opcode (8 művelet) + 9-bit utasítás cím (speciális operandus címezési módokat tett lehetővé)



# DEC PDP-8 utasításformátuma

- 3-bites opcode [11:9]: 8 művelet , ebből:
  - 6-nak van saját címe (itt az alsó 9 bit adja a címet) pl: Add, And, Jmp, Inc\_Skip\_IfZero, DepositClearAcc
  - 2-nek nem kell cím: pl. ACC műveletekhez

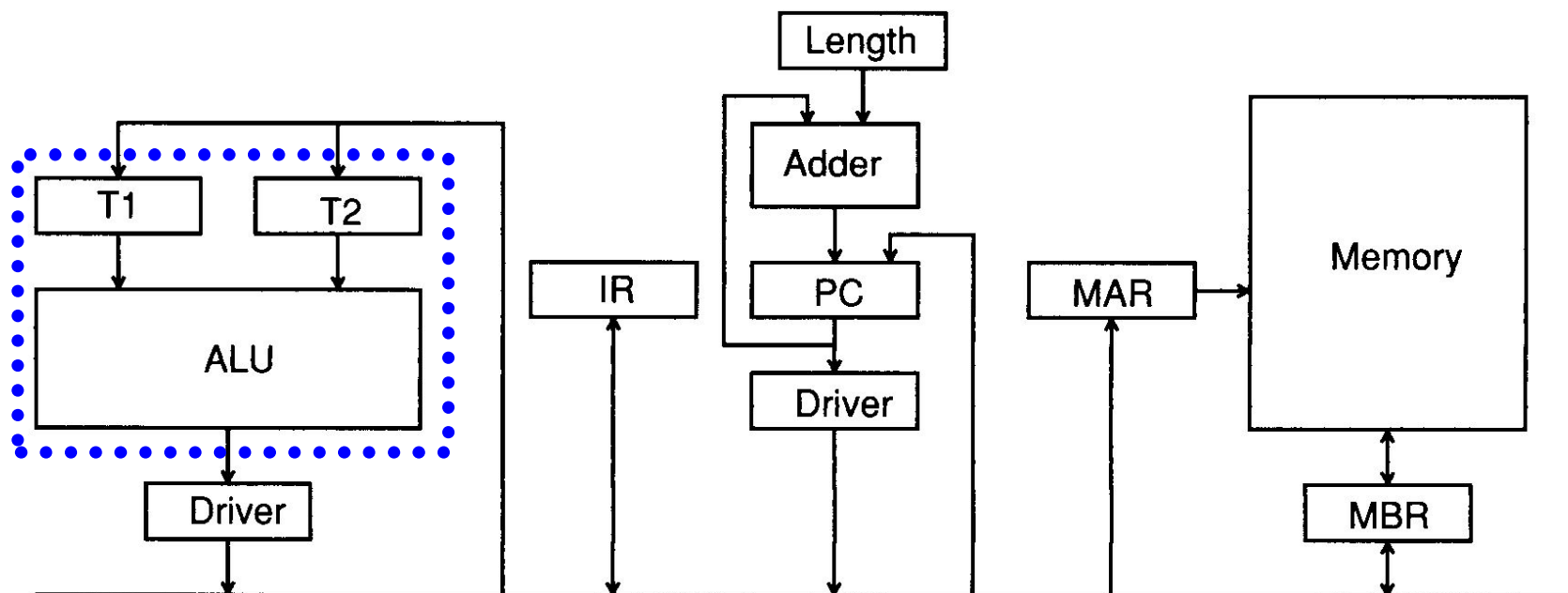


[8:3] bit -> 64 I/O eszköz

[2:0] bit: eszköz művelet

## b.) Kettő- és többcímű 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 utolsó operandus címe adja meg! [operátor],[operandus1],[operandus2]...



# Jelölés: kettő-, és többcímű gép

- Jelölés: **ADD2 X, Y** kétcímű 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: **ADD3 X,Y,Z** háromcímű 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 nélküiség*”) a T1, ill. T2 regiszter nem az utasítás-készlet architektúra része! (ezért nem keverendő össze a később említésre kerülő *regiszteres* címezéssel!)
  - Ebben az esetben csak az ALU részeként, nem pedig a rendszer gyorsítását szolgáló elkülönített regiszter bankként használjuk.



# Példa: Összeadás kétcímű géppel ADD2(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)

**Direkt  
címzést  
használunk  
itt!**

**Σ 250ns**

# Példa 2: Összeadás háromcímű géppel ADD3(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)

**Direkt  
címzést  
használunk  
itt!**

**Σ 295ns**

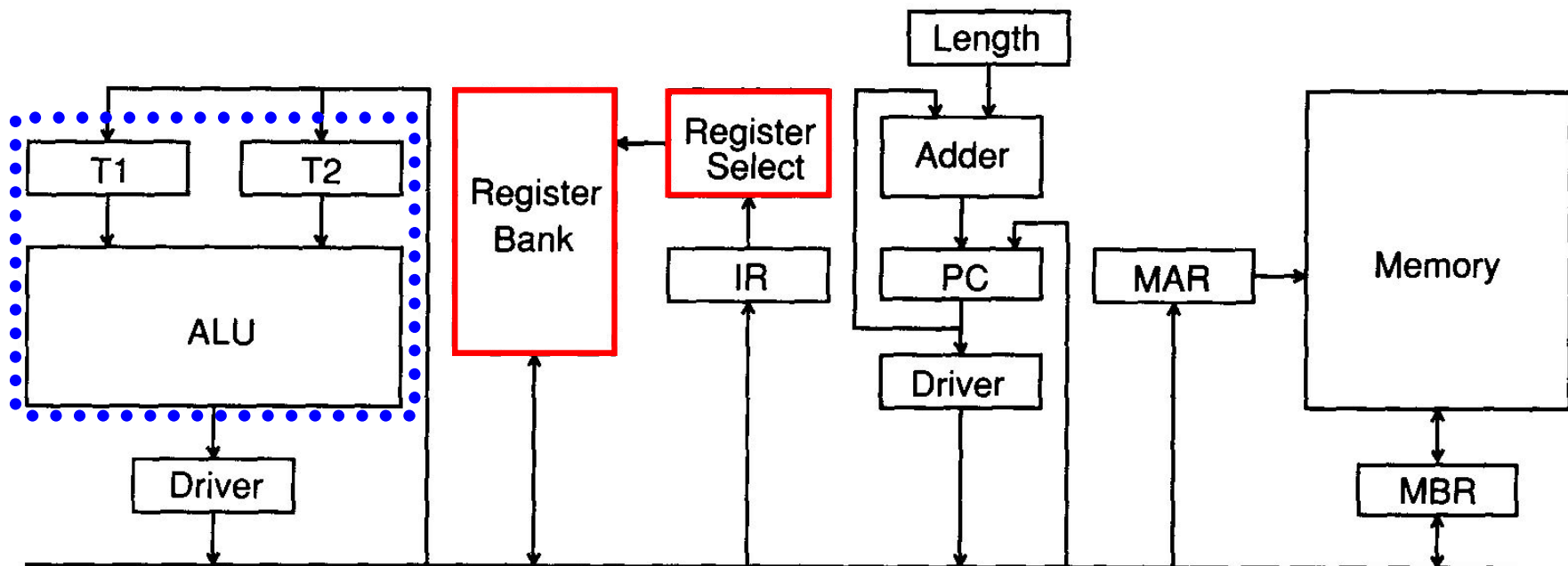
# Komplex műveletek: ADD3

- Az **ADD3** végrehajtásánál (ahogy az RTL leírásból is látszik) több időt vesz igénybe az utasítások F-D-E fázisa, mint az ADD2 esetén, mivel egyel több címre kell hivatkozni.
- Azonban, az **ADD3** jelentősége a komplexebb műveletek elvégzésekor mutatkozik meg: tömörebb forma, kevesebb utasítással
- Példa: Legyen  $X = Y * Z + W * V$  (oldjuk meg ADD2-vel és ADD3-al)

ADD2	ADD3
MOVE Y to X	AND3 Y,Z,T
AND2 Z,X	AND3 W,V,Y
MOVE W to Y	ADD3 T,Y,X
AND2 V,Y	
ADD2 Y,X	

## c.) Kettő- és többcímű 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)
- A regiszterbank használata csökkenti a végrehajtási időt: itt a lassú memória-intenzív műveletek helyett gyorsabb regiszterműveleteket használunk. (A regiszterbank  $2^N$  számú és szóhosszúságú ált. célú regisztert tartalmazhat.)



# Példa 1: Összeadás kétcímű géppel $\text{ADD2}(\text{R}_X, \text{R}_Y)$

Időszükségletek feltüntetésével!

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

*Fetch: (regiszterek feltöltése, utasításhívások):*

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

*Decode: (a dekódolást általában 0 idejűnek feltételezzük)*

*Execute: (végrehajtás)*

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

$\Sigma$  70ns

**Regiszteres  
direkt-  
címezést  
használunk  
itt!**

## Példa 2: Összeadás kétcímű géppel $\text{ADD3}(\text{R}_X, \text{R}_Y, \text{R}_Z)$

Időszükségletek feltüntetésével!

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

*Fetch: (regiszterek feltöltése, utasításhívások):*

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

*Decode: (a dekódolást általában 0 idejűnek feltételezzük)*

*Execute: (végrehajtás)*

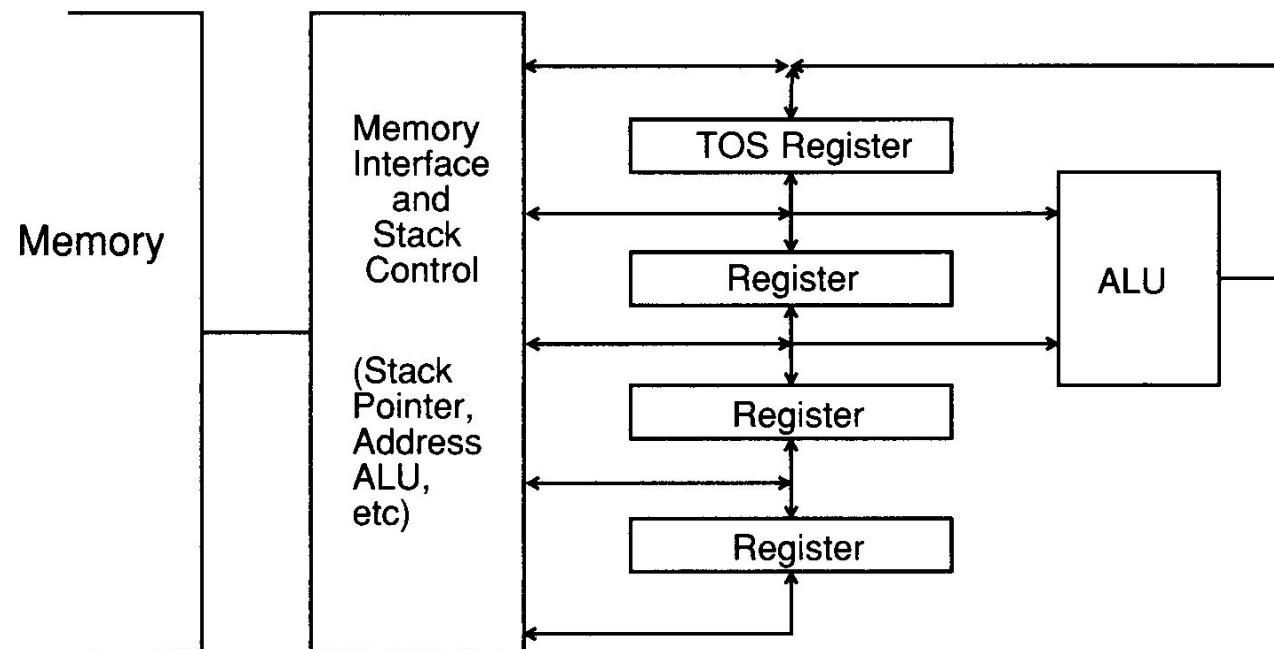
<b>RX</b> → <b>T1</b>	[5ns] <b>RX</b> értékét T1-be töltjük
<b>RY</b> → <b>T2</b>	[5ns] <b>RY</b> értékét T2-be töltjük
<b>T1 + T2</b> → <b>RZ</b>	[10+5ns] ADD2 művelet elvégzése, <b>RZ</b> -be töltjük

$\Sigma$  70ns

**Regiszteres  
direkt-  
címezést  
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.
- Különböző aritmetikai kifejezések hajthatók végre elég hatékonyan stack használatával: a szükséges operandusokat a stack egy-egy rekeszében tároljuk. A megfelelő operandusokat (felső kettő regiszterből) vesszük ki, elvégezzük rajtuk a műveleteket, és az eredményt a verem tetejére tesszük. Azért nevezzük **zéró címűnek**, mivel az operandusok azonosítására szolgáló utasításhoz nem használunk címeket. Az ábra a HW orientált stack rendszert ábrázolja.



# Példa: Zéró-vagy 0 című gép

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

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

- Az **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.

- A **2. módszernél** az aritmetikai kifejezést hátulról előre felé 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.

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 G	POP G





# Operandus címezési módok

# Operandus címezési módok



- Utasítás végrehajtásakor a kívánt operandust el szeretnénk érni, címével hivatkozhatunk a pontos helyére, azonosítjuk őt.
- Többféle címezési mód is létezik:
  - ☐ közvetlen (directed),
  - ☐ közvetett (indirected),
  - ☐ indexelt (indexed),
  - ☐ regiszteres megvalósítású (register relative).
- Ezek kombinációja igen sokféle, összesen akár 10-féle azonosítási módot tesz lehetővé.
- Jelölés: EA= Effektív (valódi) címe egy operandusnak

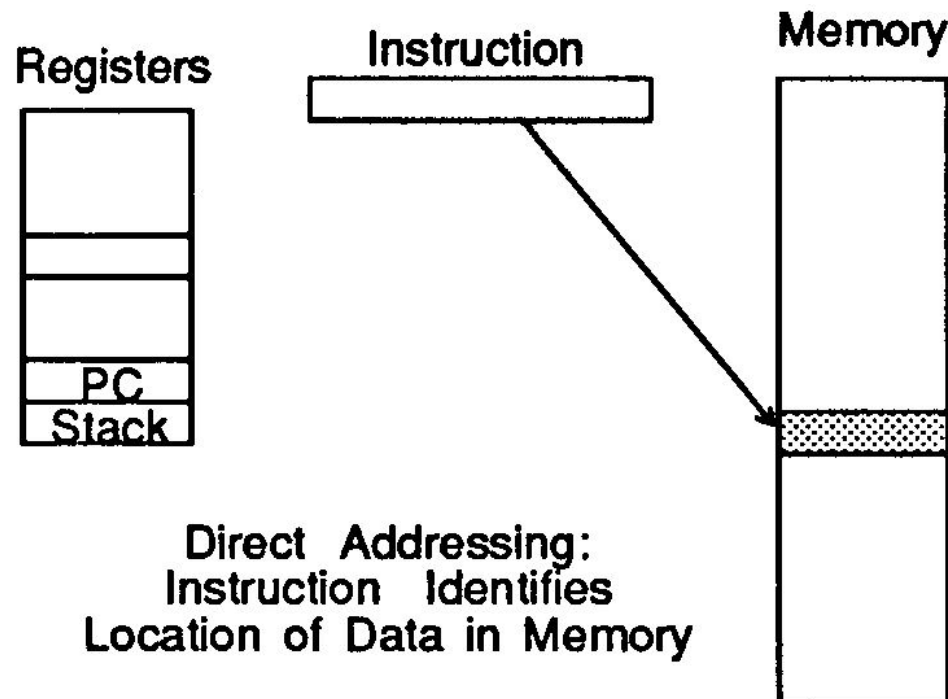
# 1. Direkt címezés (X)

- Az utasítás egyértelműen, közvetlenül azonosítja az operandus helyét a memóriában. (effektív cím EA= valódi címén tárolt érték) Jel: EA=A.
- **Jel: ADD2 X,Y** (X-ben tárolt op1 értéket hozzáadjuk az Y-ban tárolt op2 értékhez, az eredmény az Y-ban lesz.)

□ EA op1 = X

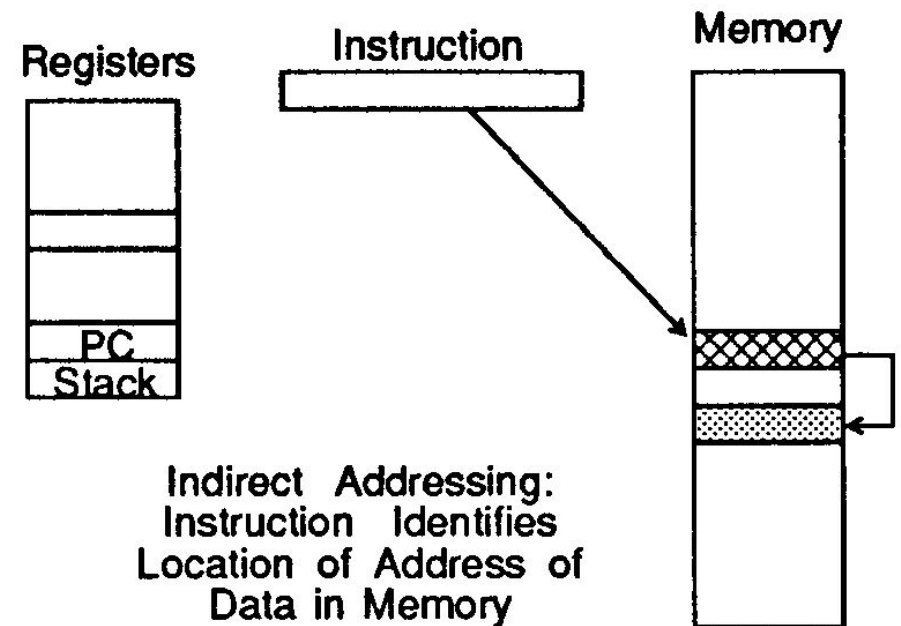
□ EA op2 = Y

Data:   
Address: 



## 2. Indirekt címezés (\*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: **ADD2 \*X,\*Y** (\*: indirekció)
  - EA op1 = MEM[X]
  - EA op2 = MEM[Y]
- Ezt különböző gyártók többféleképpen jelölik. Általában az indirekt címezési módot (\*)-al jelölik:
- Példa: **ADD2 \*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 címezés a *memóriatömb* elemeinek elérhetőségét biztosítja!



Direkt-indirekt kombináció is lehetséges:

PI: ADD2 X,\*Y

# Példa: Összeadás kétcímű géppel ADD2(\*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

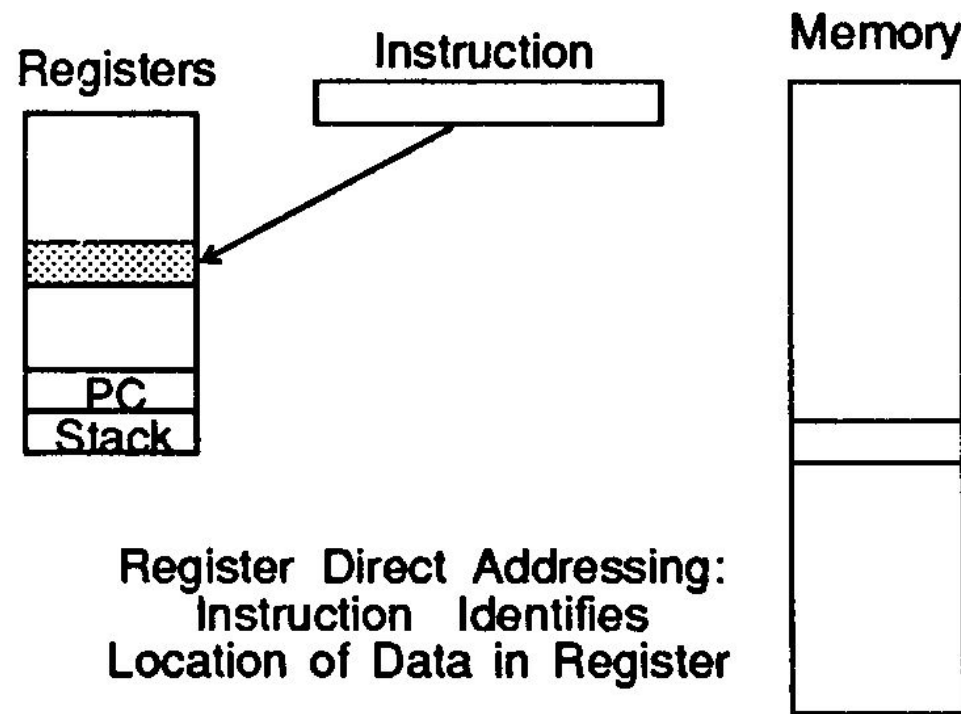
T1 + T2→MBR	[10+5ns] ADD2 művelet elvégzése, MBR-be töltjük
MBR→M[MAR]	[30ns] Eredményt a memóriában tároljuk el (ahol Y volt)

**Indirekt  
címzést  
használunk  
itt!**

**Σ 320ns**

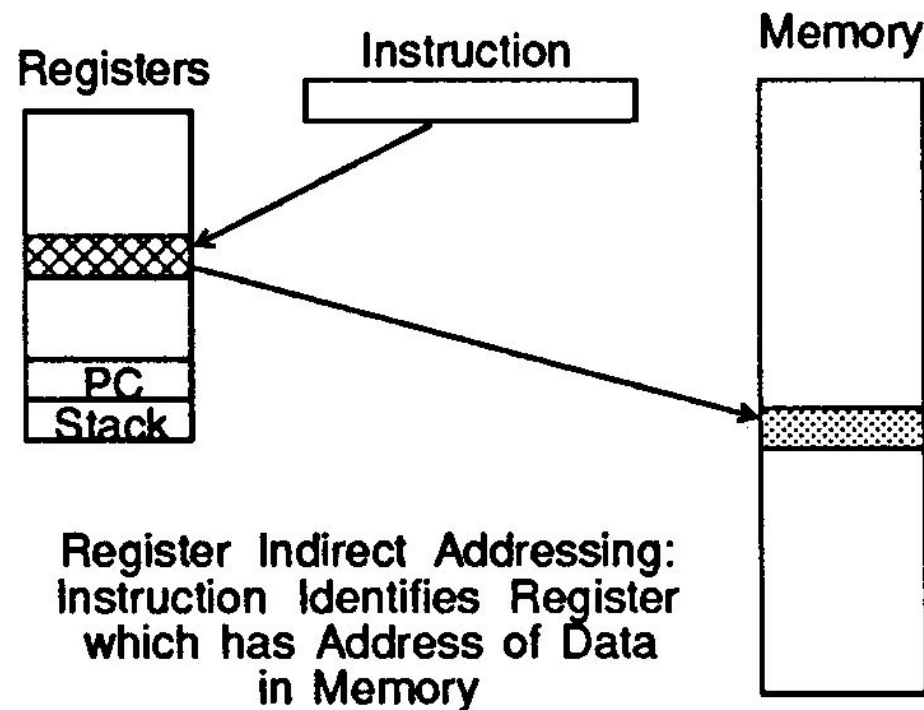
### 3. Regiszteres direkt címezés ( $R_x$ )

- Hasonló, mint a direkt címezés, 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.



## 4. Regiszteres indirekt címzés (\*R<sub>x</sub>)

- 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.



# Példa: Összeadás kétcímű géppel $\text{ADD2}(*R_X, *R_Y)$

Időszükségletek feltüntetésével!

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

*Fetch: (regiszterek feltöltése, utasításhívások):*

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

*Decode: (a dekódolást általában 0 idejűnek feltételezzük)*

Regiszteres  
**indirekt-**  
címzést

*Execute: (végrehajtás)*

használunk itt!

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

---

$\Sigma$  170ns



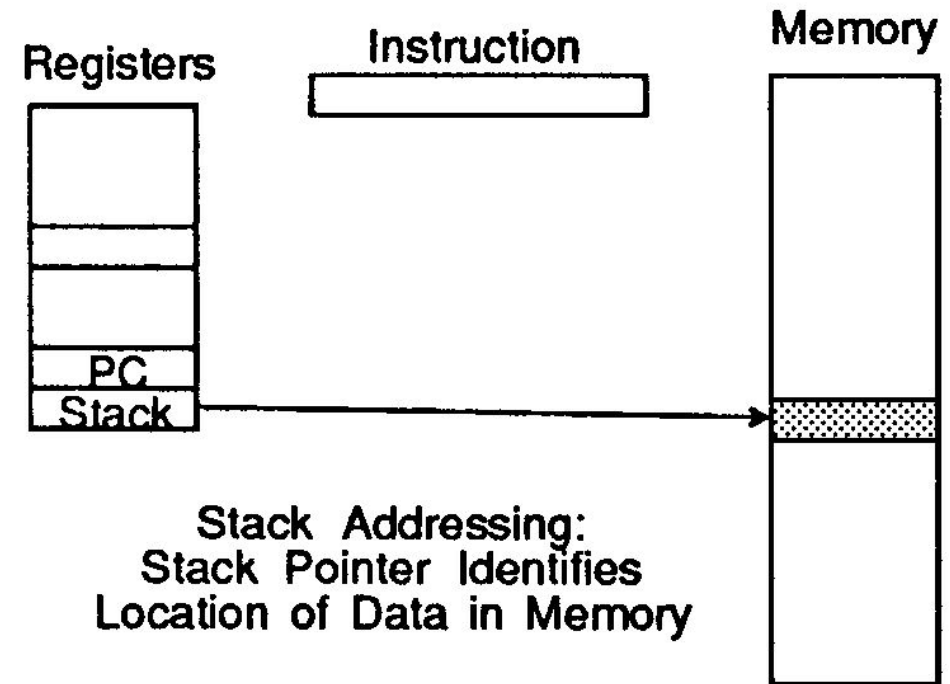
# Összehasonlító táblázat – I.

- 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

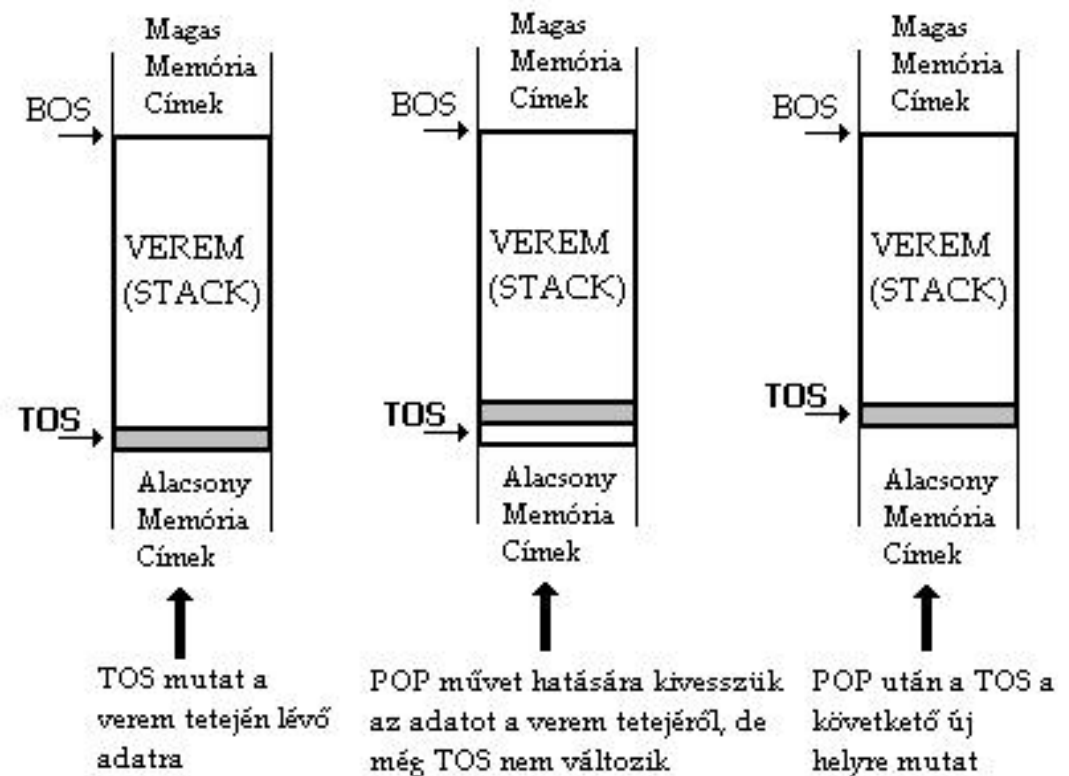
- Verem (STACK) címzés, vagy regiszteres indirekt autoincrement címzési mód: *indirekt* módszerrel az operandus memóriában elfoglalt helyét a címével azonosítjuk, és akár az összes memóriatömbben lévő elem megcímezhető. *Autoincrement* is, mivel a címeket automatikusan növeli. Ezt (+) jellel jelöljük: **\*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

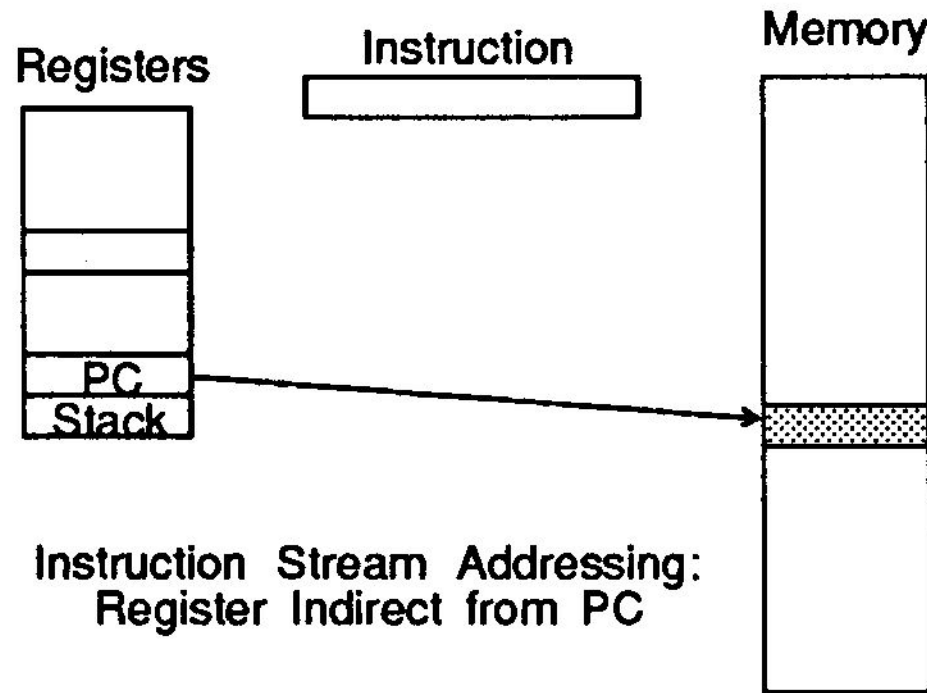
- **POP művelet** kivesszük a stack tetején lévő adatot (TOS), és egy Rx regiszterbe rakjuk. Ezután a stack pointer automatikusan inkrementálja a címet, amivel a következő elemet azonosítja a verem tetején.  
Jel: **MOVE \*R (stack pointer)+, Rx**
- **PUSH művelet:** a stack pointer automatikusan dekrementálja a címet, amivel a verem tetején lévő elemet azonosítja. Majd ezután berakjuk az Rx regiszterben lévő elemet a stack tetejére, a pointer átlát mutatott címre.  
Jel: **MOVE Rx, \*R (stack pointer)–**

Példa: POP műveletre



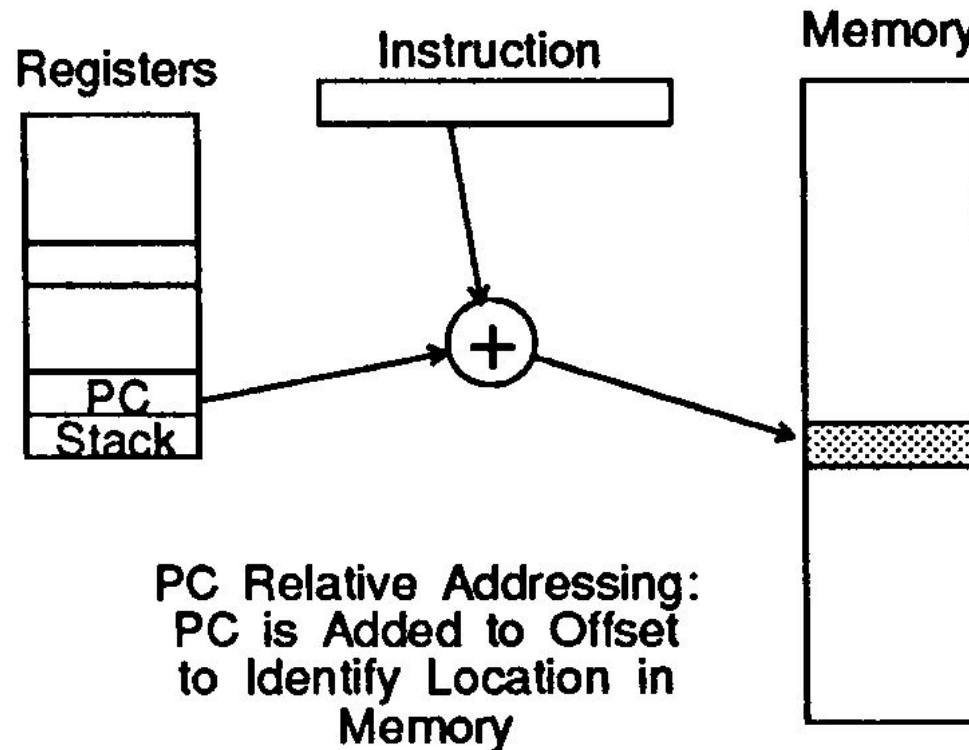
## 6. Instruction Stream címzés

- A PC közvetlenül azonosítja a memóriában lévő adatot és címet. Az utasítás végrehajtásakor visszkapjuk 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 utasításfolyamban.



# 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

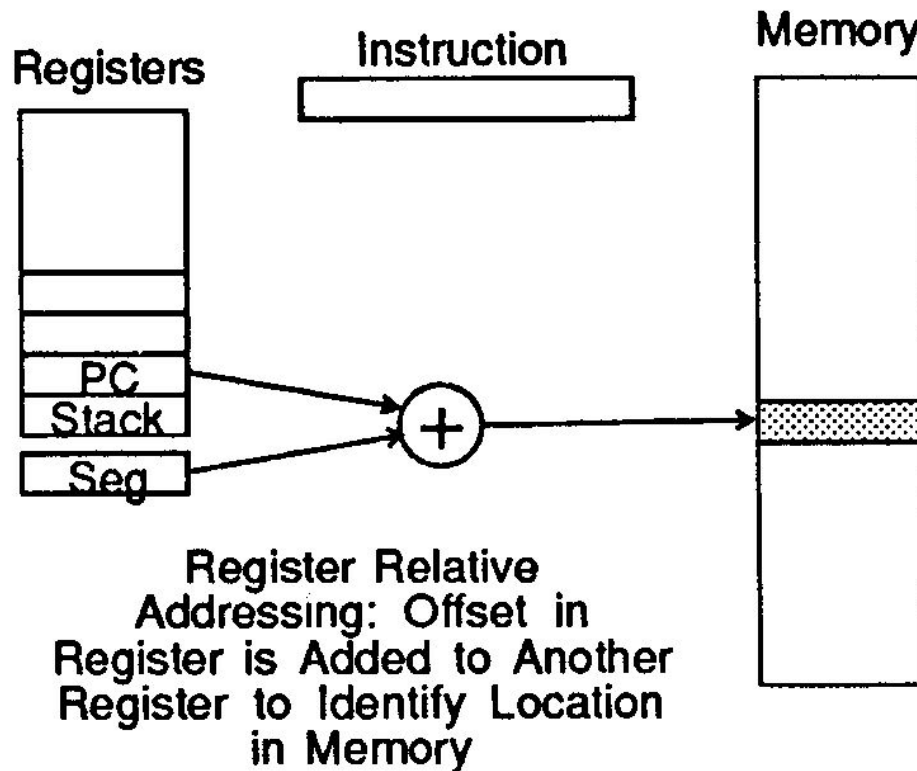


# 8. Regiszter relatív címzés

- A PC-regiszter eltolási értékéhez hozzáadódik *egy, vagy akár 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

PI: Intel 80x86  
segments:

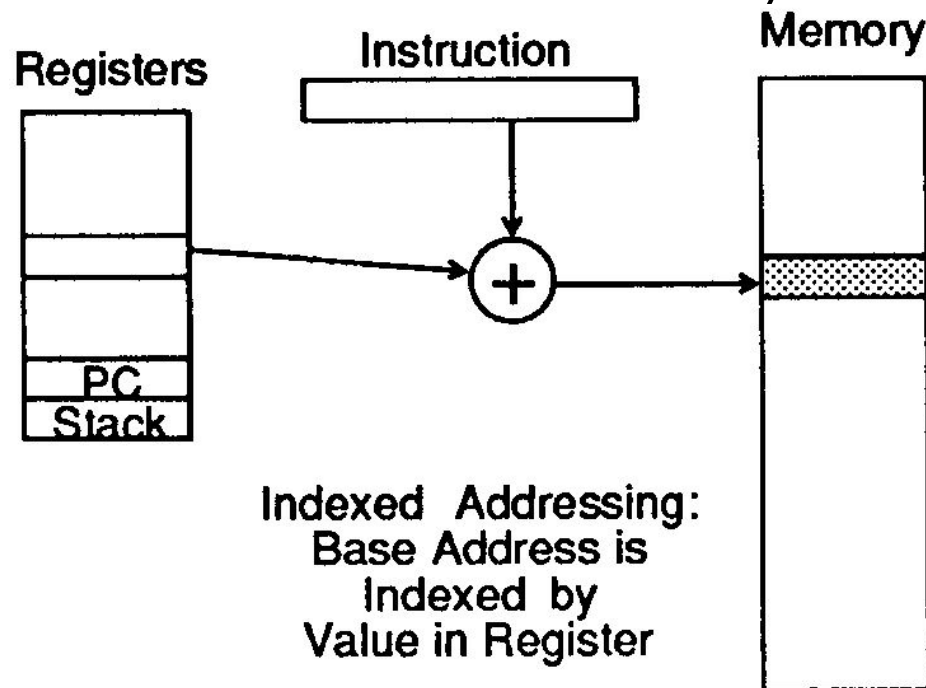
ds: data  
cs: code  
ss: stack  
es: extra



# 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



```
int main(void){
    int i;
    ptr = &my_array[0];    /* point our pointer to the first
                             element of the 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;
}
```

# Összehasonlító táblázat – II.

- Címzési módok és jelöléseik összefoglaló táblázata

**Table 4.1.** Addressing Modes and their Nomenclatures.

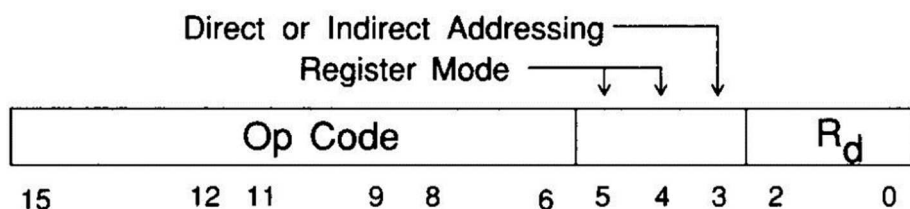
<i>Addressing Mode</i>	<i>Represented By</i>	<i>Comment</i>
Direct	@<address>	Address is part of instruction.
Register Direct	Rn	Operand is found in register.
Indirect	*(<address>)	Address is part of instruction; operand is located in memory at that address.
Register Indirect	*Rn	Address found in register; operand in memory at that address.
Instruction Stream	#<value>	Value is stored in instruction stream.
Register Indirect Autoincrement	*Rn+	Register used as address; value in register incremented at end of instruction.
Stack Addressing	Push Pop	Stack pointer identifies location in main store for transfers; value in stack pointer adjusted as necessary.
PC Relative	\$<offset>	Offset identifies target address relative to current location identified by program counter.
Memory-Based Index	(<address> i Rm)	Operand is located in memory at address which is sum of <address> and Rm.
Register-Based Index	(Rn i Rm)	Operand is located in memory at address which is sum of Rn and Rm.



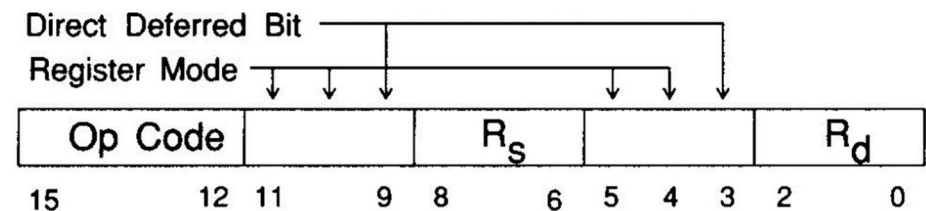
# Példa 1: DEC PDP11 működése

- Különböző címzési módokat használt egyszerre
- *16-bites* gép: utasítás opcode-ja + további infók (cím)
  - 3 biten: 8 általános célú regiszter ([2:0]) – **R<sub>d</sub>**-nek lefoglalt rész
  - További 3 biten: **R<sub>source</sub>** (regiszter specifikáció használatához)
  - !Dupla operandus esetén: csak 4 bit marad az utasítások kódolására (**opcode**)
  - Egyszeres operandus esetén: 10 bites **opcode**
- Egyszeres- és dupla operandusú utasítás formátum

Single Operand Instruction Format



Double Operand Instruction Format



# PDP11 utasítás-kódolása

**Table 4.2.** Encoding of Instructions for the PDP 11 Architecture.

<i>Op Code</i>	<i>Function Performed</i>
0 0 0 0	Single address and special function instructions
0 0 0 1	Move instruction
0 0 1 0	Compare instruction
0 0 1 1	Bit test instruction
0 1 0 0	Bit clear instruction
0 1 0 1	Bit set instruction
0 1 1 0	ADD2 instruction
0 1 1 1	Single address instructions
1 0 0 0	Single address and special function instructions
1 0 0 1	Move instruction (byte)
1 0 1 0	Compare instruction (byte)
1 0 1 1	Bit test instruction (byte)
1 1 0 0	Bit clear instruction (byte)
1 1 0 1	Bit set instruction (byte)
1 1 1 0	Subtract instruction
1 1 1 1	Special purpose instructions

Dupla operandusú címzés esetén a maradék felső 4-biten, [15...12]-ig azonosítja az utasításokat (itt összesen 16 lehet).

# PDP11 címzésének kódolása

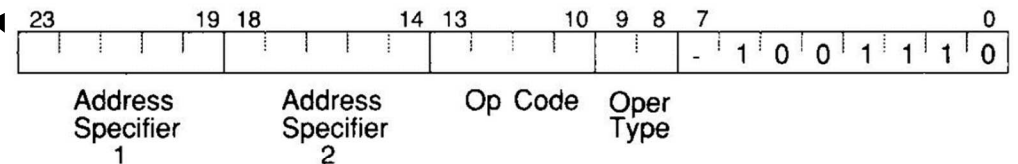
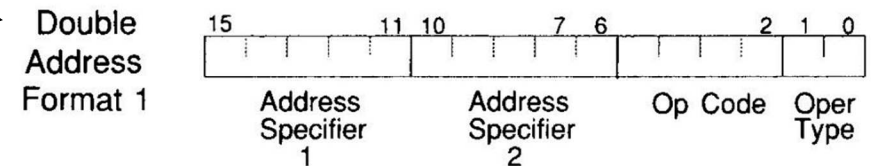
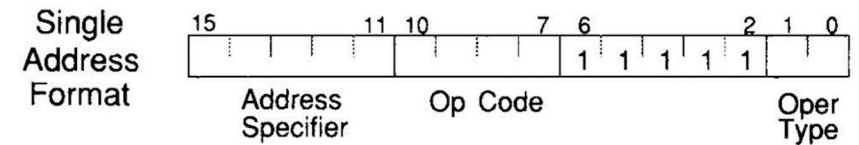
**Table 4.3.** Encoding of Addressing Information  
in the PDP 11 Architecture.

<i>Addressing modes for PDP 11 operands</i>		
<i>Addr bits</i>	<i>Addressing mode</i>	
0	0 0	Register direct
0	0 1	Register indirect
0	1 0	Register indirect — autoincrement
0	1 1	Two level indirect, autoincrement register
1	0 0	Register indirect — autodecrement
1	0 1	Two level indirect, autodecrement register
1	1 0	Indexed
1	1 1	Indexed indirect
11.	9. bit	

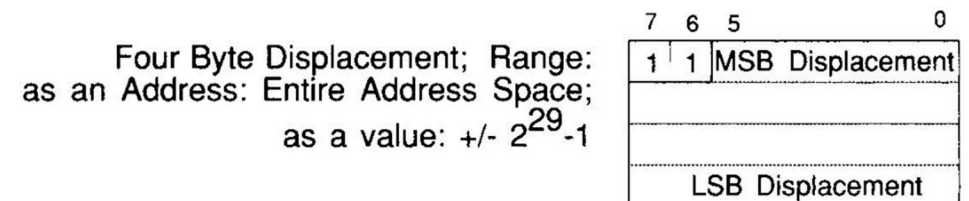
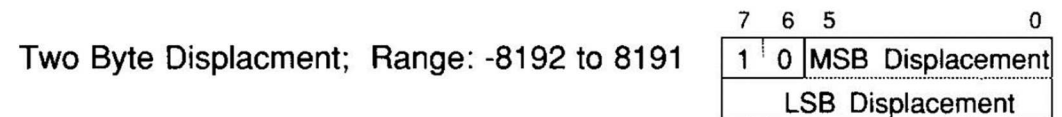
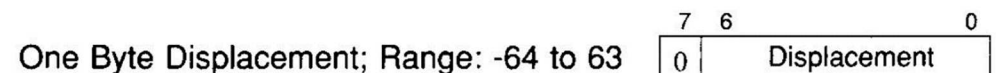
<i>Addressing modes when PC is target register</i>		
<i>Addr bits</i>	<i>Addressing mode</i>	
0	1 0	Immediate mode (Instruction stream)
0	1 1	PC absolute mode
1	1 0	PC relative
1	1 1	PC relative, indirect
2.	0. bit	

# Példa 2. NS32032 – bővíthető műveleti kód

- National Semiconductor – 32 bites processzora
- Egy- (JUMP, JSR) és két-című utasítások dekódolása
  - Format 1: általános használatra (Add, Sub, Comp, Mov. Stb)
  - Format 2: bővített (Div, Test, Shift, Abs)
- Több címezési mód
- Addr. Specifier(ek): 5 biten (32 lehetséges címkombináció)
- Addr. Displacement (eltolás): 1, 2, 4 byte-on.



Double Address Format 2





# Programszervező utasítások

# Programszervező utasítások

- Program végrehajtásának szabályozása:
  - Feltételes, feltétel nélküli utasítások (IF  
BRANCH)
  - Szubrutin (eljárás) hívás (CALL) / visszatérés  
(RETURN)
  - Ciklus (iteratív végrehatás)
  - Ugró utasítások (JUMP)

# JUMP utasítás RTL leírása (pl. PDP 11 gépen)

*Fetch: (regiszterek feltöltése, utasításhívások):*

**PC→MAR**

Elsőként a PC-ből az utasítás címe (tartalmazza a cél címét is) a MAR-ba töltődik

**M[MAR]→MBR**

Memóriában lévő utasítás beírása az MBR-be

**PC+2→PC**

Az utasítás hosszával növeli a PC értékét

**MBR→IR**

Majd az MBR-ben lévő adatot az IR-be tesszük

*Decode:*

*Execute: (végrehajtás)*

**PC→MAR**

PC-vel a cél (ugrás akt.+1) címét szeretnénk kiolvasni

**M[MAR]→MBR**

Ezt az ugrási címet az MBR-be tesszük, majd ezzel

**MBR→PC**

Aktualizáljuk a PC értékét (program számláló adott pontra fog mutatni)

# IF feltételes elágazás RTL leírása (pl. PDP 11 gépen)

*Fetch: (regiszterek feltöltése, utasításhívások):*

**PC→MAR**

Elsőként a PC-ből az utasítás címe (tartalmazza a cél címét is) a MAR-ba töltődik

**M[MAR]→MBR**

Memóriában lévő utasítás beírása az MBR-be

**PC+2→PC**

2-vel növeli a PC értékét

**MBR→IR**

Majd az MBR-ben lévő adatot az IR-be tesszük

*Decode:*

*Execute: (végrehajtás)*

**IF (carry == 1)**

feltételes elágazás (itt a carry bit/flag tartalmától függően)

**{**

Feltétel ellenőrzés, ha Igaz akkor...

**PC+(2xIR<7:0>)→PC**

PC-vel a következő utasítás (cél) címére mutatunk (PC+offset)

**}**

IR<7:0> : 8 bites offset cím (előjel kiterjesztés 16-bitre ,páros szóhatárra)

**ELSE**

**{**

**PC+2→PC**

Ha a feltétel Hamis, megváltoztatjuk PC címét

**}**



# Call (szubrutin hívás) RTL leírása (PDP 8 gépen)

*Fetch: (regiszterek feltöltése, utasításhívások):*

<b>PC→MAR</b>	Elsőként a PC-ből az utasítás címe (tartalmazza a cél címét is) a MAR-ba töltődik
<b>PC+1→PC</b>	Az utasítás hosszával növeli a PC értékét
<b>M[MAR]→MBR</b>	Memóriában lévő utasítás beírása az MBR-be
<b>MBR→IR</b>	Majd az MBR-ben lévő adatot az IR-be tesszük

*Decode:*

*Execute: (végrehajtás)*

<b>IR→MAR</b>	Feltételezzük, hogy utasítás már tartalmazza a címet
<b>PC→ M[MAR]</b>	A visszatérési címet a szubrutin elejére fogja tenni (indirekt ugrás)
<b>IR→PC</b> fogja	Aktualizáljuk a PC értékét (az előző a visszatérési címet tartalmazni), majd
<b>PC+1→PC</b>	A szubrutin első utasítására fogunk mutatni PC-vel ( <i>hívás</i> )

„Subroutine linkage”: biztosítja hogy az alprogram végrehajtása után a hívó (call) eljáráshoz vissza tudjunk térni (return)

# Return (visszatérés) RTL leírása (PDP 8 gépen)

*Fetch: (regiszterek feltöltése, utasításhívások):*

**PC→MAR**

Elsőként a PC-ből az utasítás címe (tartalmazza a cél címét is) a MAR-ba töltődik

**PC+1→PC**

Az utasítás hosszával növeli a PC (nem aktuálisan használt) értékét

**M[MAR]→MBR**

Memóriában lévő utasítás (mint ugrás) beírása az MBR-be

**MBR→IR**

Majd az MBR-ben lévő adatot az IR-be tesszük

*Decode:*

*Execute: (végrehajtás)*

**IR→MAR**

Feltételezzük, hogy utasítás már tartalmazza a címet (szubrutin elejét azonosítja ez a cím)

**M[MAR]→MAR**

A **visszatérési címet** kinyerjük és a MAR-ba tesszük (cím a szubrutin elején található), majd ezzel

**M[MAR]→PC**

Aktualizáljuk a PC értékét (program számláló a visszatérési címre fog mutatni, amely után a főprogram folytatja feladatát, a szubrutinból való visszatérés után) - *visszatérés*

„Subroutine linkage”: biztosítja hogy az alprogram végrehajtása után a hívó (call) eljáráshoz vissza tudjunk térni (return)

# PDP-8 Call, Return utasításai


## ■ Problémák:

- Memóriát használ (lassú)
- Egy időben csak egy hívó rutin „hívhatja” a szubrutint (visszatérési cím mindig a memória egy adott részén helyezkedik el) → nem biztosít „Time Sharing” üzemmódot
- Nem lehet rekurzió (egymásba ágyazott hívás)
- ROM-ot alkalmazó rendszerekben nem működik

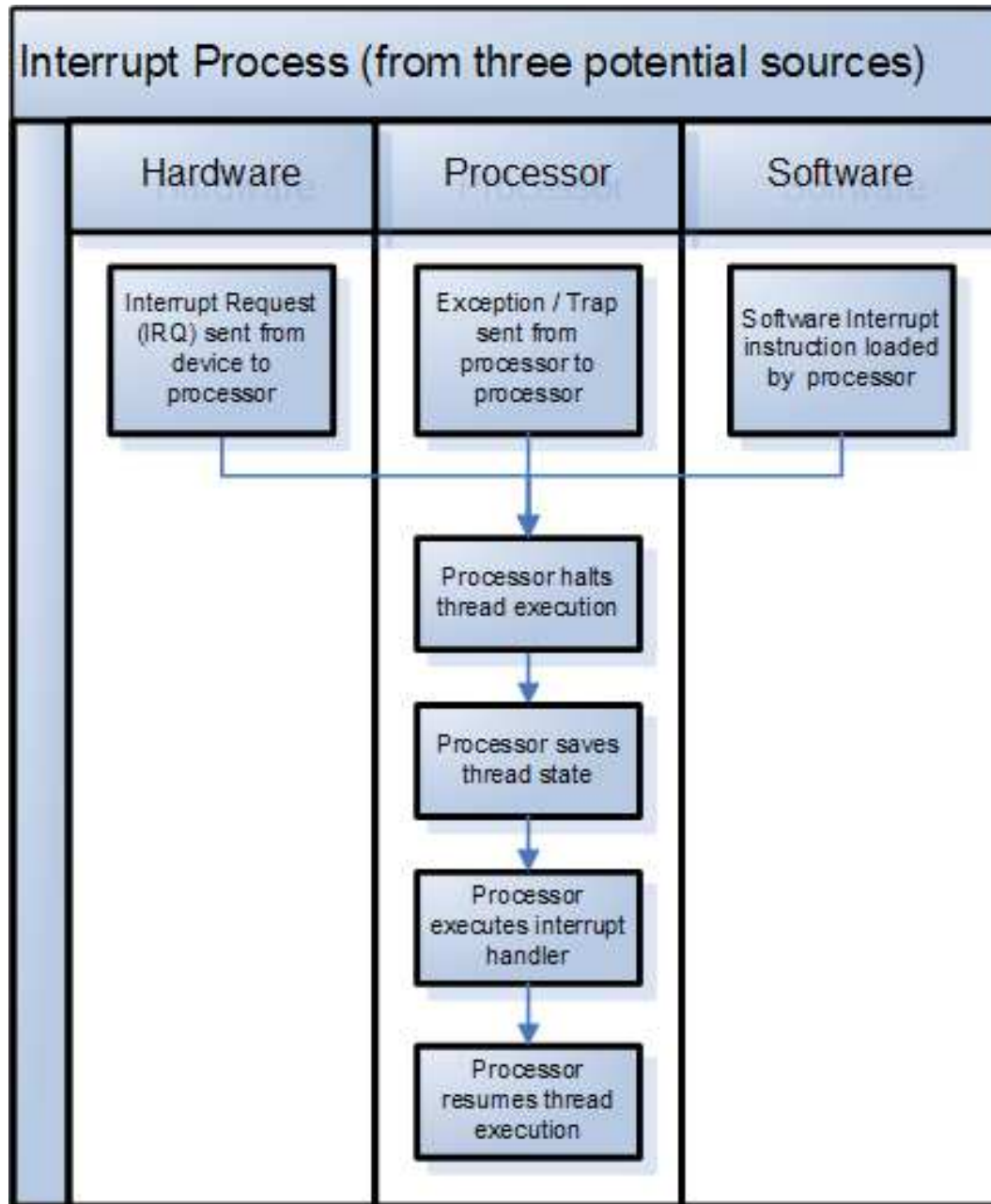
## ■ **Megoldás:** visszatérési cím tárolása általános célú regiszterben, Stack használatával (SP)

- Példa: Motorola 68000 **JSR**, **RTS** műveletekkel

# További programvezérlési (program-control) módszerek

- 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



# RISC és CISC processzorok utasításkészletei

# Utasítás készletek

- Fontos paraméter: utasítások száma
- Kezdetben egyszerű felépítésű gépek
- Egyszerű utasítások és gépi nyelv
- Azonban a komplex problémákat kívántak megoldani (magasabb szintű leírással) → „**Szemantikus rész**”
- Megoldás: compiler
- ISA (Instruction Set Architecture): CISC, RISC 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 címezési 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):

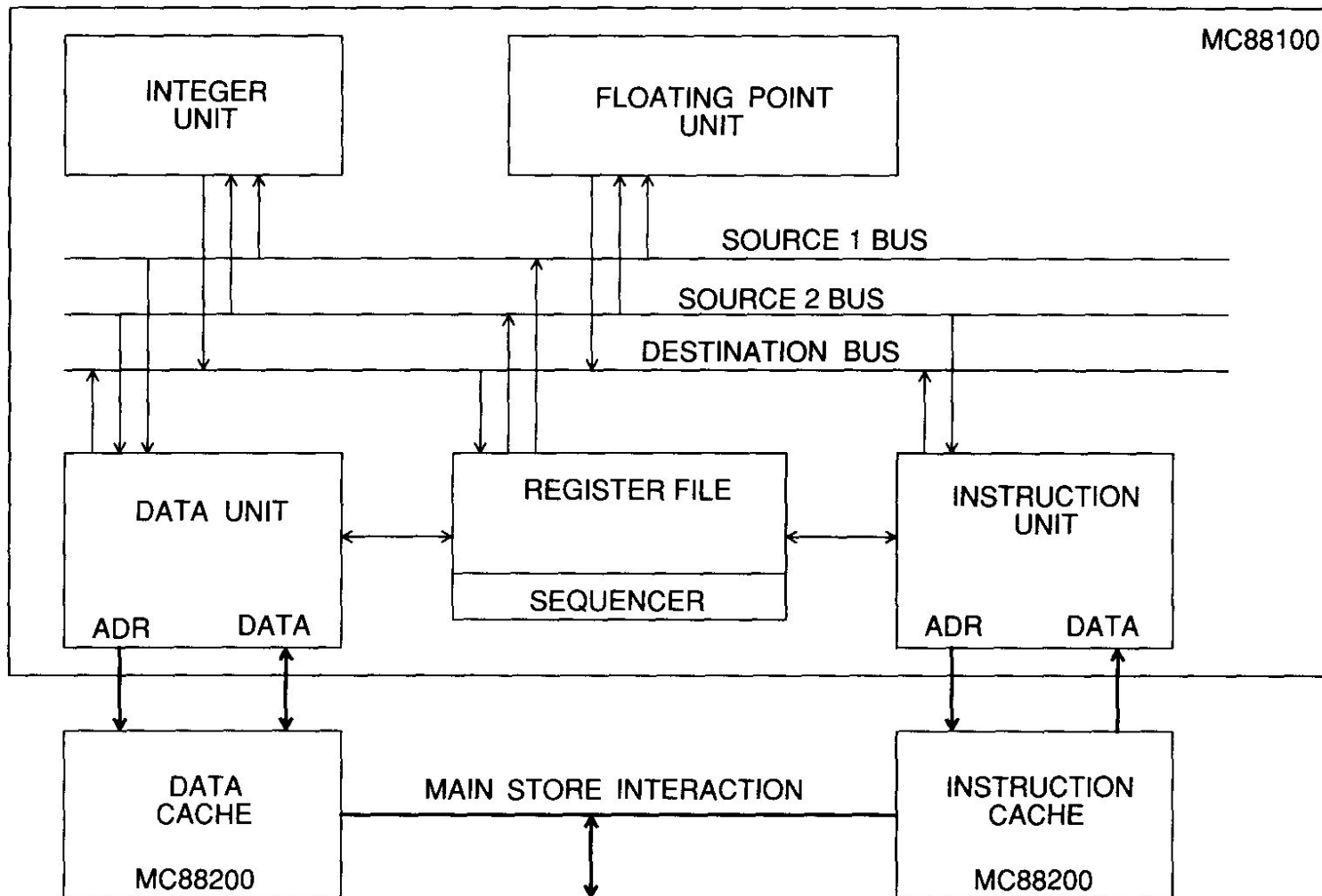
- *Huzalozott (hardwired) utasításdekódolás* (a hardveres dekódolás megvalósításához kombinációs logikát használ, azonban a mai memóriaalapú mikro-kódú gépeknél ez lassabb).
- *Egyszeres ciklusvégrehajtás*: (minden egyes ciklusban egy utasítást hajt végre, ha ezt sikerülne elérni optimális lenne az erőforrás kihasználás - VLSI technológiától függő. 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: **utasítás 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. Sourcel-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



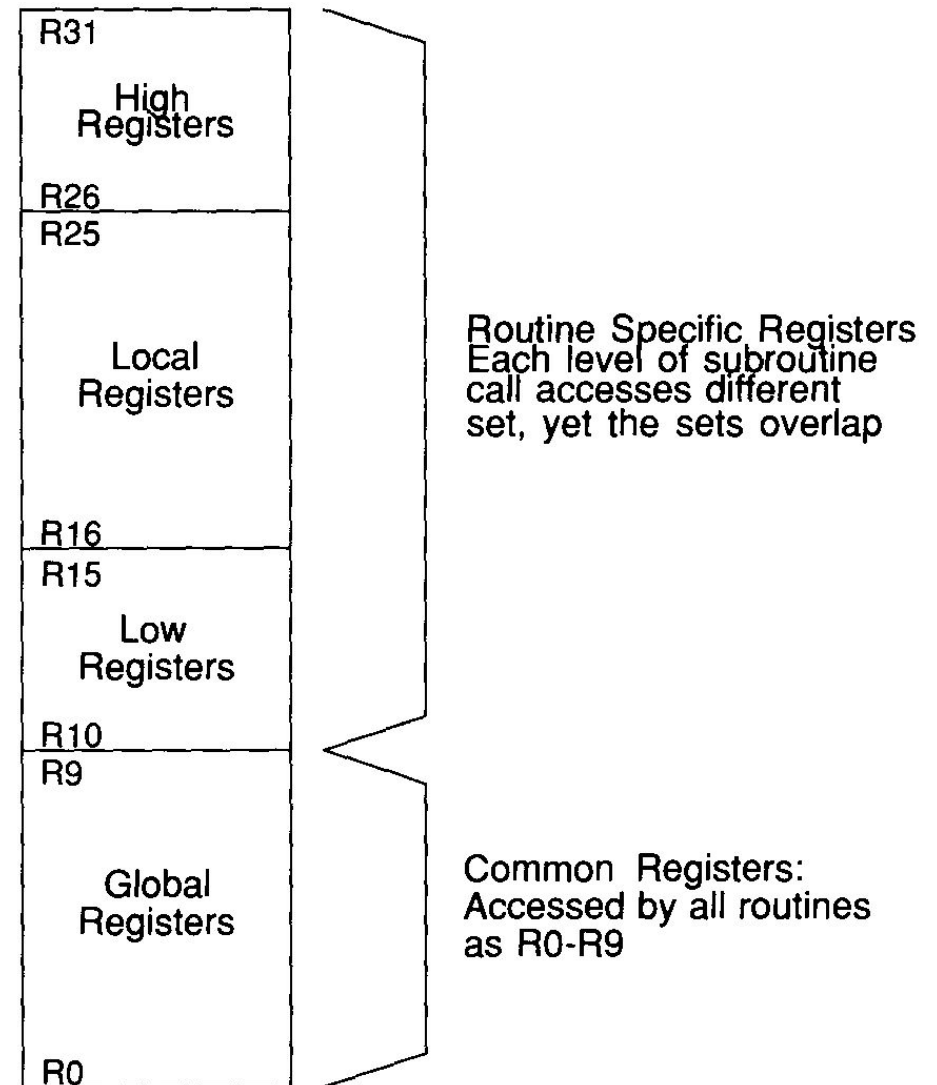
Hardveres  
párhuzamosítás:

- Buszok
- Cache
- Data /  
Instruction Unit  
(Harvard Arch!)

# Berkeley RISC-I rendszere:

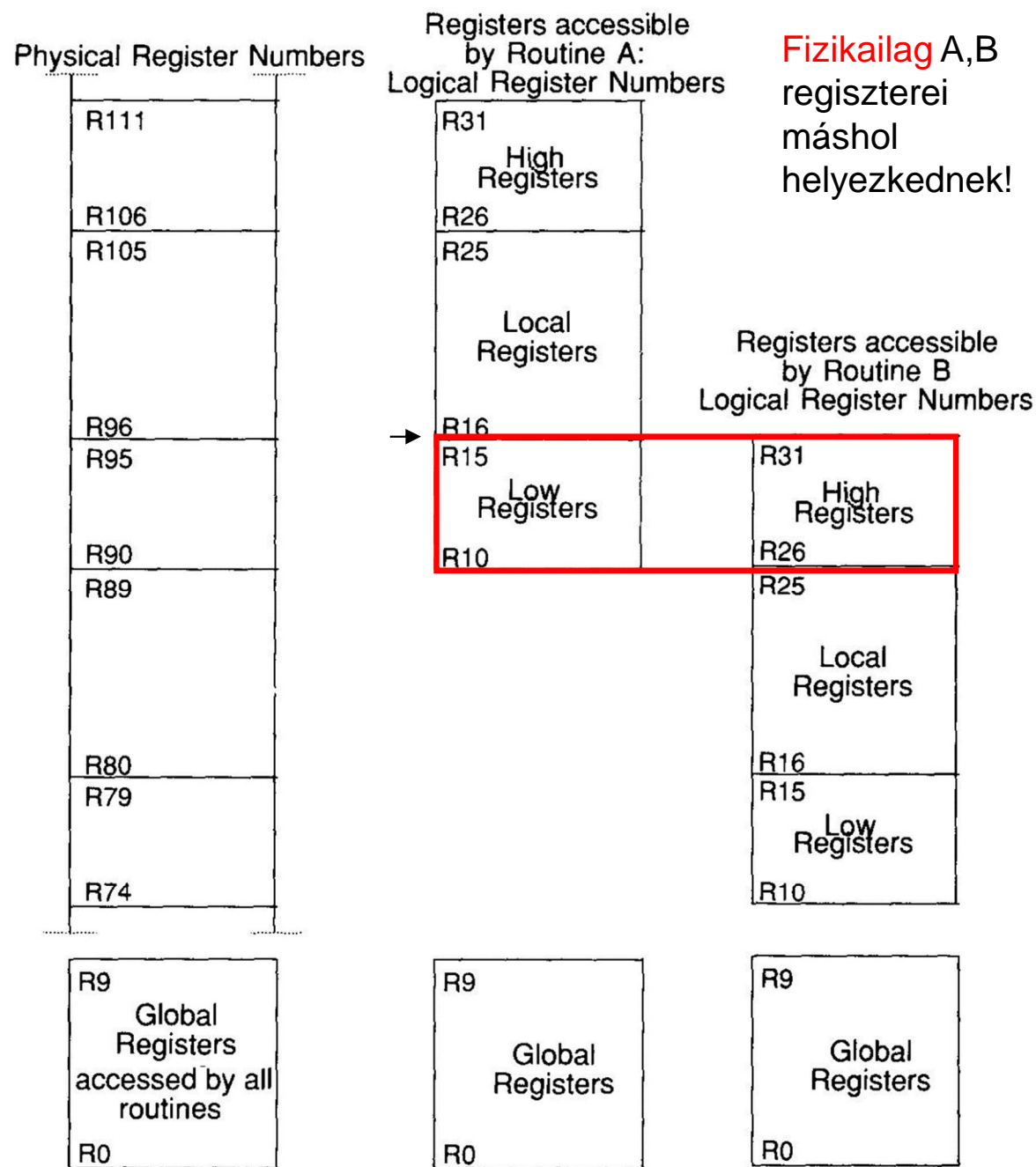
## Regiszter használat

- **Regiszter ablak:** a szubrutin híváshoz (call) / visszatéréshez (ret) szükséges processzor-időt kívánták minimalizálni nagy számú regiszter stack használatával.
- Regisztereknek 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 ablakon belül, 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.



# CISC Processzorok jellemzői

- **CISC:** Complex Instruction Set Computer
- Nagyszámú utasítás-típust, és címezési 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 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 (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)

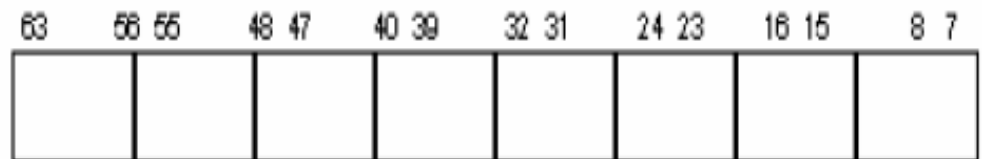
- Közvetlen memória-elérés (DMA) és összetett/bonyolult műveletek jellemzők rá.
- Mikro-programozott vezérlési mód
  - a CISC processzor esetén a fordító (compiler) a programot egyszerűbb szintre fordítja, majd ezután a mikroprogram (ami meglehetősen összetett lehet) veszi át a vezérlést – mikroutasítások 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



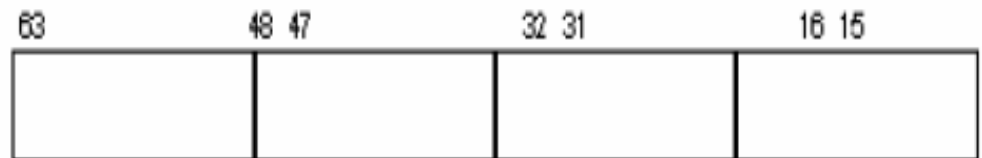
# PI. MMX kiterjesztés

- 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.

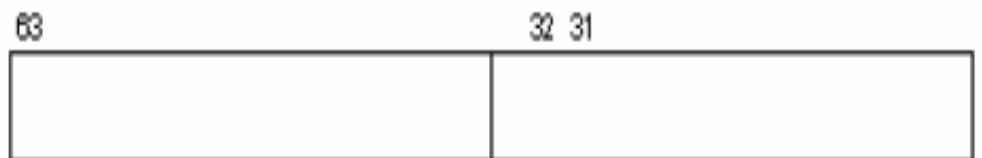
Packed byte (eight 8-bit elements)



Packed word (four 16-bit elements)



Packed doubleword (two 32-bit elements)



Quadword (64-bit element)





# PI. SSE, SSE2 kiterjesztés

Eredeti nevén **KNI**: *Katmai New Instructions* (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