

Pannon Egyetem

Villamosmérnöki és Információs Tanszék



Számítógép Architektúrák II.

(MIVIB344ZV)

6. előadás: Utasítás végrehajtás folyamata,
címezési 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 → Számítógép Architektúrák II.

- ([chapter04.pdf](#))

- Fóliák, óravázlatok .ppt (.pdf)

- Feltöltésük folyamatosan



Utasítás végrehajtás folyamata

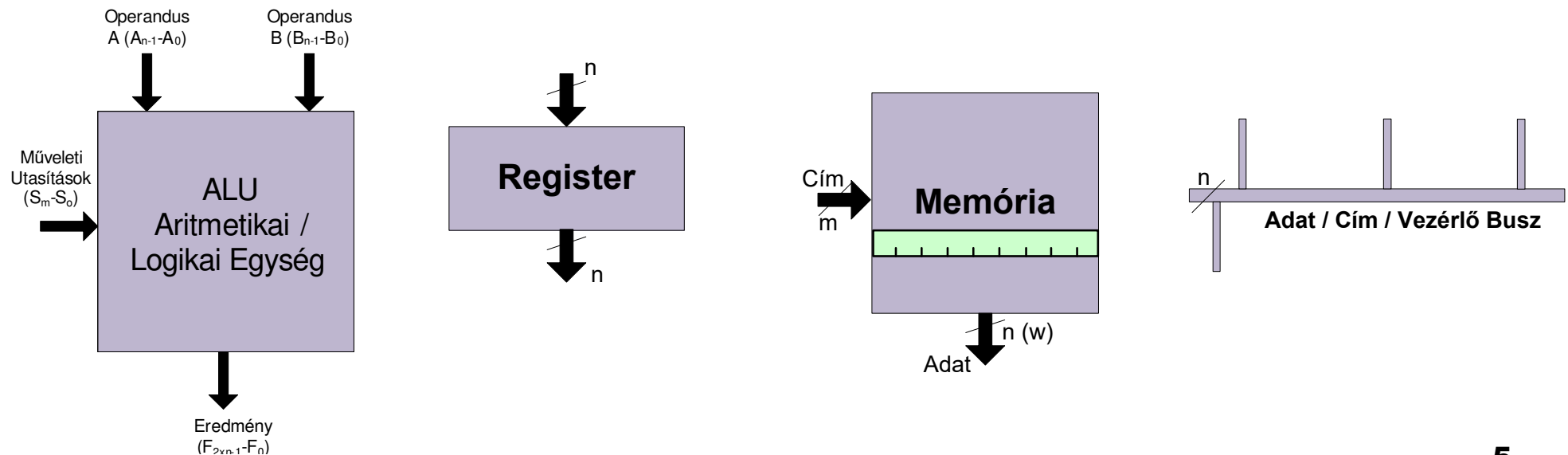
- Utasítás kódok
 - Programvezérlő utasítások
- Címzési módok
- RISC vs. CISC processzor architektúrák



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 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

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 teszi a tárolt adatokat, vagy például egy vezérlőjel hatására, *lépteti (shift-eli)* a benne lévő adatokat.

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 $n\text{-bit}$ (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.

További digitális építőelemek

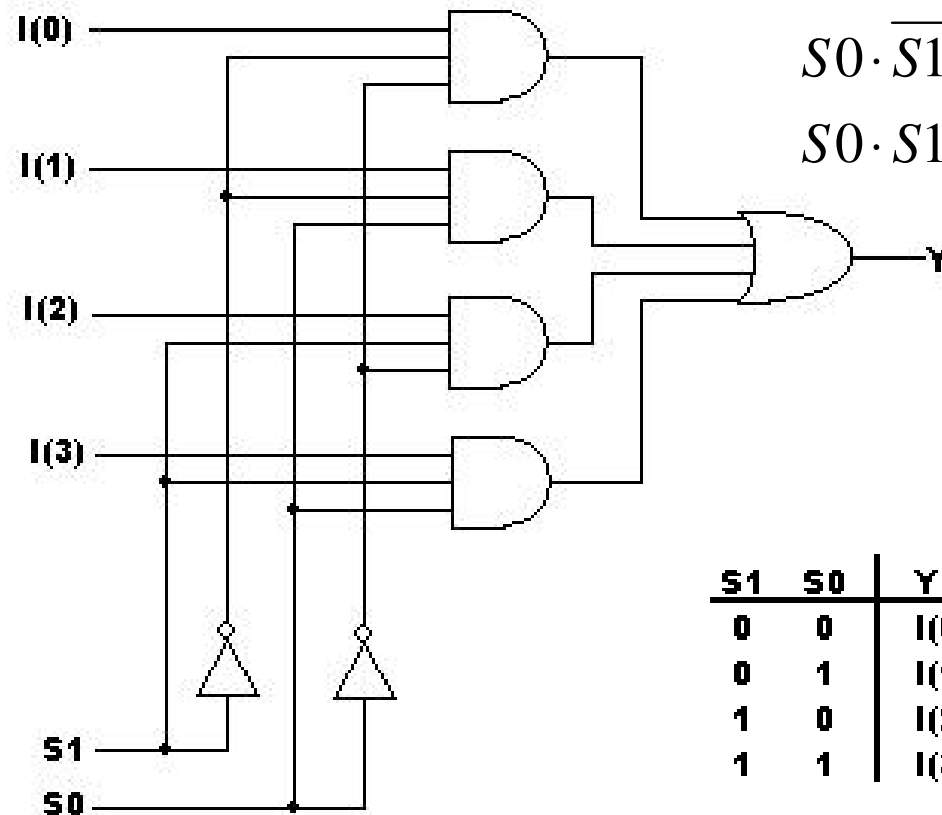
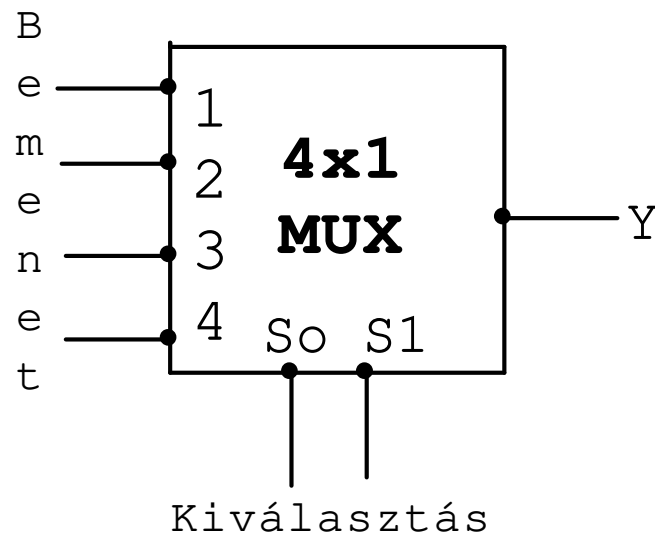
- a.) MUX
- b.) DEMUX
- c.) Dekódoló (decoder)
- d.) Kódoló (encoder)
- e.) Komparátor (Comparator)

a.) Multiplexer (MUX) – útvonal választó

■ N kiválasztó jel, 2^N bemenet \rightarrow 1 kimenet

■ Példa: 4:1 MUX (74LS153)

$$Y = \overline{S_0} \cdot \overline{S_1} \cdot I_0 + \overline{S_0} \cdot S_1 \cdot I_1 + S_0 \cdot \overline{S_1} \cdot I_2 + S_0 \cdot S_1 \cdot I_3$$

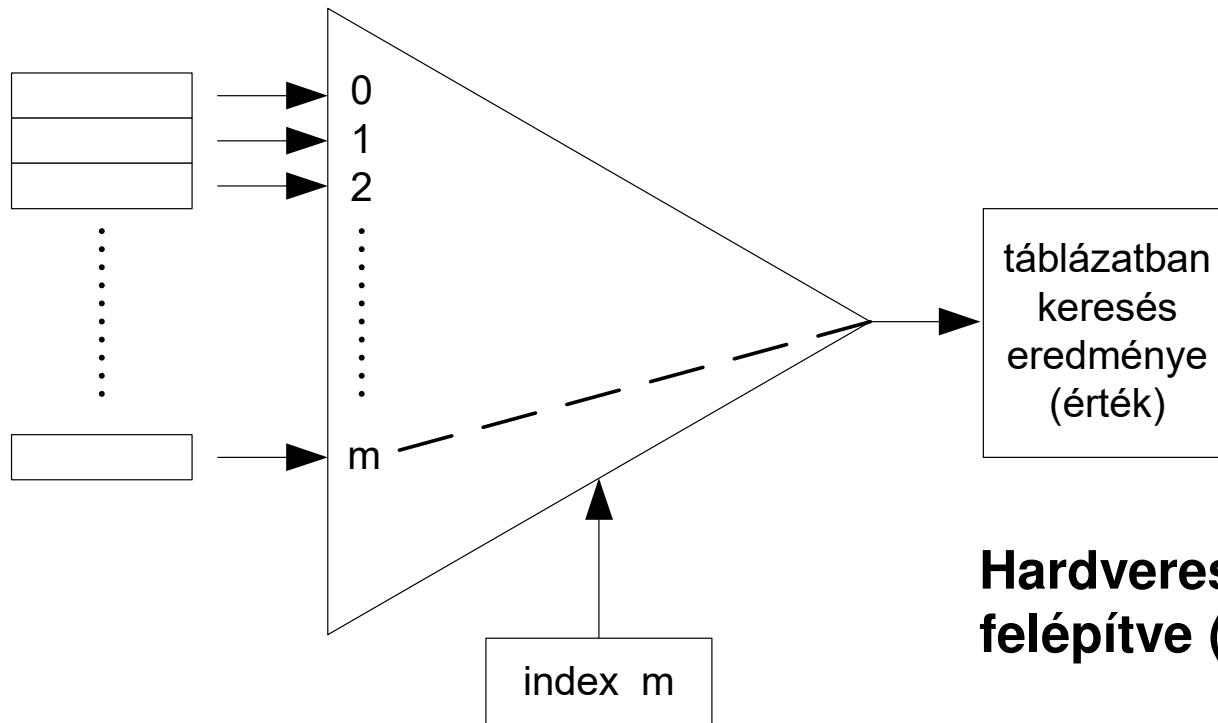


2^N számú bemenet közül választ egyet (Y), mint egy kapcsoló.
Rendelkezik EN bemenettel is.

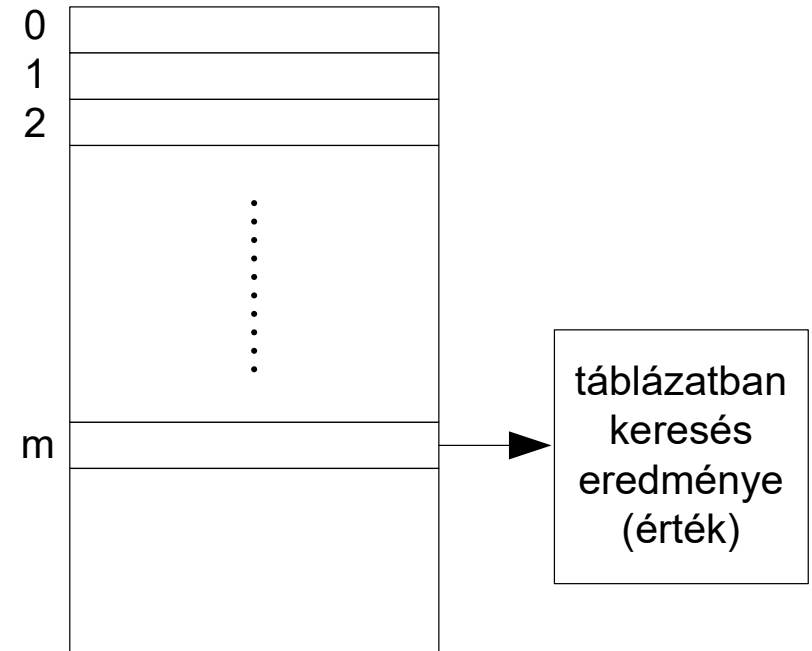
S_1	S_0	Y
0	0	$I(0)$
0	1	$I(1)$
1	0	$I(2)$
1	1	$I(3)$

PI. LUT megvalósítások:

Szoftveres Look-up-Table



$n(w)=1$ -bites
bejegyzések a
memóriában



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

b.) Példa - 1:4 Demultiplexer

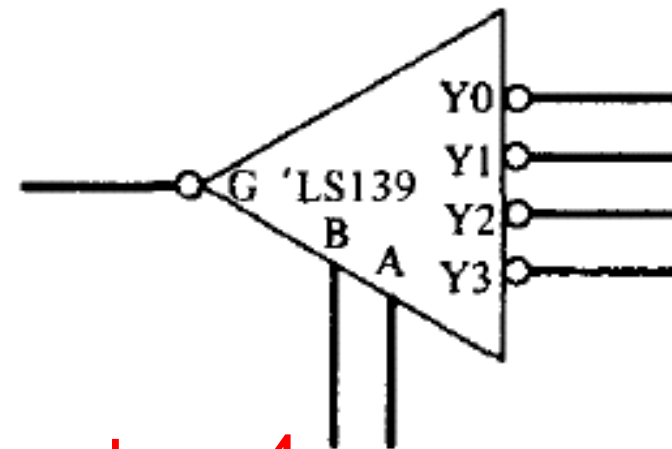
■ TTL 74'LS139 dual 1:4 demultiplexer

□ G: egy bemenetű

□ A,B: selector jelek

□ N kiválasztó jel, 1 bemenet → 4 kimenet valamelyikére „továbbítja”

- 4-kimenet mindegyike False=1, egyet kivéve, amelyik a kiválasztott (annak az értéke a bemenettől függően lehet T/F)



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

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

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

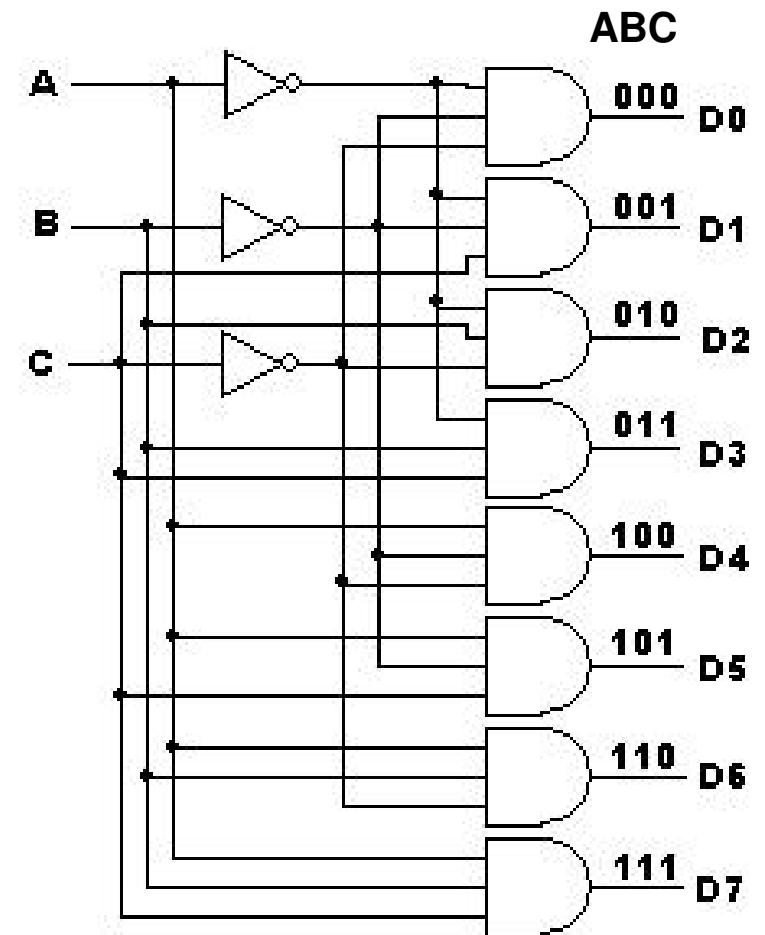
$$Y3 = B \cdot A \cdot G$$

T=L !

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

c.) Dekódoló áramkör

- N bemenet esetén $\rightarrow 2^N$ dekódolt kimenete van
- N bemenetből mindig csak egy aktív logikai értékű
- Példa: 3 \rightarrow 8 dekóder áramkör (SN 74LS138N)
 - Példa: Hamming-kódú hibajavító áramkör

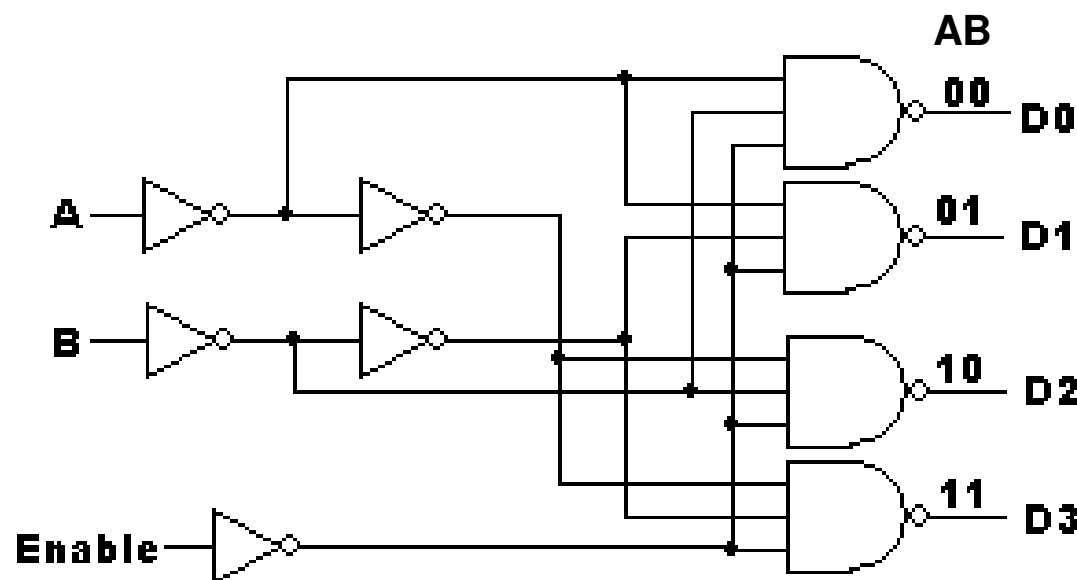


$T=H$

SELECT INPUTS			OUTPUTS							
C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
L	L	L	L	H	H	H	H	H	H	H
L	L	H	H	L	H	H	H	H	H	H
L	H	L	H	H	L	H	H	H	H	H
L	H	H	H	H	H	L	H	H	H	H
H	L	L	H	H	H	H	L	H	H	H
H	L	H	H	H	H	H	H	L	H	H
H	H	L	H	H	H	H	H	H	L	H
H	H	H	H	H	H	H	H	H	H	L

Példa: 2→4 Dekódoló áramkör engedélyező bemenettel

- SN74HC139
- EN: alacsony aktív állapotban működik
- 2 bemenő jel (A,B)
- 4 kimenet (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
 - 2^N bemenet esetén \rightarrow N kódolt kimenete van
 - Pl: $8 \rightarrow 3$ kódoló (pl. 74LS148)
 - **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, címek 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:

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

Ismétlés: EQ/XNOR/Coincidence operátor

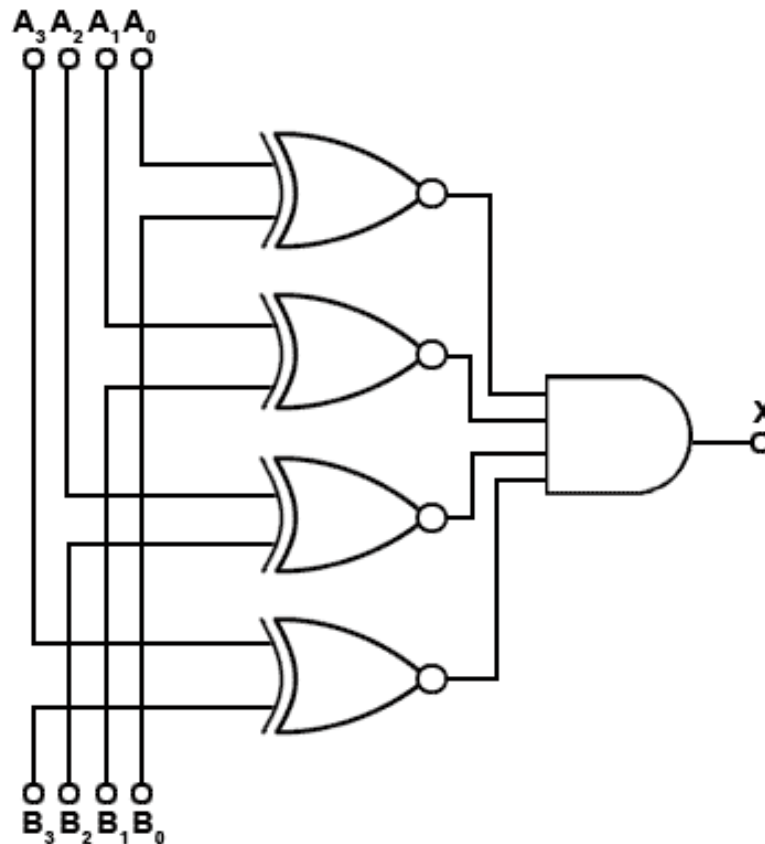
- Logikai egyenlet: $A.EQ.B = A \odot B = A \cdot B + \overline{A} \cdot \overline{B}$
- Referenciabit szerinti megkülönböztetés:
 - ha a referencia bit (B), amihez hasonlítunk **konstans**
 - ha a referencia bit (B) egy **változó** mennyiség
- Példa: ha B referencia konstans -> egyszerűsítése A-nak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

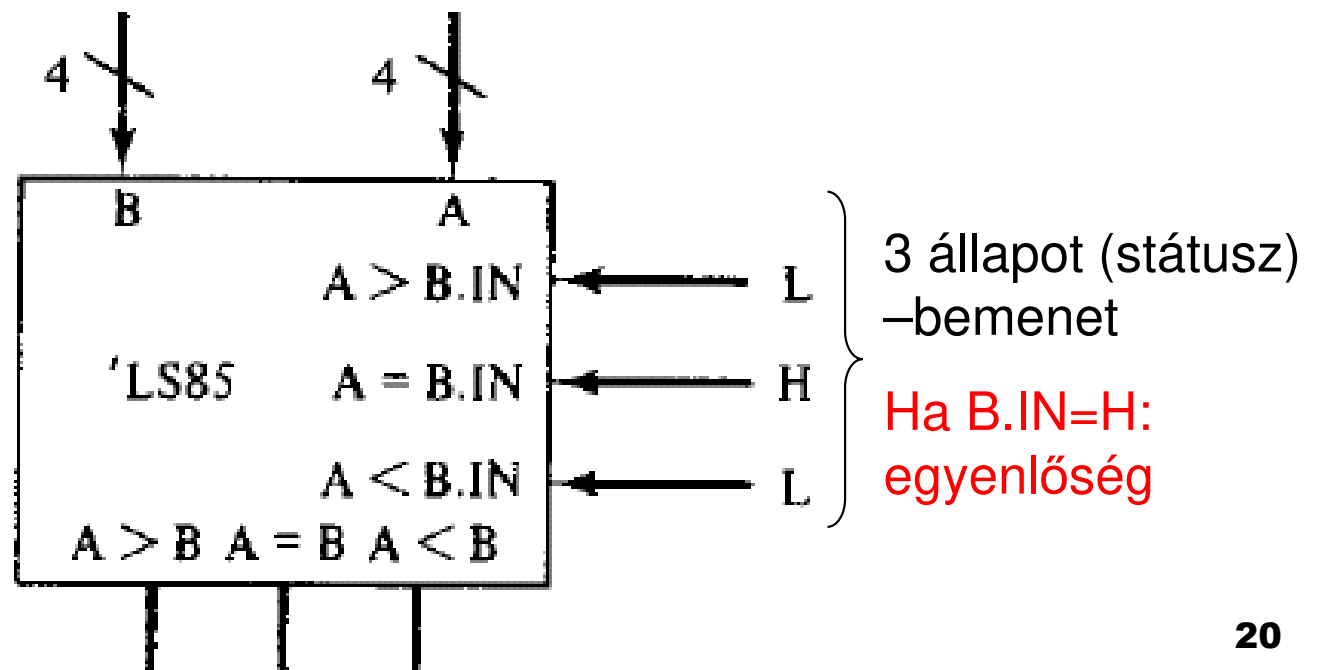
- Mixed-logic kapcsolási rajza, és log.egyenlete:

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



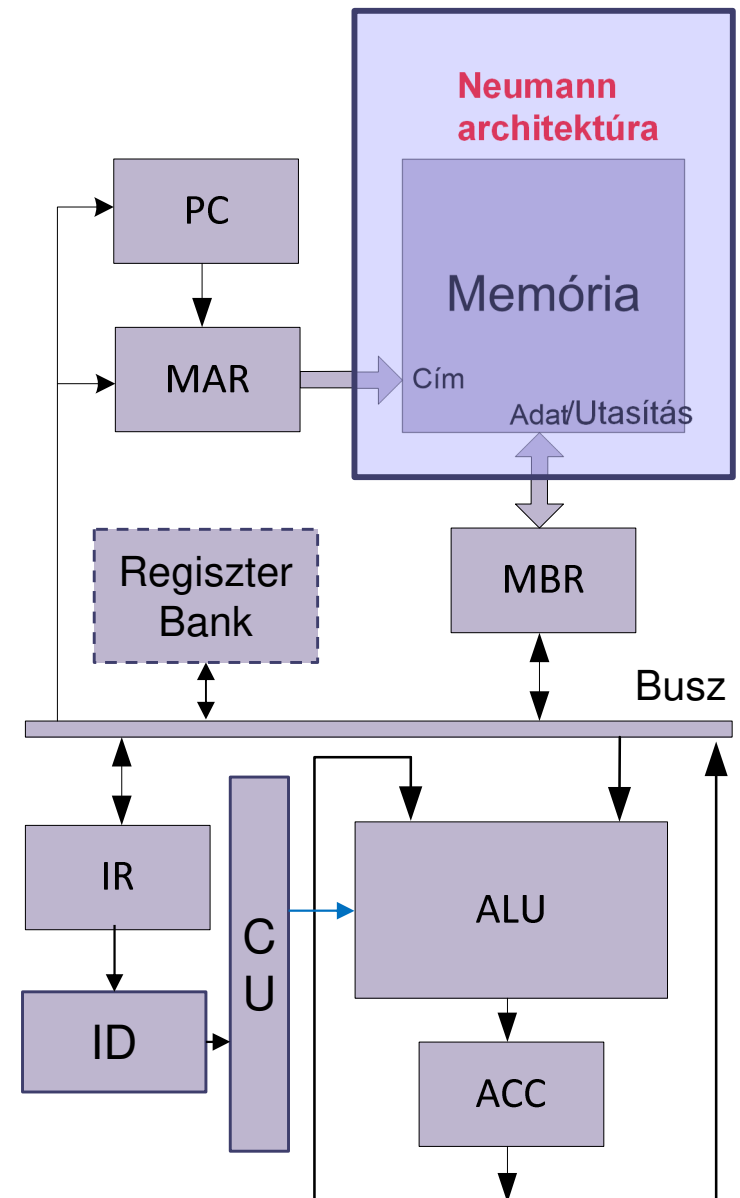
SN74LS85 4-bit Magnitude Comparator

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



Egyszerű (1-című) számítógép felépítése

- **MAR: Memory Address Register** (Memória-cím Regiszter): adott memóriacímen lévő adat/utasítás helyét azonosítja.
- **MBR: Memory Buffer Register** (Memória Puffer Regiszter): tárolja a memóriába beírt, ill. kiolvasott információt. Mivel az adat kiolvasásakor akár törölődhet (destruktív memória).
- **PC: Program Counter** (Programszámláló): a soron következő (végrehajtandó) utasítás helyét azonosítja. Ha 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, ALU egységek vezérlő vonalait (CU) a végrehajtáshoz szükséges működési módba állítsák. Olyan széles, hogy az utasítás műveleti kódja ill. a hozzá tartozó egyéb utasítások ideiglenes másolatai eltárolhatók.
- **ID: Instruction Decoder** (Utasítás dekódoló): az IR-be töltött aktuális utasítás értelmezése, részműveletekre bontása, majd pedig a **CU (Control Unit)** vezérli a teljes rendszert, vezérlő jelek generálása.
- **ACC: Accumulator regiszter** (tároló regiszter): eredmény ideiglenes tárolására használjuk, bizonyos utasítás kódok esetén mindenképpen használni kell (pl. 1-című).
- **Regiszter Bank:** általános célú regisztereket tartalmazó tároló tömb, a műveletek gyorsításához (operandusok betöltése innen történhet)





Utasítások kódolása

Gépi kódú utasítások

- Binárisan kódolt vezérlő információ, amely a processzort valamilyen művelet végrehajtására utasítja.
- A számítógépek gépi utasításainak összességét a számítógép **gépi nyelvének** = **utasításkészletének** nevezzük (**ISA** = Instruction Set Architecture).
- A gépi nyelv általában kötött (fix, nem bővíthető).
- Egy gépi kódú program csak akkor futtatható egy adott számítógépen, **ha a gép tudja értelmezni (interpreter) a gépi kódú a programot ↔ illető gép "nyelvén" írták.**

Gépi kódú utasítások

Két részből állnak :

- 1. rész: műveleti kód (operátor)
- 2. rész: operandus(ok)

Műveleti kód részei:

- 1. operandus címe (helye)
- 2. operandus címe (helye)
- végeredmény

1-, 2, 3-
című
utasításkód

- következő utasítás tárcíme
 - műveleti kód
 - 1. operandus címe
 - 2. operandus címe

4-, vagy
többcímű
utasításkód

Utasítás végrehajtás

M. J. Flynn taxonómia (1972) – rendszertan

■ Soros/szekvenciális

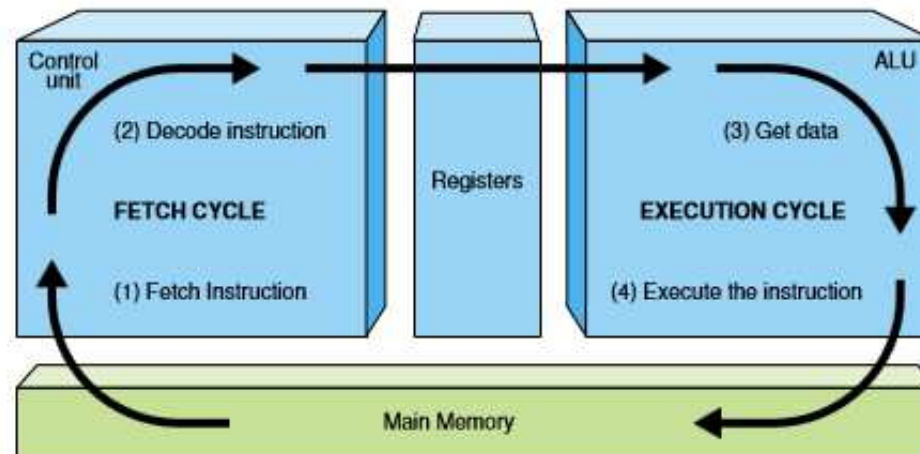
- *SISD* = **S**ingle **I**nstruction – **S**ingle **D**ata (lásd Neumann elvek sorrendi végrehajtása)
- *SIMD* = **S**ingle **I**nstruction – **M**ultiple **D**ata (vektor-, mátrix műveletek)

■ Párhuzamos/parallel

- Több feldolgozó egység megléte szükséges (több processzor vs. több mag vs. több szál)
- *MISD* = **M**ultiple **I**nstruction – **S**ingle **D**ata
- *MIMD* = **M**ultiple **I**nstruction – **M**ultiple **D**ata (pl. GPU-k, mai modern többmagos CPU-k, MPU-k, szuperszámítógépek)

FDE mechanizmus

- Egy utasítás végrehajtásának 3 fő lépését a **Fetch-Decode-Execute** (FDE) mechanizmussal definiálhatjuk:
 - **F-Fetch:** az utasítás betöltődik a memóriából az utasításregiszterbe (RTL művelet)
 - **D-Decode:** utasítás dekódolása (értelmezése), azonosítása
 - **E-Execute:** a dekódolt utasítást végrehajtjuk az adatokon, aminek eredménye visszakerül a memóriába
- Mai modern CPU architektúrák 15-30!! fázissal rendelkeznek (FDE-t további al-fázisokra osztva)
- További lépések lehetnek még (tipikusan) „5-lépcsős”:
 - **MEM: Memory operations:** következő utasítás letöltése a memóriából
 - **WB: Memory Write-Back:** eredmény visszaírása



Gépi kódú utasítások formái

■ 0-című (zéró-című) utasítás

Műveleti kód

(PI: Verem/Stack/Push-Down-Automata)

■ 1-című utasítás

Műveleti kód

Operandus címé (eredmény is)

ACC: akkumulátor bevezetése

■ 2-című utasítás

Műveleti kód

1. Operandus címé

2. Operandus címé (eredmény is)

■ 3-című utasítás

Műveleti kód

1. Operandus címé

2. Operandus címé

Eredmény címé

■ 4-című utasítás

Műveleti kód

1. Operandus címé

2. Operandus címé

Eredmény címé

Következő ut. címé...

PC: Program számláló bevezetése

Assembly

- **(ASM) Alacsony-szintű gépi kódú utasítások nyelve**
- Minden utasítás (**mnemonic**) a processzor egy utasításának felel meg! Rövid tömör kód.
- Régen ebben programozták a CPU-kat.
- Manapság: ált. magas-szintű nyelvek alkalmazása (C, C++)
 - „**Szemantikai rés**”: magas-, vs. alacsony-szintek közötti leképezést a fordítóprogram (compiler) biztosítja!

$Z = X + Y$

Megfelel: $\text{ADD}_3 \ R_1, R_2, R_3$
(3-című, R=Regiszteres utasítás)

```
LD      X, $R1           //Load
LD      Y, $R2
ADD     $R1, $R2, $R3
ST      $R3, Z           //Store
```

RTL leírás:

- Minden utasítás végrehajtása az **RTL leírás (Regiszter-Transzfer Nyelv)** segítségével írható le. A szükséges adatátvitelleket ezzel a nyelvvel specifikáljuk az egyik fő komponenstől a másikig.
- Továbbá megadható az engedélyezett adatátvitelhez tartozó **blokkdiagram** (vagy **gráf** leírás) is, az éleken adott irányítással, melyek 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.

1-című utasítás

Műveleti
kód

Operandus címe
(eredmény is)

- Tipikusan egyszerűbb CPU-k utasításai (pl. 8-, 16-bites)
- **Egy operandust azonosítunk**, melyek a memóriából olvasunk ki, a másik operandust, ill. az eredményt az CPU speciális regiszteréből (ACC – akkumulátor) olvassuk ki, ill. tároljuk el.
 - Egyetlen operandus címezhető (memória), a másik operandus már eleve a CPU belső (ACC) regiszterében van – „ACC intenzív” használat,
- Címzési mód alapján a közvetett (operandusok címe) és közvetlen adatelérést (érték szerint) is támogatja
 - Közvetett (indirekt) címzés: memória
 - Közvetlen (direkt) címzés: „speciális” tároló → általános célú regiszter
 - Regiszter utasítások gyorsak → HW felépítés + beírás/kiolvasás a memóriából nem kell.

$\text{ADD}_1 \text{ X} = \text{ACC} + \text{X} \Rightarrow \text{ACC} \Rightarrow \text{X}$

Példa: Egy-című utasítás (RTL kód)

2's komplement képzés ACC-vel

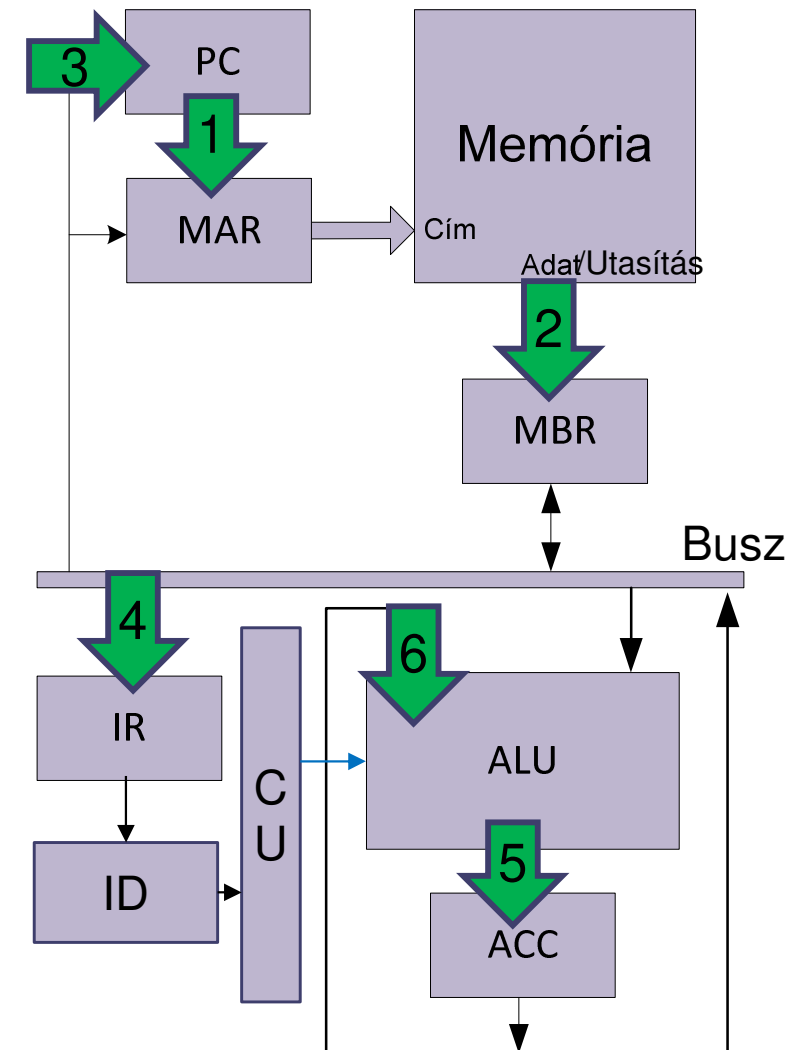
Fetch: (regiszterek feltöltése, utasításhívások):

1. **PC**→**MAR**
PC-ből a következő utasítás címe a MAR-ba töltődik
2. **M[MAR]**→**MBR**
Memóriában lévő utasítás kiolvasása az MBR-be
3. **PC+I_len**→**PC**
Az utasítás hosszával (I_len) növeli a PC értékét
4. **MBR**→**IR**
Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (~min 1 órajel ciklus idő)

Execute: (végrehajtás)

5. **ACC** →**ACC**
ACC 1's komplementjét az ACC-be töltjük
6. **ACC+1**→**ACC**
majd ACC-t '1'-el inkrementáljuk (eredmény)



Példa: DEC PDP-8
számítógépe

Példa: 1-című gép (Kivonás SUB₁X)

„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 értékét	Az utasítás hosszával (I_len) növeli a PC
MBR→IR	Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (~min 1 órajel ciklus idő)

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 SUB₁X)

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: (~min 1 órajel ciklus idő)

Execute: (végrehajtás)

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

2-című utasítás

Műveleti
kód

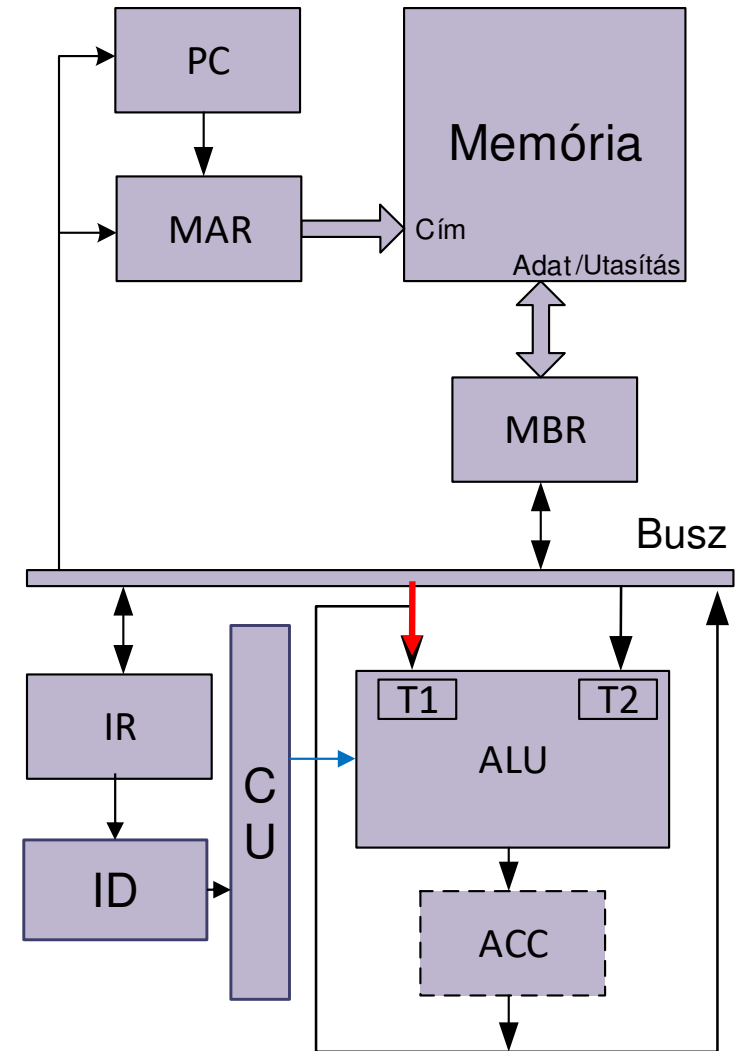
1. Operandus
címe

2. Operandus
címe (eredmény is)

- A korszerű CPU-k utasításai (32-, 64-bites)
- **Két operandust** azonosítunk, ill. a művelet elvégzése után az eredmény a második operandus helyén tárolódik el (memóriában!)
 - Külön gondoskodni kell a második operandus mentéséről!
- Címzési módok támogatása (lásd: 1-című utasítás)
 - Közvetett (indirekt) vs. Közvetlen módszerek
 - CPU-ban gyorsító tároló elem → általános célú regiszter

SUB₂ X, Y = $X - Y \Rightarrow Y$

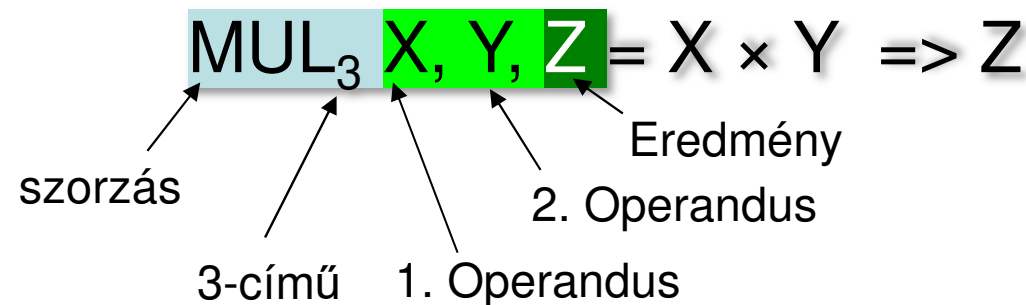
kivonás 2-című 1. Operandus 2. Operandus (eredmény is)



3-című utasítás

Műveleti kód	1. Operandus címe	2. Operandus címe	Eredmény címe
--------------	-------------------	-------------------	---------------

- A korszerű CPU-k utasításai
- **A két operandust és az eredményt külön** azonosítjuk!
- A művelet elvégzése után az eredmény egy új, különálló helyen tárolódik el (memóriában!)
 - **Az eredmény nem írja felül így már az operandus(oka)t!**
 - **3-, és többcímű számítógép felépítése azonos a 2-című architektúrával!**
- Közvetett (cím), és közvetlen adatelérést (érték szerint) is támogat
 - Közvetett (indirekt) vs. Közvetlen módszerek (lásd: 1-című utasítás)
 - CPU-ban gyorsító tároló elem → általános célú regiszter



Jelölés: kettő-, és többcímű gép

- Jelölés: **ADD₂ 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: **ADD₃ 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 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 T1, T2-t „csak” az ALU részeként, nem pedig a rendszer gyorsítását szolgáló elkülönített regiszter bankként használjuk.

Példa: Összeadás két-című géppel $\text{ADD}_2(X,Y)$

Időszükségletek feltüntetésével!

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

Fetch: (regiszterek feltöltése, utasításhívások):

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: (~min 1 órajel ciklus idő)

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] ADD_2 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: Összeadás három-című géppel $\text{ADD}_3(X,Y,Z)$

Időszükségletek feltüntetésével!

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

Fetch: (regiszterek feltöltése, utasításhívások):

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: (~min 1 órajel ciklus idő)

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
címezést
használunk
itt!**

Σ 295ns

Komplex műveletek:

MŰV₂ vs. MŰV₃ összehasonlítása

- A **MŰV₃** –nál (ahogy az RTL leírásból is látszott) egy sor végrehajtása több időt vesz igénybe az utasítások F-D-E fázisánál, mint az MŰV₂ esetén, mivel egyel több címre kell hivatkozni.
- Azonban, az **MŰV₃** jelentősége a **komplexebb műveletek** elvégzésekor mutatkozik meg → tömörebb forma, **kevesebb** utasítás sor!
- Példa: Legyen **$X = Y * Z + W * V$** (oldjuk meg MŰV₂-vel, és MŰV₃-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	

Példa: Kvadrátikus egyenlet

(2-, és 3-című utasításokkal)


Számítsuk ki 3-című utasításokkal az alábbi egyenletet!

$$X = (-b + \text{sqrt}(b^2 - 4*a*c)) / (2*a)$$



```
T1 := b * b
T2 := 4 * a
T3 := T2 * c
T4 := T1 - T3
T5 := sqrt(T4)
T6 := 0 - b
T7 := T5 + T6
T8 := 2 * a
X := T7 / T8
```

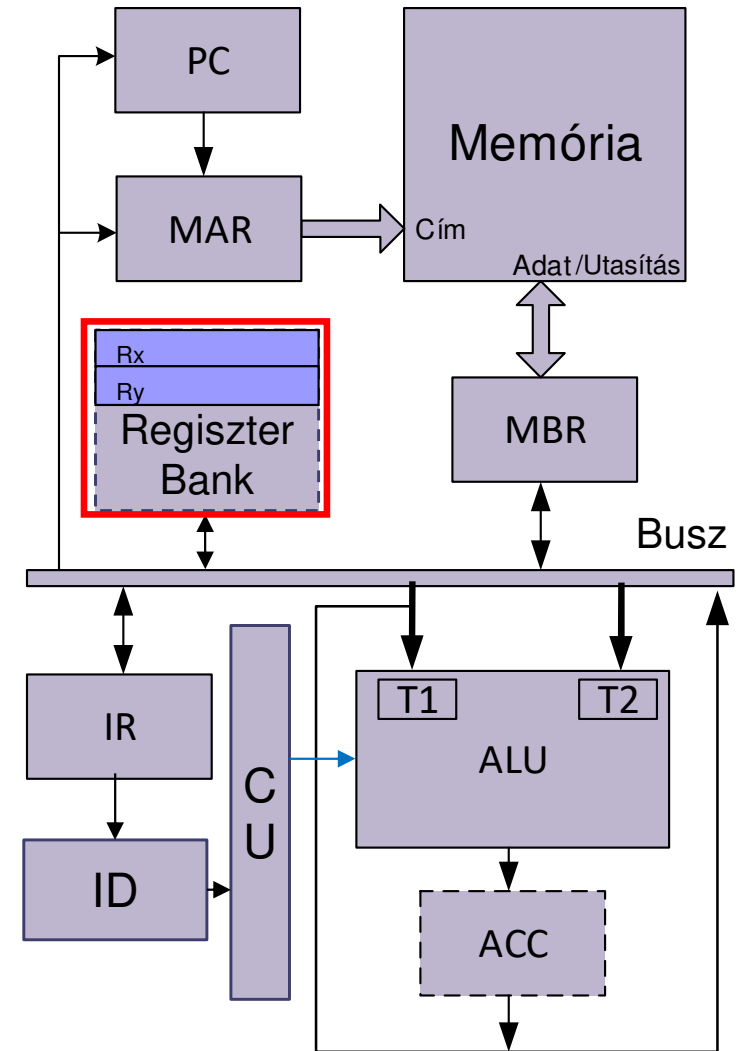
RTL utasítások



```
MUL3    b, b, T1
MUL3    4, a, T2
MUL3    T2, c, T3
SUB3    T1, T3, T4
SQRT2   T4, T5
SUB3    0, b, T6
ADD3    T5, T6, T7
MUL3    2, a, T8
DIV3    T7, T8, 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 regiszterműveleteket használnak. (A regiszterbank 2^N számú regisztert tartalmazhat.)



Példa 1: Összeadás kétcímű géppel $\text{ADD}_2(R_X, R_Y)$

Időszükségletek feltüntetésével!

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

Fetch: (regiszterek feltöltése, utasításhívások):

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: (~min 1 órajel ciklus idő)

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 $\text{ADD}_3(R_X, R_Y, R_Z)$

Időszükségletek feltüntetésével!

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

Fetch: (regiszterek feltöltése, utasításhívások):

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: (~min 1 órajel ciklus idő)

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

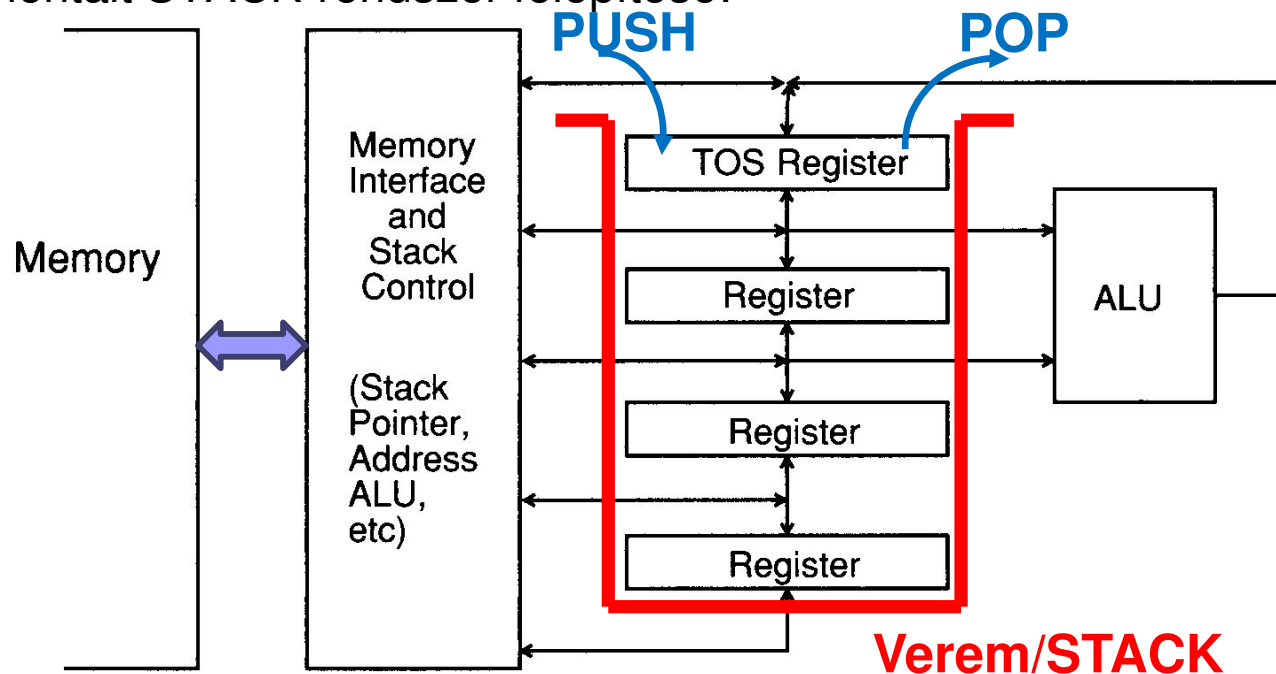
**Regiszteres,
direkt-címzést
használunk
itt!**

d.) Zéró-, vagy 0-című gép

- Zéró-című utasítás végrehajtásához **vermet (STACK)** használunk
- **Verem:** „**LIFO**”-típusú tároló (**L**ast-**I**n, **F**irst **O**ut), amelyből az utoljára betett adatot vesszük ki elsőként.
- STACK alkalmazása: **MŰV₀ ()** **PUSH, POP** →

Műveleti kód

 - Függvény hívás, és visszatérés (CALL-RETURN eljárások)
 - Aritmetikai műveletek is hatékonyan végrehajthatók használatával:
 - a szükséges operandusokat a STACK egy-egy rekeszében tároljuk: mindig a felső (kettő) regiszterből vesszük ki, és elvégezzük rajtuk a műveletet, majd az eredményt a verem tetejére tesszük vissza. **Azért nevezik zéró címűnek, mivel az operandusok azonosítására szolgáló utasításhoz nem használ címeket.**
 - A HW-orientált STACK-rendszer felépítése:



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, ill. ADD, DIV, MULT**

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 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 **(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 „címezési módok”

Gépi kódú utasítások (ism)

■ 0-című (zéró-című) utasítás

Műveleti
kód

(PI: Verem/Stack/Push-Down-
Autómata)

■ 1-című utasítás

Műveleti
kód

Operandus
címe (eredmény is)

ACC: akkumulátor bevezetése

■ 2-című utasítás

Műveleti
kód

1. Operandus
címe

2. Operandus
címe (eredmény is)

■ 3-című utasítás

Műveleti
kód

1. Operandus
címe

2. Operandus
címe

Eredmény
címe

■ 4-című utasítás

Műveleti
kód

1. Operandus
címe

2. Operandus
címe

Eredmény
címe

Következő
ut. címe...

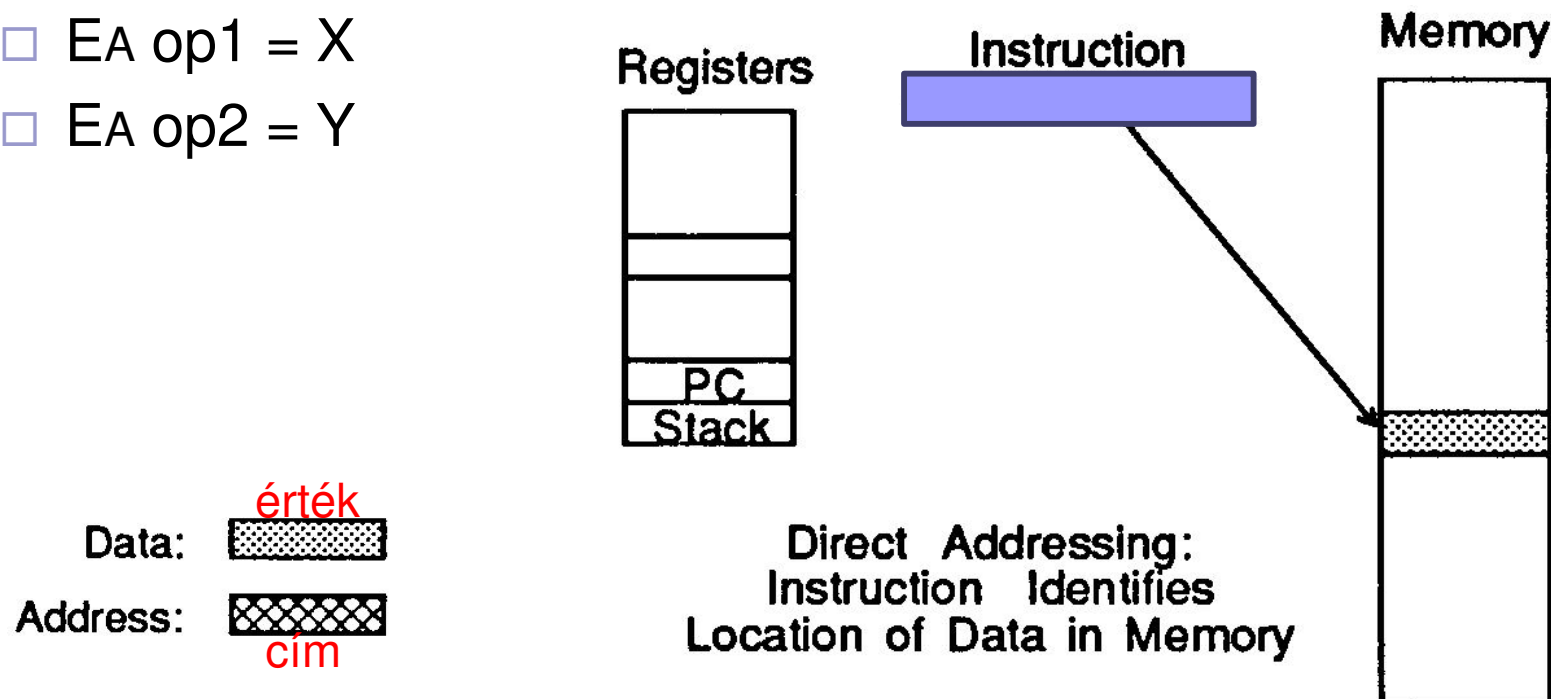
PC: Program
számláló
bevezetése

Operandus címezési módok

- Utasítás kód része: hogyan érjük el az operandust!
- Utasítás végrehajtásakor a kívánt operandust is azonosítjuk, címével hivatkozhatunk a pontos helyére (címre, értékre).
- Többféle címezési mód létezik:
 - ☐ közvetlen (directed),
 - ☐ közvetett (indirected),
 - ☐ indexelt (indexed),
 - ☐ regiszteres megvalósítású (register relative).
- Ezek kombinációja igen sokféle lehet: összesen akár 10-féle azonosítási módot tesz lehetővé.
- Jelölés: **E_A** = **Effektív (valódi) címe** az operandusnak

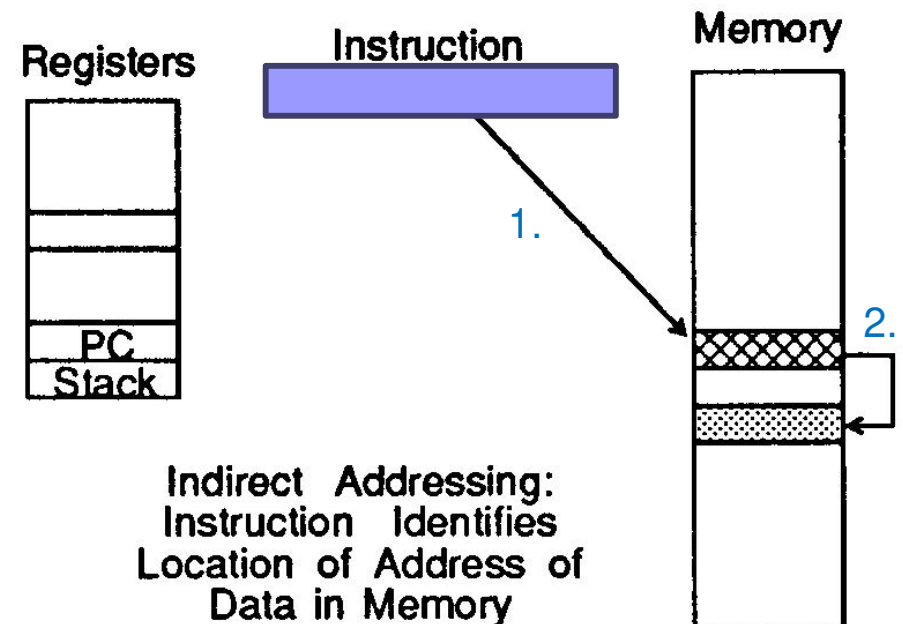
1. Direkt címezés (X)

- 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.)
 - $E_A \text{ op1} = X$
 - $E_A \text{ op2} = Y$



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, pointer**)
 - 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!



Data:  **érték**

Address:  **cím**

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

Időszükségletek feltüntetésével!

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

Fetch: (regiszterek feltöltése, utasításhívások):

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: (~min 1 órajel ciklus idő)

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 értéket megkapjuk MBR-ben
MBR→T1	[5ns] X értéket 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 értéket megkapjuk MBR-ben
MBR→T2	[5ns] Y értéket T2-be töltjük

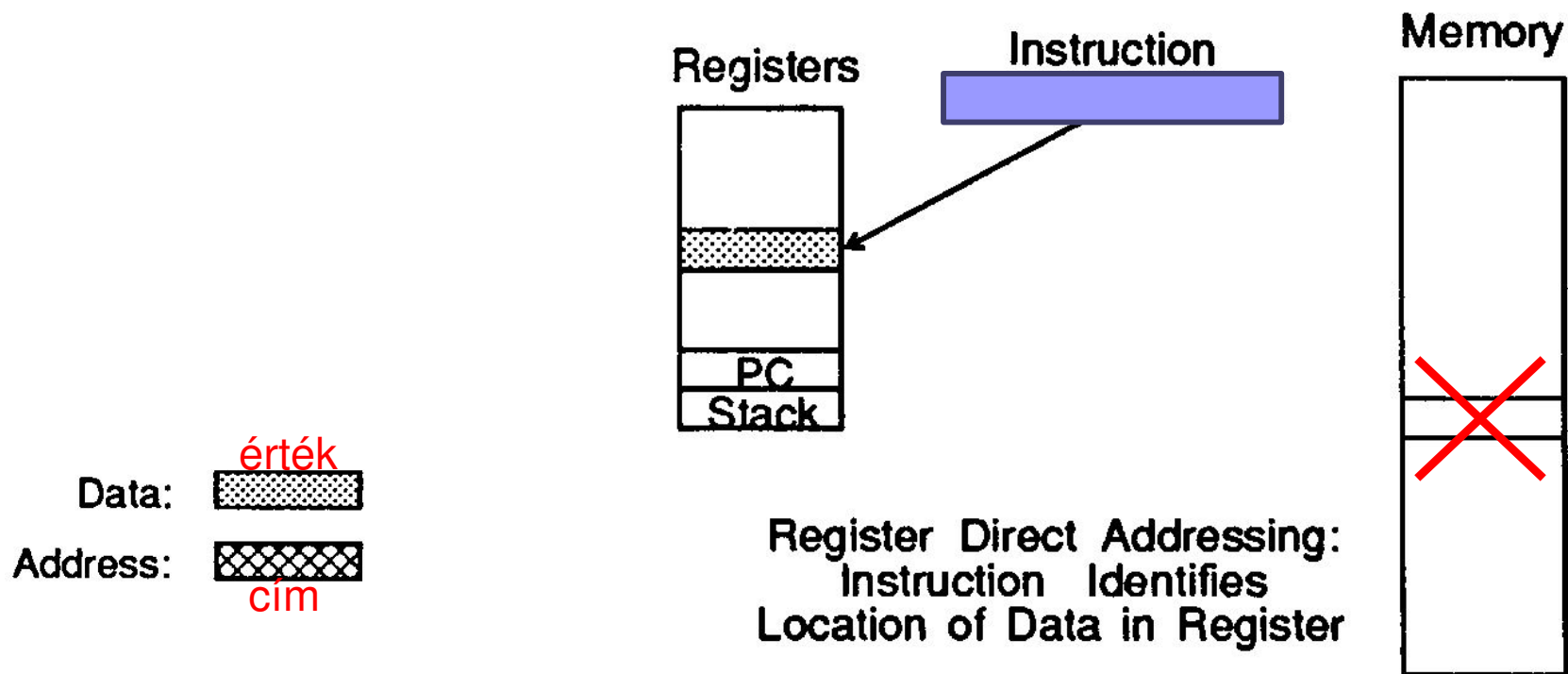
T1 + T2→MBR	[10+5ns] ADD2 művelet elvégzése, MBR-be töltjük
MBR→M[MAR]	[30ns] Eredményt a memóriában tároljuk el (ahol Y volt)

**Indirekt
címezést
használunk
itt!**

Σ 320ns

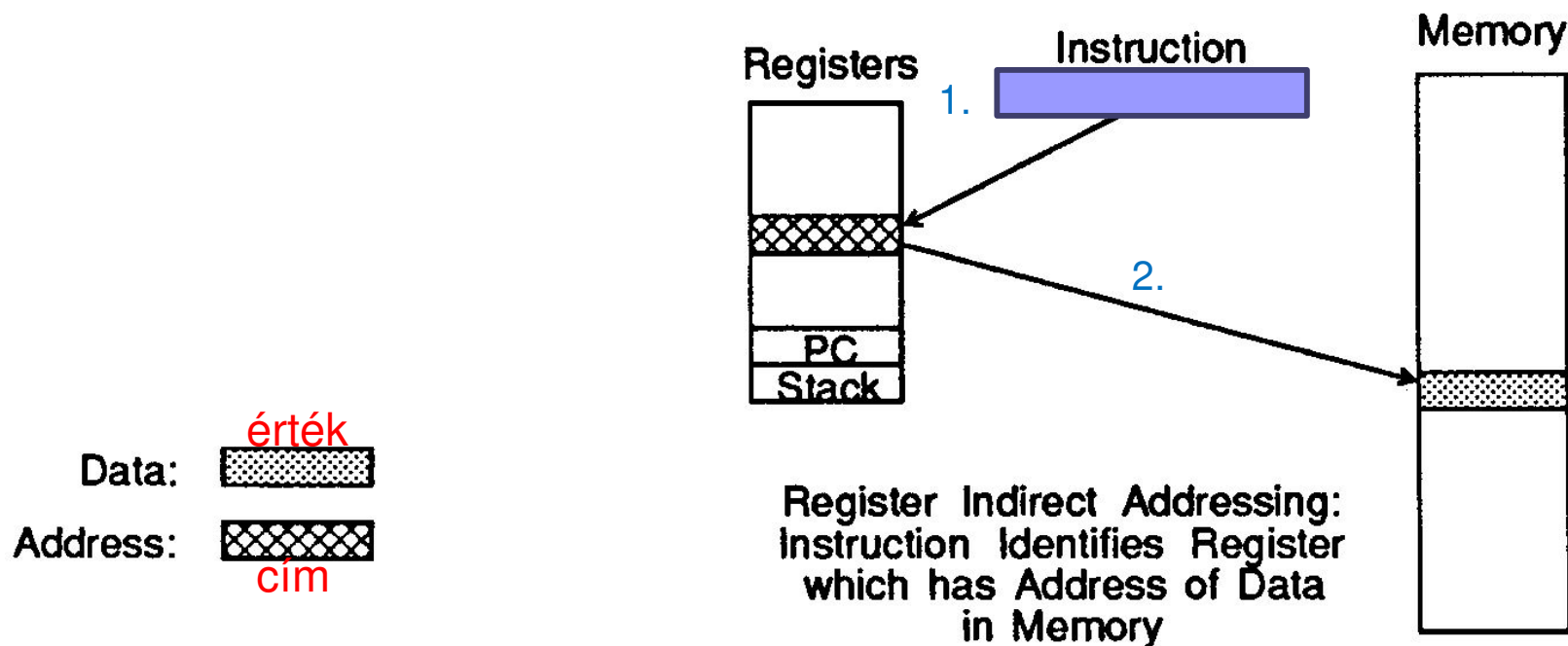
3. Regiszteres direkt címzés (R_x)

- Hasonló, mint a direkt címzés, de sokkal **gyorsabb**, mivel a memória intenzív-műveletek helyett az **operandusok értékeit** a CPU gyors **belső regisztereiben** 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_x)

- Hasonló, mint az indirekt címzés, de sokkal gyorsabb, mivel a memória-intenzív műveletek helyett a köztes **címeket** a CPU **gyors belső regiszterekben** tárolja, és csak a végén hivatkozik a memóriára (operandus értékére). A 3.) regiszteres módszer után ez a második leggyorsabb.



Példa: Összeadás kétcímű géppel $\text{ADD}_2(*R_X, *R_Y)$

Időszükségletek feltüntetésével!

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

Fetch: (regiszterek feltöltése, utasításhívások):

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: (~min 1 órajel ciklus idő)

Execute: (végrehajtás)

RX → MAR	[5ns] RX címét a MAR-ba töltjük	Regiszteres indirekt- címzést használunk itt!
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

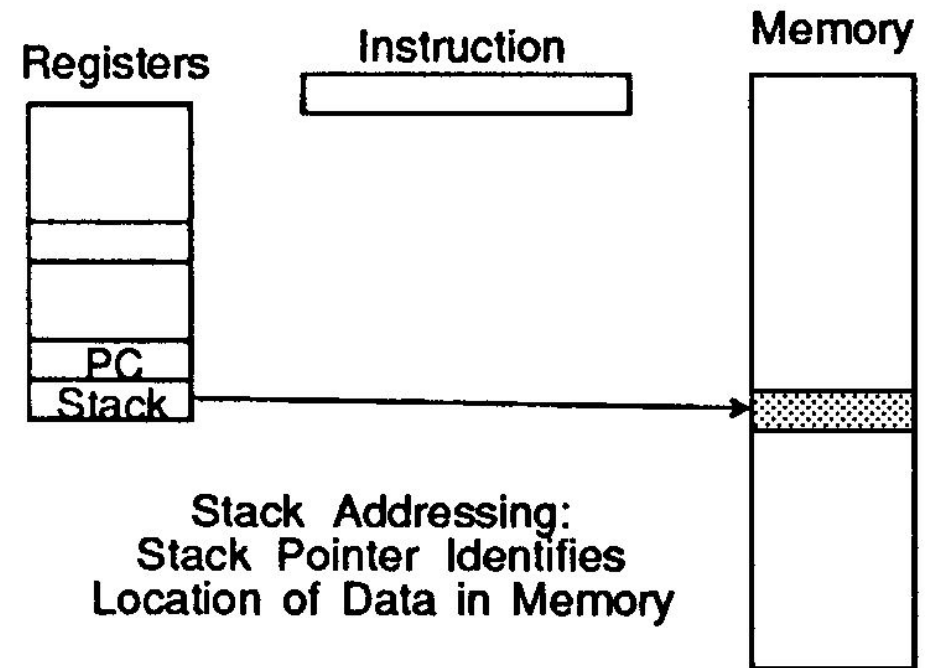
Összehasonlító táblázat – I.


- 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:* mivel a címeket automatikusan növeli. Ezt (+) jellel jelöljük: ***Rx+**
- A Stack-et a regiszterekből foglalhatjuk le. A stackben lévő információra a stack pointerrel (SP-mutatóval) hivatkozunk a memóriára. A Stack egy *LIFO tároló*.
- A stack pointer (SP) címe jelzi verem tetejét (ToS-Top of Stack), ahol a hivatkozott információ található, ill. címmel azonosítható a következő elérhető hely.



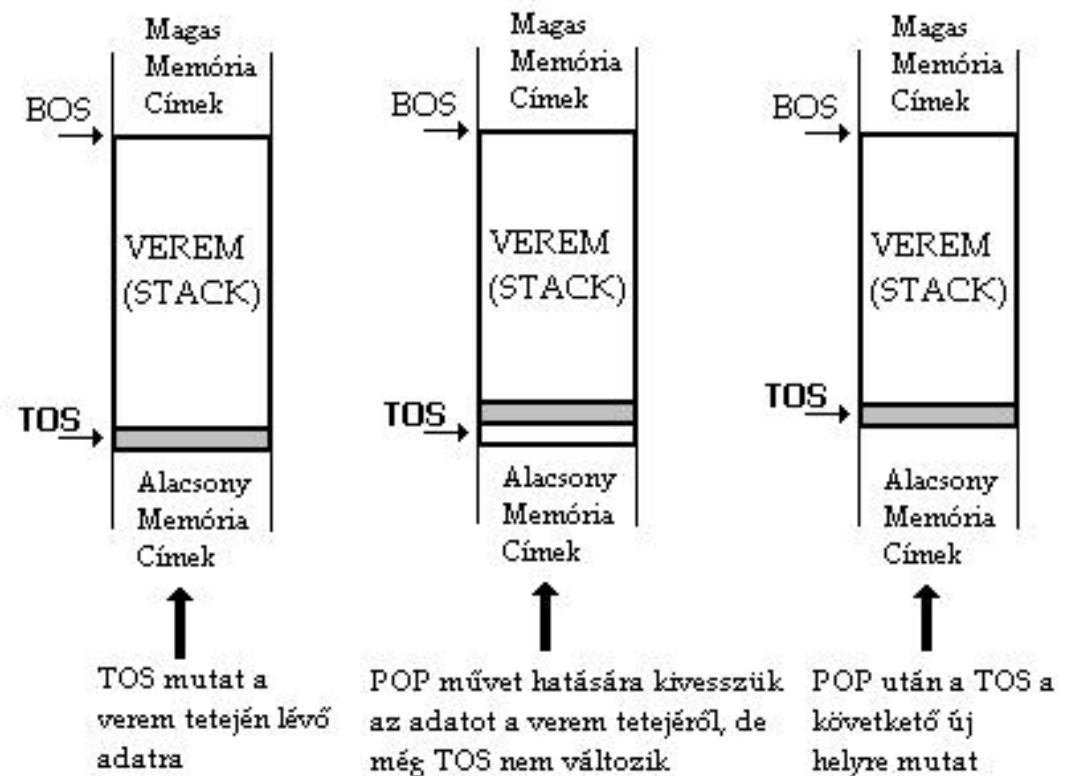
Data:  érték

Address:  cím

Verem – PUSH, POP műveletek

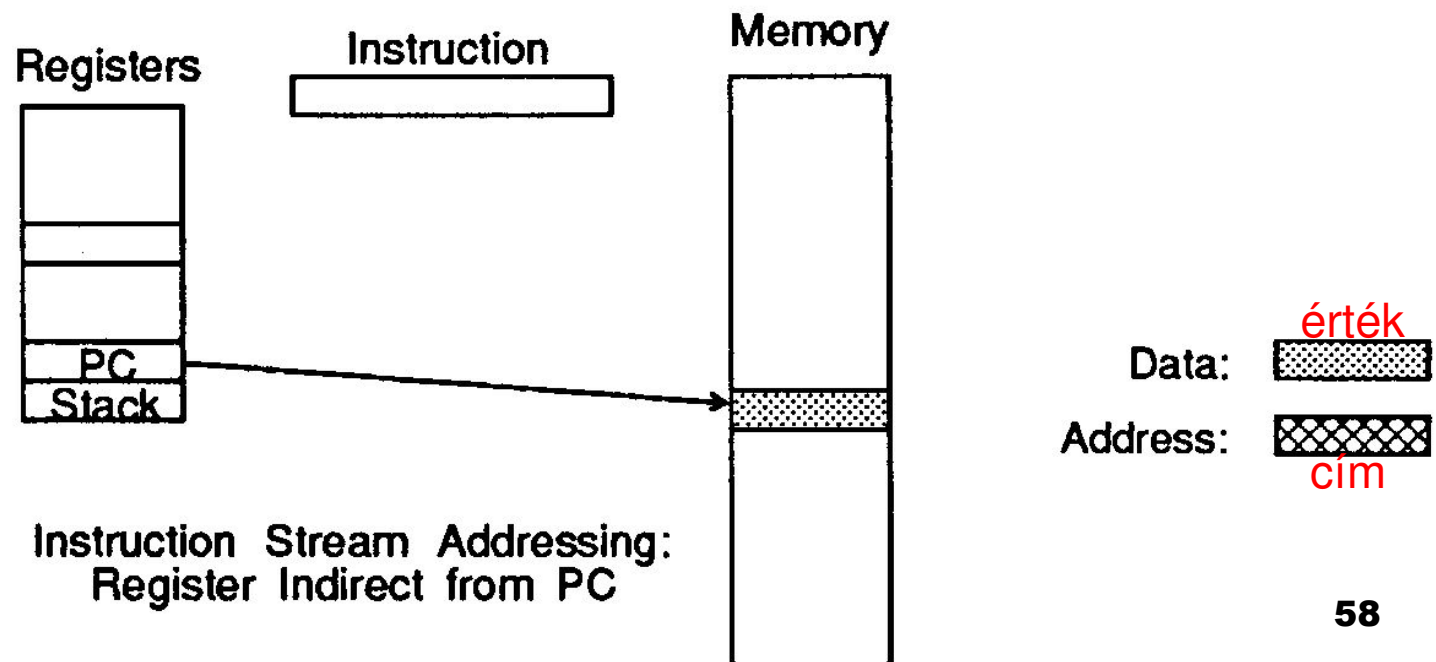
- **POP művelet** kivesszük a stack tetején lévő adatot (TOS), és egy Rx regiszterbe rakjuk. Ezután a stack pointer automatikusan inkrementálja a címet, amivel a következő elemet azonosítja a verem tetején.
Jel: **MOVE *R (stack pointer)+, Rx**
- **PUSH művelet:** a stack pointer automatikusan dekrementálja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



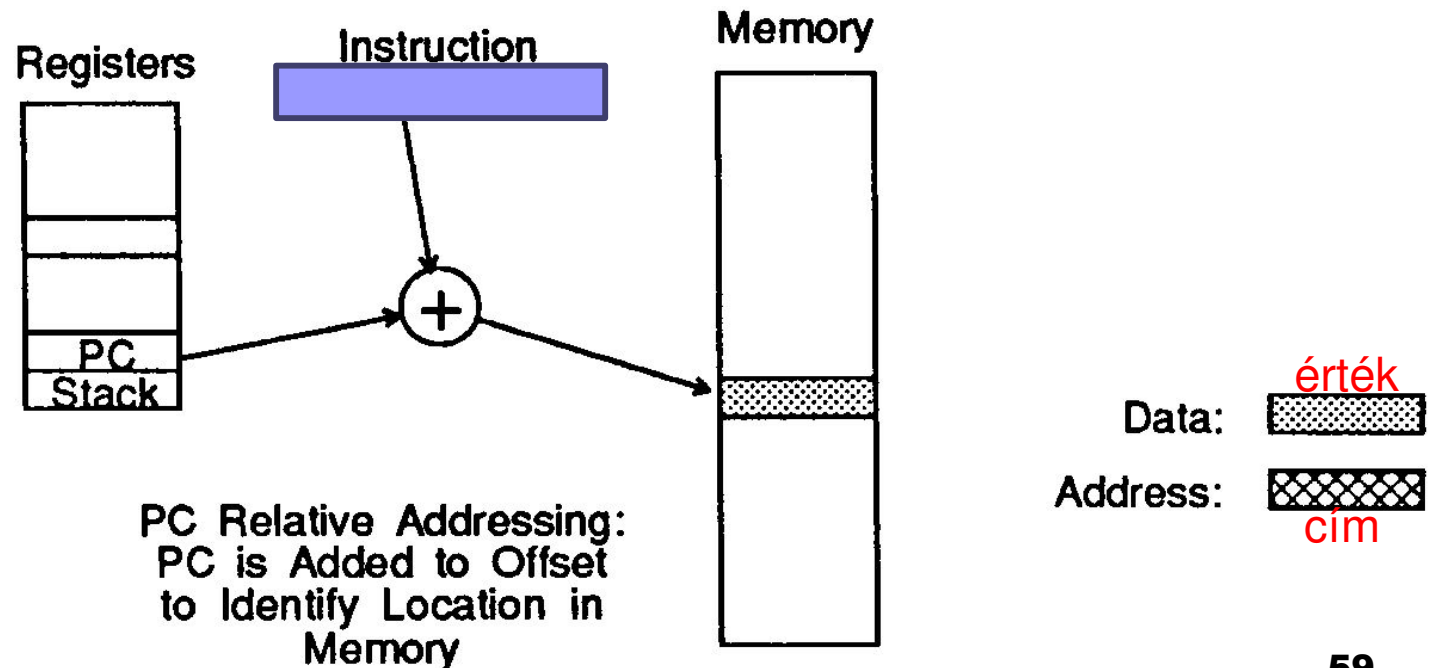
6. Instruction Stream címzés

- A PC-ben tárolt utasítás cím segítségével közvetlenül azonosítja a memóriában lévő operandust. 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, előre definiált címek szerepelhetnek az utasítás-folyamban.



7. PC-relatív címzés

- Memóriában lévő operandusra a regiszteren belüli PC értéke, és az utasítás eltolási (offset) értéke együttesen azonosítja.
 - Effektív cím = Regiszteren belüli PC értéke + Eltolás (offset) értéke

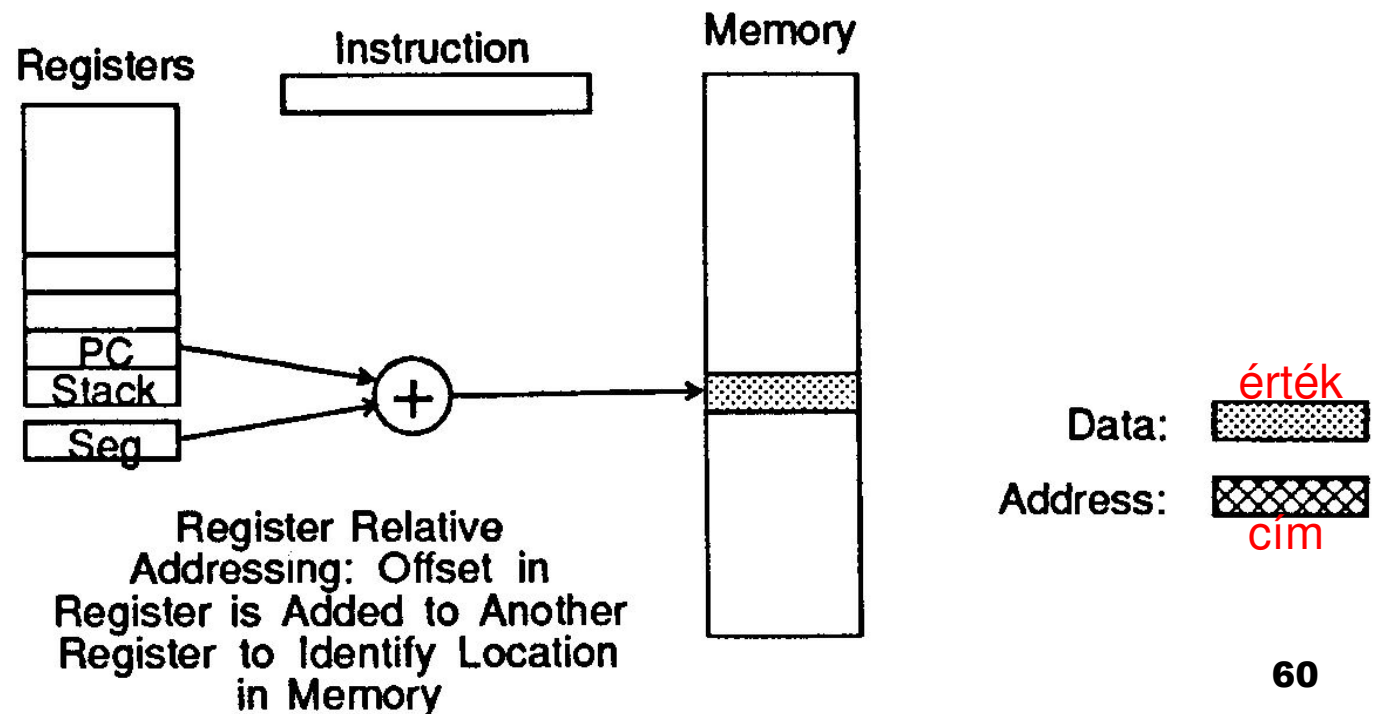


8. Regiszter relatív címzés

- A memóriában lévő operandusra a regiszteren belüli PC 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

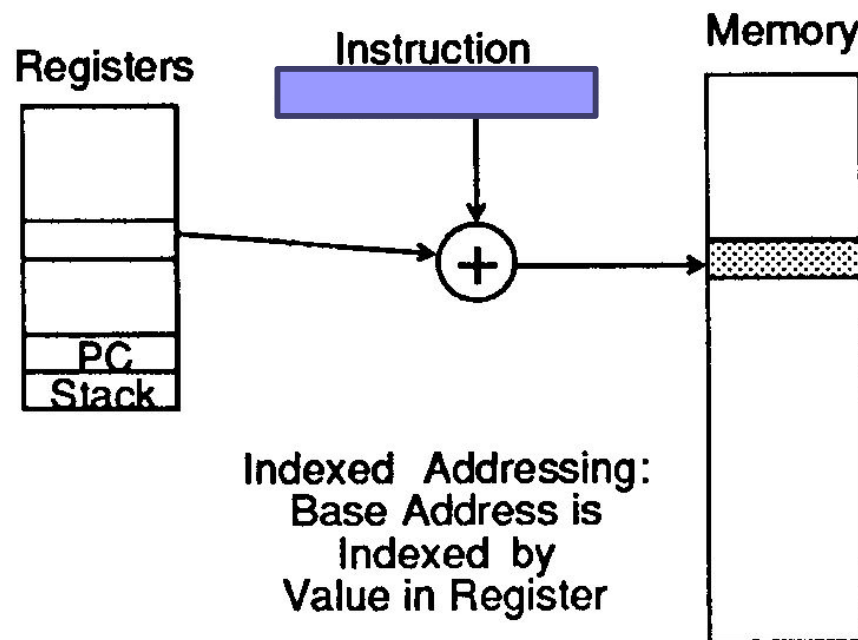
PI: Intel 80x86
szegmensek:

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 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;  
}
```

Összehasonlító táblázat – II.

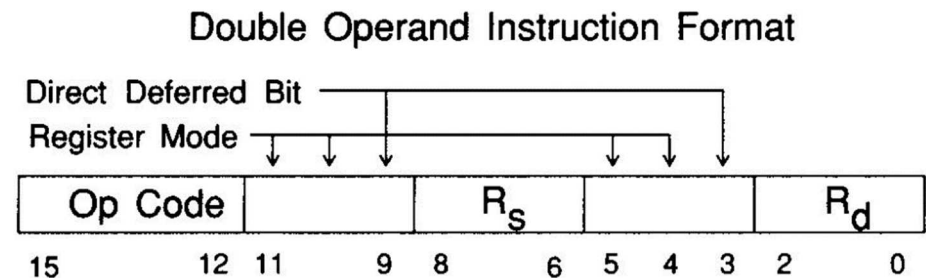
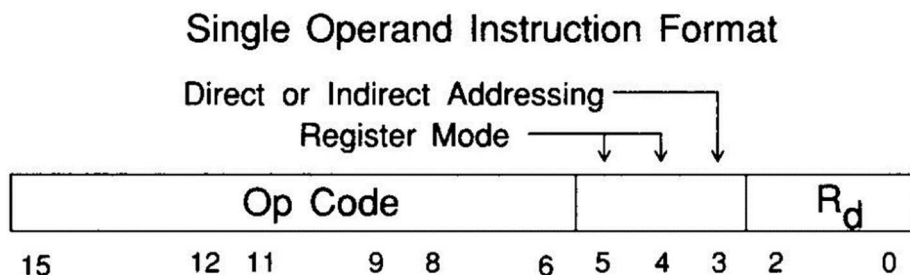
- Címzési módok és jelöléseik összefoglaló táblázata

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

- Egyszerű számítógépünk támogatott utasítása(i)
- Különböző címezési módokat használt
- *16-bites* gép: utasítás opcode-ja + további infók (cím)
 - 3 biten: 8 általános célú regiszter ([2:0]) – **R_d**-nek lefoglalt rész
 - További 3 biten: **R_{source}** (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






Programszervező utasítások

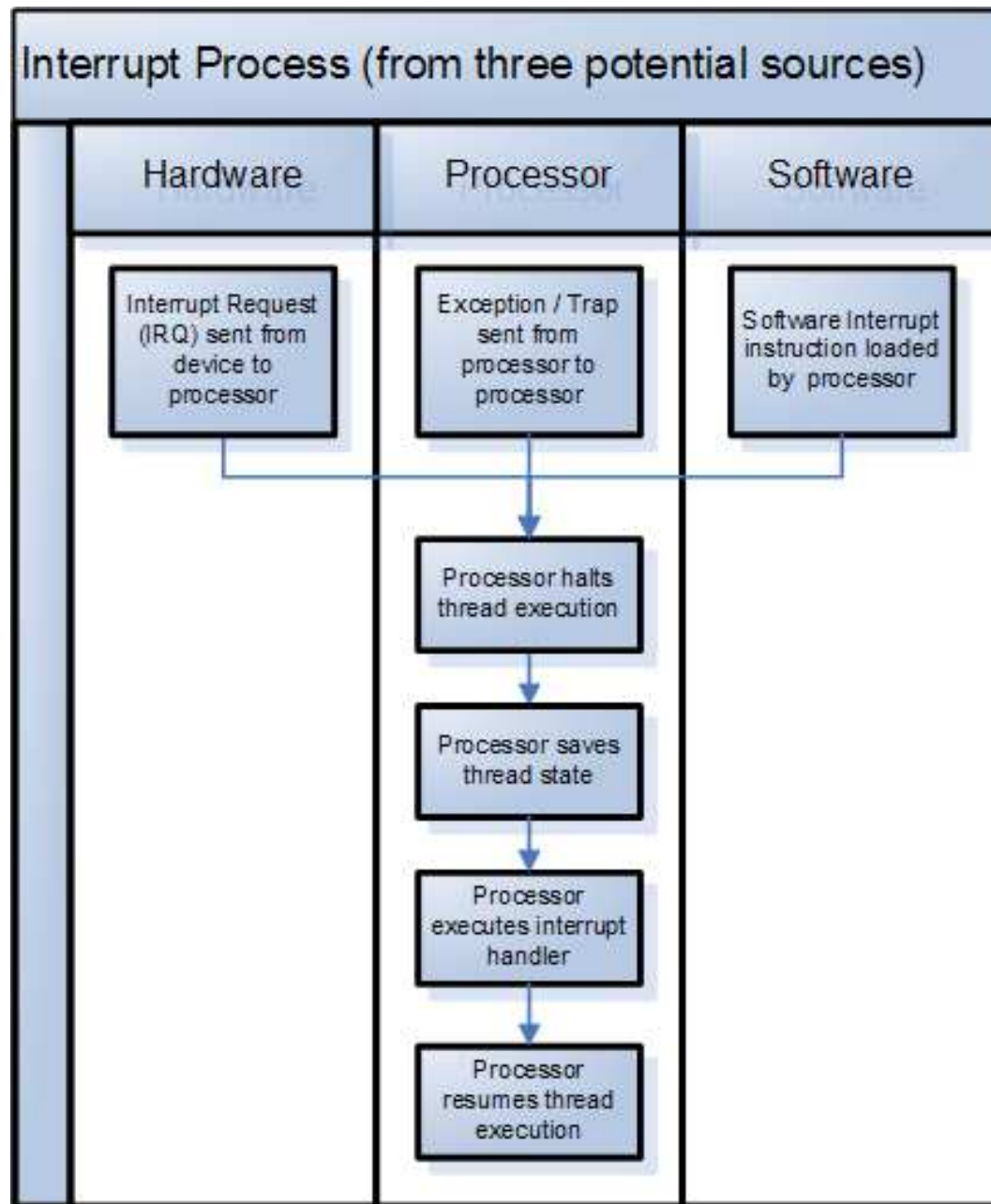
Programszervező utasítások

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

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 kezelése (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

Utasításkészlet csoportosítása

- Aritmetikai utasítások
- Logikai utasítások
- Vezérlésátadó utasítások
- Verem-kezelő utasítások
- Adat-kezelő utasítások
- Lebegőpontos utasítások
(nem mindegyik CPU támogatja)
- Multimédiás utasítások (Pentium utáni CPU-k)
 - MMX, SSE

RISC processzorok jellemzői (1):

- **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)
- Például: Beágyazott rendszerek speciális feldolgozó egységei: pl. MCU=mikrokontrollerek, DSP=Digitális jelfeldolgozó processzorok, FPGA-CPLD: Programozható logikai áramkörök beágyazott processzorai,
- További példák: Motorola 88000 RISC rendszere, vagy Berkeley RISC-I rendszere, Alpha, SPARC, MicroChip MCU, ARM, DSP sorozatok, IBM PowerPC stb.

RISC processzorok jellemzői (2):

- **Huzalozott (hardwired) vezérlés és utasításdekódolás** (CU a hardveres dekódolás megvalósításához fix-kombinációs logikát használt, azonban a mai memóriaalapú mikro-kódú gépeknél ez már módosítható).
- **Egyszeres ciklusvégrehajítás:** (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 „látszólagos” párhuzamosítása)**, többszörös adatvonalak, nagyszámú gyors regiszterek alkalmazásával.

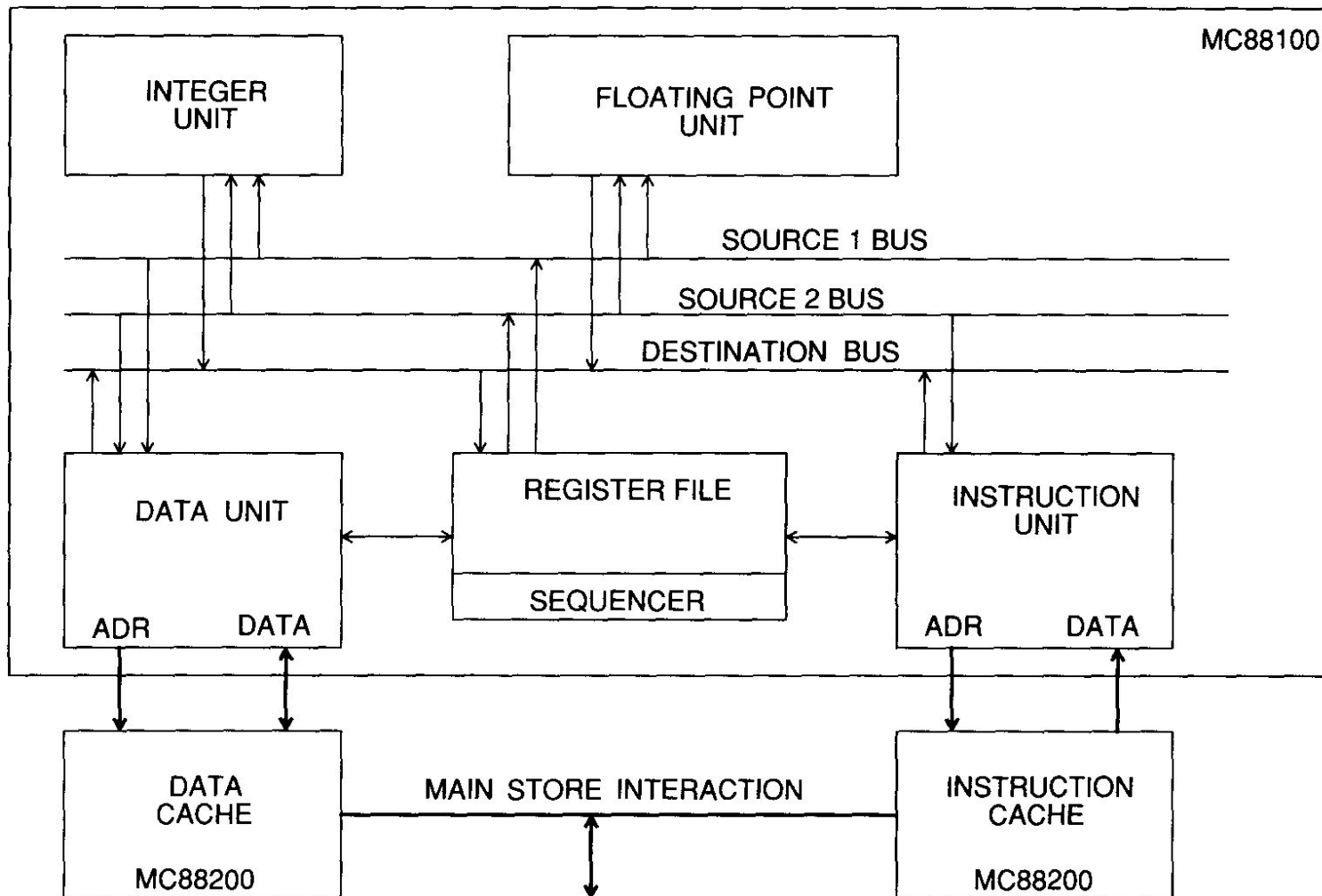
RISC: „Pipe-line” technika

- **Utasítás szintű „látszólagos” 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.)

3 lépcsős pipe-line

	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:

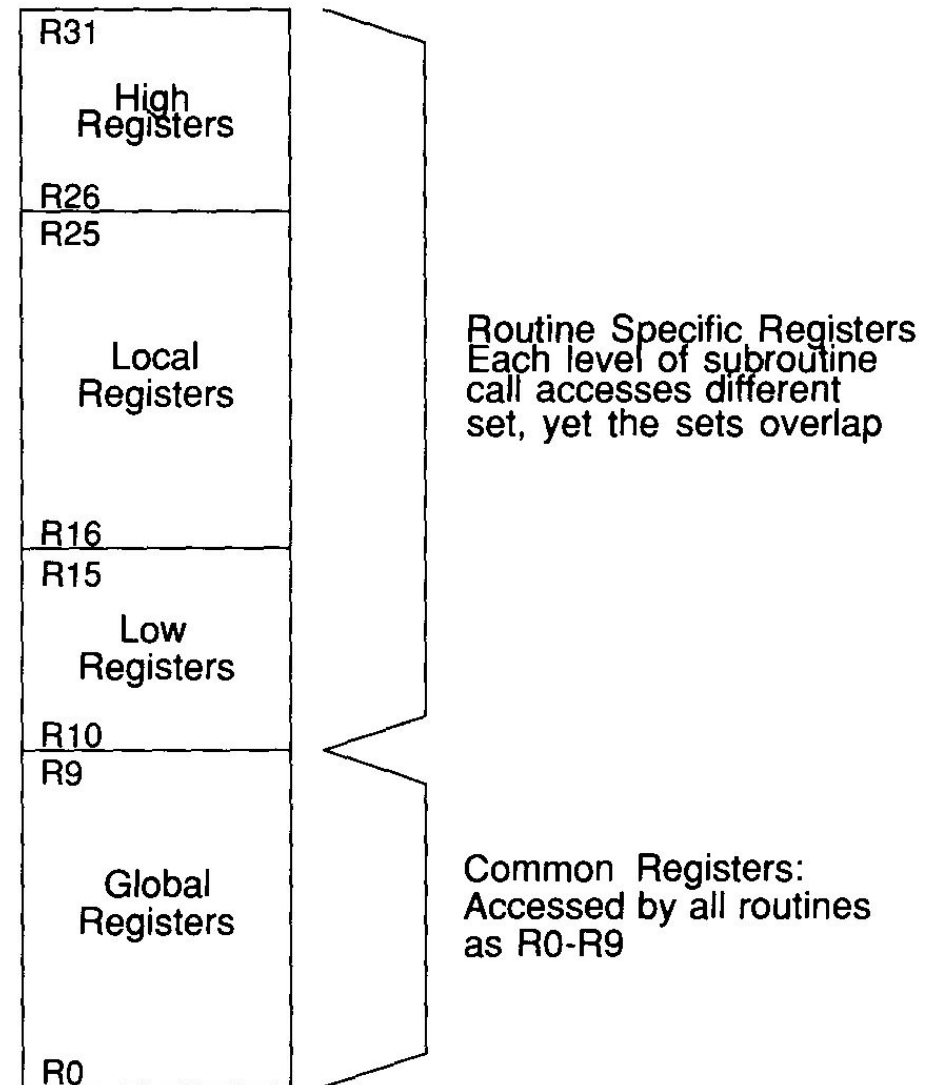
- Buszok
- Cache
- Data / Instruction Unit

- (Harvard Arch!)
- Max 3-című utasítások támogatása

Berkeley RISC-I rendszere:

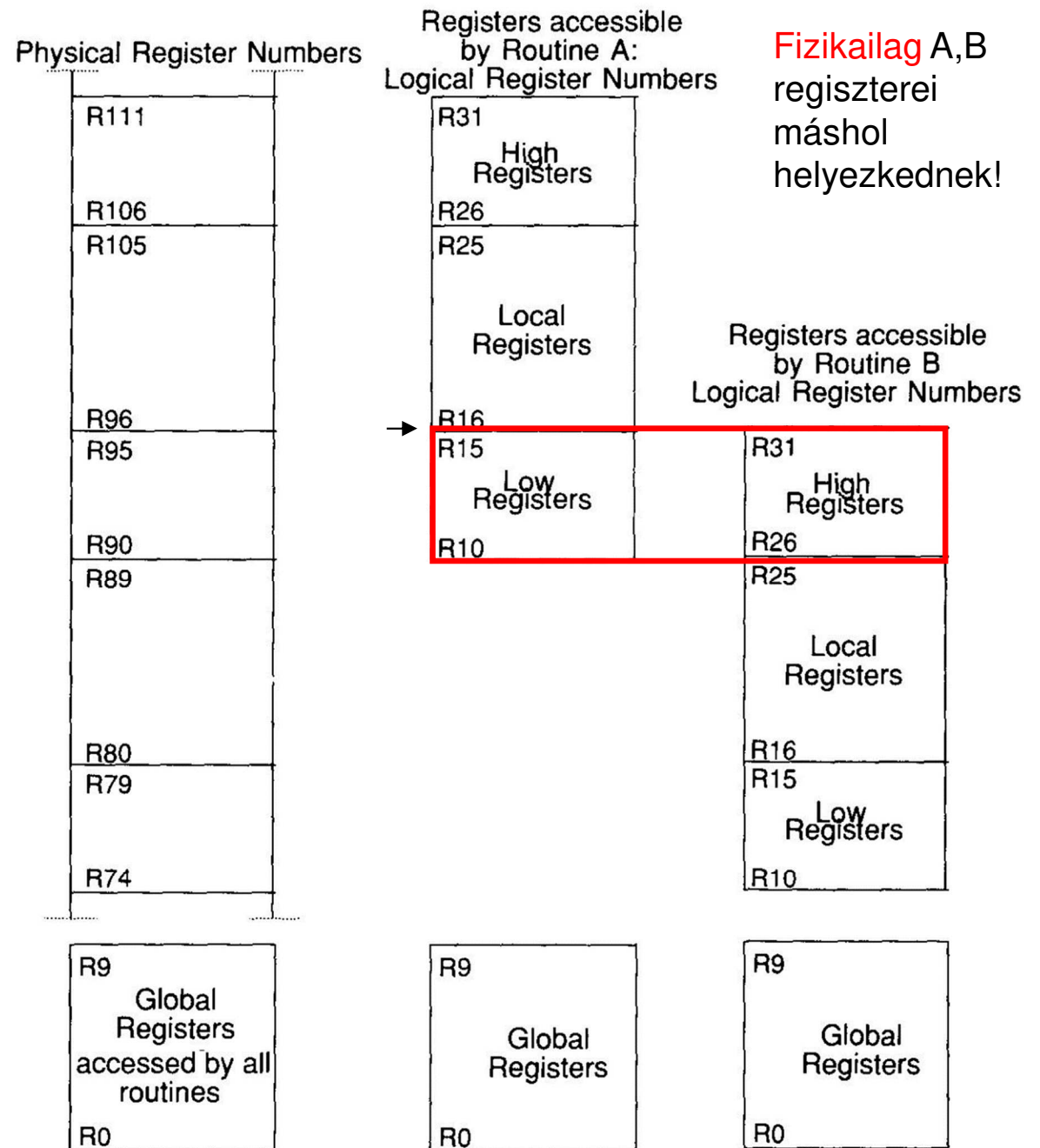
Regiszter használat

- **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.



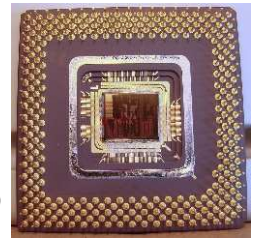
CISC Processzorok jellemzői

- **CISC:** *Complex Instruction Set Computer*
- **Nagyszámú utasítás-típust**, és **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.
- **Lassabb** a változó méretű utasítások **dekódolása**!
- A korai gépeknek egyszerű a felépítése, de bonyolult a nyelvezete. Összetett problémákat megoldása a gépi kódnál magasabb szintű nyelven. **Szemantikus rés**= a gépi 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)

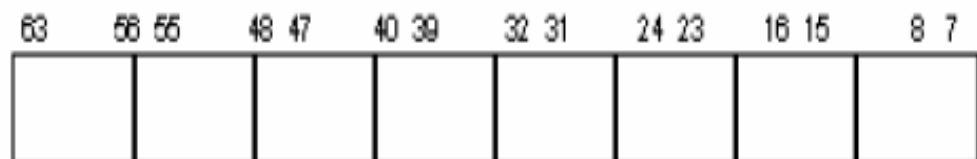
- Közvetlen memória-elérés (**DMA**) és összetett/bonyolult műveletek jellemzők rá.
- **Mikro-programozott** vezérlés (CU)
 - a CISC processzor esetén a fordító (compiler) a programot egyszerűbb szintre fordítja, majd ezután a mikroprogram (ami meglehetősen összetett lehet) veszi át a vezérlést – mikroutasítások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

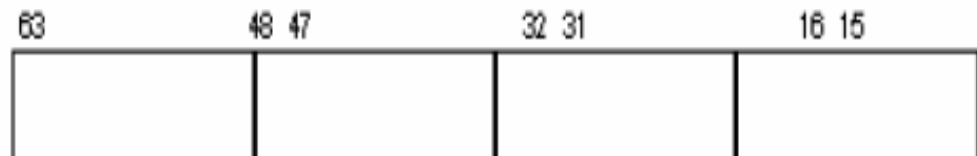


- **MMX: Multi-Media Extension** (Intel Pentium sorozat 1996) –
 - SIMD: Single Instruction / Multiple Data alapú **integer!** stream data feldolgozásra (jelfeldolgozás)
 - 8 db MM0..7 regiszter (8 bit/reg)
 - Regiszterek adatait 4 különböző formátumban lehet tárolni (packet)
 - 57 MMX utasítás, 6 fő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

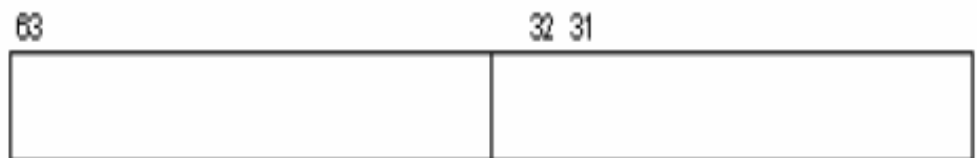
Packed byte (eight 8-bit elements)



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**: Sstreaming SIMD Extension (**lebegőpontos** és **fixpontos** adatfolyam utasításai) //Intel, AMD CPU-k

- 32-bites módban 8 db, egyenként 128-bites regiszter csomag (xmm0...7)

- SSE-1:

- ☐ 128-bit „packed” IEEE *single-precision* floating-point műveletek (~70 utasítás).
- ☐ 2 órajel ciklus alatt számíthatóak

- SSE-2:

- ☐ 128-bit „packed” IEEE *double-precision* SIMD floating-point műveletek (~144 utasítás), vagy
- ☐ 128-bit „packed” egész típusú SIMD műveletek
 - 8-, 16-, 32-, és 64-bites operandusok támogatása
- ☐ 2 órajel ciklus alatt számíthatóak

