Számítógép-architektúrák 2

Jegyzet

Tartalomjegyzék

1. SZÁMÍTÓGÉPEK OSZTÁLYOZÁSA. TELJESÍTŐKÉPESSÉG ÉS SZOLGÁLTATÁSBIZTONSÁG.

1.1. A teljesítőképesség meghatározása

Egy **számítógép architektúrája** alatt az utasításkészletének felépítését, a számítógép szervezését és a hardveres megvalósítását értjük. A számítógép-tervezők feladata, hogy olyan architektúrát hozzanak létre, amely eleget tesz a funkcionális követelményeknek, de ugyanakkor megfelel a teljesítőképesség, megbízhatóság, fogyasztás és ár célkitűzéseinek is.

A felhasználási terület függvényében, a különböző számítógép-típusokkal szemben állított követelmények fontossága különböző lehet:

- Az általános célú, otthoni **személyi számítógépek** (personal computers, PC) esetében fontos a minél alacsonyabb ár/teljesítőképesség arány, amit olcsóbb processzorokkal, de fejlett grafikai, videó és audió lehetőségekkel érnek el.
- A tudományos és tervezői tevékenységben használt asztali **munkaállomások** (workstations) esetében fontos a nagysebességű lebegőpontos feldolgozás és a grafika.
- A kereskedelmi **szerverek** (servers) esetében elsődleges az időegység alatt feldolgozott kérések száma, de – a folyamatos igénybevétel miatt – kritikus a magas fokú szolgáltatásbiztonság is. Mivel gyakoriak a szolgáltatások növelésére irányuló igények, ezeknél a gépeknél lehetővé kell tenni a skálázhatóságot, azaz a feldolgozási és tárolókapacitás, valamint az I/O sávszélesség növelését.

- A mindinkább elterjedőben lévő, nem számítástechnikai célú termékekbe **beágyazott számítógépek** (embedded computers) esetében fontos a valós idejű működés biztosítása, de minél alacsonyabb áron, ezért nem célszerű a szükségesnél nagyobb teljesítőképességű processzor és nagyobb méretű memória alkalmazása. Ugyanakkor nem elhanyagolható a minél alacsonyabb fogyasztás megvalósítása sem.
- A nagy teljesítőképességű **mainframe**-ekre és **szuperszámítógépekre** (supercomputers) jellemző az időegység alatt minél nagyobb mennyiségű – első sorban lebegőpontos – utasítás feldolgozása, amit manapság multiprocesszoros architektúrával, vagy asztali számítógépek klaszterekbe kapcsolásával biztosítanak.

A számítógépek teljesítőképességének (performance) mérésénél két mennyiség jöhet számításba:

- A **végrehajtási idő** (execution time) – más néven ***válaszidő*** (response time) –, ami egy feladat kezdete és befejezése között eltelt időt jelenti. Ez lehet egy programnak a végrehajtási ideje (másodpercben), vagy egy lemezolvasás ideje (miliszekundumban). A minél kisebb végrehajtási idő az asztali számítógépes-alkalmazások esetében fontos a felhasználónak.
- Az **átbocsátóképesség** (throughput) – más néven ***sávszélesség*** (bandwidth) –, ami alatt egy bizonyos idő alatt elvégzett munkát értjük. Ez lehet például egy szerver által másodpercenként feldolgozott tranzakciók száma (tranzaction/s),
- vagy egy lemezegységnél a másodpercenként átvitt megabájtok száma (MB/s). Egy szerver adminisztrátora a gép minél magasabb átbocsátóképességében érdekelt.

A közelmúlt fejlődését figyelemmel követve azt tapasztaljuk, hogy a sávszélesség növekedése a végrehajtási idő négyzetének növekedésével arányos.

Általában egy A számítógép teljesítőképességét egy másik B gép teljesítőképességével szokás összehasonlítani. Elmondhatjuk, hogy „A gép teljesítőképessége n-szer nagyobb, mint B gépé”, ha a végrehajtási időt figyelembe véve

$$n = \frac{\text{Teljesítőképesség}_A}{\text{Teljesítőképesség}_B} = \frac{\frac{1}{\text{Végrehajtási idő}_A}}{\frac{1}{\text{Végrehajtási idő}_B}} = \frac{\text{Végrehajtási idő}_B}{\text{Végrehajtási idő}_A}$$

illetve, az átbocsátóképességet figyelembe véve

$$n = \frac{\text{Teljesítőképesség}_A}{\text{Teljesítőképesség}_B} = \frac{\text{Átbocsátóképesség}_A}{\text{Átbocsátóképesség}_B}$$

A terminológiával kapcsolatban megjegyezzük, hogy a következő kijelentések ekvivalensek:

- „A gép n-szer gyorsabb, mint B” (például 3-szor gyorsabb)
- „A gép 100(n–1)%-kal gyorsabb, mint B” (például 200%-kal gyorsabb)

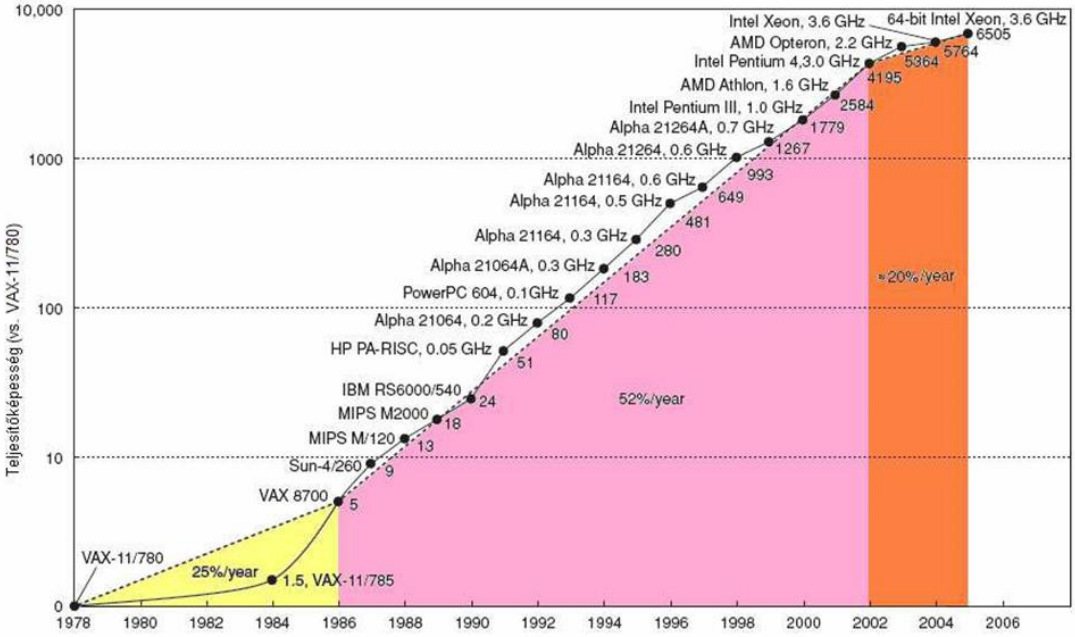
1.2 A teljesítőképesség mérése

A számítógépek teljesítőképességét benchmark programokkal tudjuk mérni. Ezek olyan programok, amelyeket úgy írtak meg, hogy megpróbálják a valós alkalmazások profilját és viselkedését utánozni. Hogy minél több fajta (például fixpontos, lebegőpontos, grafikus) alkalmazást fedjenek le, a benchmark programokat gyűjteményekben szokták forgalmazni. Íme néhány ismertebb benchmarkgyűjtemény:

- SPEC CPU 2006 (Standard Performance Evaluation Corporation) – CPU- teljesítőképességet mérő programok asztali gépek számára. A programok a CPU-időt mérik, igyekezve minimálissá tenni az I/O műveletek okozta időveszteséget.
- SPECWeb – több kliens weboldalak iránti kérését szimulálva méri a szerver átbocsátóképességét.

- TPC (Transaction Processing Council) – a szerver tranzakció-feldolgozó képességét méri, adatbázis-hozzáféréseket és aktualizálásokat szimulálva.
- EEMBC (EDN Embedded Microprocessor Benchmark Consortium) – különböző beágyazott alkalmazás (autóipari, távközlési, irodaautomatizálási stb.) teljesítőképességét méri.

A következő ábra a SPECint (fixpontos, azaz integer) benchmarkkal mért számítógép-teljesítőképességek időbeli alakulását mutatja be, a DEC VAX-11/780 géphez viszonyítva.



Az 1986–2002-es időszakban évi 50%-osnál nagyobb teljesítőképesség-növekedés figyelhető meg, ami nagyrészt a számítógépek architekturális és szervezési fejlesztéseknek (RISC utasításkészlet, párhuzamosítás, cache) tudható be.

A számítógépek teljesítőképességének kifejezésére hagyományosan szokás a processzor által időegység alatt feldolgozott utasítások számát használni – ez lényegében a processzor átbocsátóképességét fejezi ki (utasítás/s). A gyakorlatban használatos mértékegységek:

- **MIPS** (Millions of Instructions per Second) – a másodpercenként végrehajtott millió utasítás száma, azaz

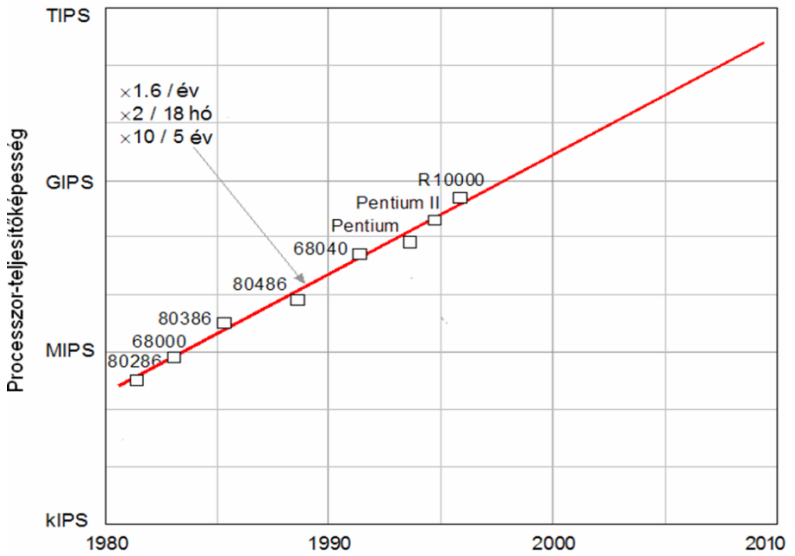
$$\text{MIPS} = \frac{\text{A program utasításainak száma}}{\text{A program végrehajtási ideje} \times 10^6}$$

- **MFLOPS** (Millions of Floating Point Operations per Second) – a másodpercenként végrehajtott millió lebegőpontos utasítás száma, azaz

$$\text{MFLOPS} = \frac{\text{A program lebegőpontos utasításainak száma}}{\text{A program végrehajtási ideje} \times 10^6}$$

(Természetesen a MFLOPS mérésére lebegőpontos profilú benchmarkra van szükség.)

A mai nagy teljesítőképességű gépek esetében szokás a mutatók többszörösét (giga, terra) használni, akkor fennáll a GIPS=1000xMIPS, illetve TIPS=1000xMIPS. Az alábbi ábra a processzorok ilyen jellegű teljesítőképességének az alakulását mutatja be az utóbbi évtizedekben. Látható, hogy a teljesítőképesség kb. ötévenként folyamatosan megtízszereződik, aminek architekturális, szervezési és technológiai téren tapasztalt haladásra visszavezethető okai vannak.



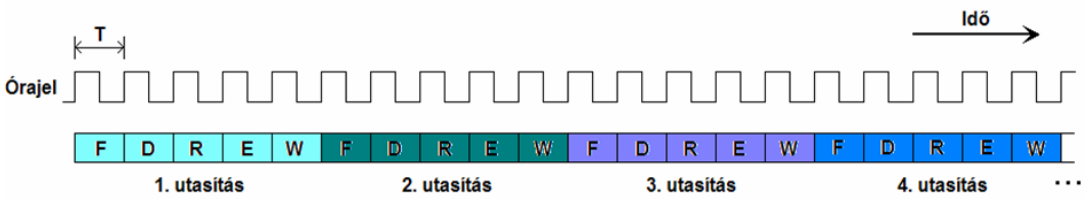
1.3. A processzor teljesítőképesség-egyenlete

A processzornak ciklikus működése van: egy program feldolgozása a benne lévő utasítások egymás utáni végrehajtását feltételezi, ami időben utasítás-ciklusok sorozatát eredményezi. Feltételezzük, hogy mindegyik utasítás-ciklus a következő öt **alciklusokból** (gépi ciklusokból) áll:

- F (***fetch***) – az utasítás lehívása a memóriából
- D (***decode***) – az utasítás dekódolása, értelmezése
- R (***read***) – az operandus kiolvasása
- E (***execute***) – az utasításban megadott művelet végrehajtása
- W (***write***) – az eredmény beírása a megadott helyre

Egy-egy utasítás feldolgozása elemi lépések sorozatát jelenti, amelyeknek a végrehajtása időben az ***órajel*** (clock) ütemeinek felel meg. (Az órajel, mint ismeretes, egy periodikus jel, amelynek periódusa a frekvencia fordítottja: T= 1/f. Például egy 1 GHz-es frekvenciának 1 ns-os órajel-periódus felel meg.) A

hagyományos, Neumann-elvű számítógép processzorának időbeli tevékenységét az alábbi ábra szerint szemléltethetjük:



(Meg kell jegyezni, hogy az egyszerűség kedvéért feltételeztük, hogy minden gépi ciklus egy órajelütem alatt elvégezhető, ami nem mindig így van.)

Az ábra alapján egyszerűen felírható a processzor ún. **teljesítőképesség-egyenlete**, ami egy program végrehajtási idejét (CPU-idő) fejezi ki:

$$\text{Program végrehajtási ideje} =$$

$$= \textit{Utasításszám} \times \textit{Órajelciklus per utasítás} \times \textit{Órajelciklus ideje}$$

Természetesen a teljesítőképesség annál nagyobb, minél kisebb a végrehajtás

$$\text{Teljesítőképesség} = \frac{1}{\text{Végrehajtási idő}}.$$

Az órajelciklus per utasítást **CPI**-nek szokás rövidíteni (*Clock Cycles per Instruction*). A rövidítéseket használva, egyenletünk felírható így is:

$$\text{Program végrehajtási ideje} = \text{Utasításszám} \times \text{CPI} \times \text{T}$$

Az ábrán lévő példa esetében CPI=5. Feltételezve, hogy az alkalmazásunk végrehajtása 10 000 utasítást jelent, és az órajel 2 GHz-es, a végrehajtáshoz szükséges processzoridő 25 μs lenne.

A képletben szereplő utasításszám nem a programban lévő utasításoknak a statikus számát jelenti, hanem – a programciklusokat, elágazásokat is figyelembe véve –, a program futása alatt valóban végrehajtott, dinamikus utasításszámot.

Mivel általában nem minden utasítás végrehajtása azonos számú elemi lépésből áll, a CPI értéke is egy átlagot fejez ki:

$$\text{CPI} = \frac{\text{A program összes órajelciklusa}}{\text{Utasításszám}}$$

Figyelembe véve, hogy a programban egy-egy utasítás többször is megtalálható, illetve a végrehajtás közben a ciklusok miatt többször is sorra kerülhet, a program összórajelciklusa kifejezhető mint

$$\text{A program összes órajelciklusa} = \sum_{i=1}^{n} U_i \times \text{CPI}_i$$

ahol n a program különböző utasításainak a száma, Ui az i utasítás sorra kerüléseinek a száma, CPIi pedig az i utasítás órajelciklusainak a száma. Ezzel a CPI kiszámítására a következő képletet kapjuk

$$\text{CPI} = \sum_{i=1}^{n} \frac{U_i}{\text{Utasításszám}} \times \text{CPI}_i$$

Az egyenletben lévő szorzatnak az első tényezője az i utasítás előfordulásának az arányát fejezi ki a program végrehajtása során.

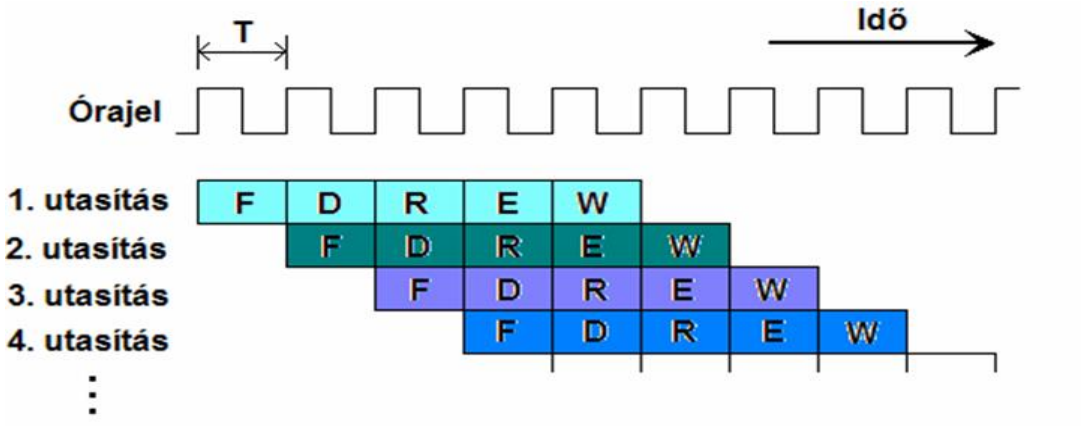
A processzor teljesítőképesség-egyenletéből levonhatjuk a következtetést, hogy a processzoridőben kifejezett teljesítőképesség egyformán függ három tényezőtől, amelyeket csökkenteni kell a teljesítőképesség növelése érdekében:

- **Utasításszám** – a hatékony fordítóprogramtól és az utasításkészlet architektúrájától (pl. CISC, RISC) függ
- **Ciklusszám per utasítás** (CPI) – függ a processzor szervezésétől és az utasításkészlet architektúrájától
- **Órajelciklus ideje** (T) – ennek csökkentése a technológiai fejlődés függvénye

A tervező által kialakítandó számítógép-architektúra szempontjából a CPI csökkentésére kell törekedni. Ennek egyik lehetősége az egyszerű utasításokból (RISC) álló utasításkészlet kidolgozása, amelyeket kevés elemi lépésből végre lehet hajtani – ez viszont másrészt a gépi kódú program utasításainak a számát növelni fogja. A másik lehetőség a tevékenységek párhuzamosítása, amelyet különböző szinten lehet megvalósítani:

- *Processzorszintű párhuzamosítás* – ide tartozik a pipeline feldolgozásmód, a szuperskalár, valamint a VLIW-szervezésű processzorok (lásd a Számítógép- architektúrák tárgy I. részét). Mint határesetet, ide sorolhatjuk a mai processzorokban található, SIMD típusú utasítások végrehajtására szakosodott egységet is, amely képes több különböző adaton (vektorelemen) azonos műveletet párhuzamosan végrehajtani.
- *Rendszerszintű párhuzamosítás* – ez a technika több processzor beiktatását jelenti (MIMD típusú multiprocesszoros rendszer), amelyek együttműködve hajtják végre az alkalmazás feladatait.

A **pipeline** szervezésű processzorokra az utasításszintű átlapolt feldolgozásmód jellemző, ami az egész utasítássorozat végrehajtási idejének a csökkenéséhez vezet. Az előző soros utasítás-feldolgozást bemutató ábra pipeline alkalmazása esetén a következőképpen módosul:



Az ábrán látható, hogy a processzor minden órajelütemben befejez (kibocsát) egy utasítást, tehát a CPI=1. Ez ötszörös gyorsulást jelent a soros feldolgozáshoz képest, így a feltételezett példa esetében a végrehajtási idő 5 µs-ra csökken.

1.4. Amdahl törvénye

A párhuzamosítás eredményeként valamilyen teljesítőképesség-növekedést várunk el, amit **gyorsításban** (*speedup*) fejezünk ki. Ez a feljavított számítógép által nyújtott teljesítőképességet viszonyítja az eredeti gép teljesítőképességéhez, ugyanazon feladat végrehajtásánál:

$$\text{Gyorsítás} = \frac{\text{Teljesítőképesség a feljavított géppel}}{\text{Teljesítőképesség a feljavításnélkül}}$$

A kérdés az, hogy mennyi lesz ez a teljes gyorsítás, ha a párhuzamosítás által létrehozott gyorsítás a feladat-végrehajtási folyamatnak csak egy töredékét érinti? Erre a kérdésre adja meg a választ **Amdahl törvénye**:

A gyorsabb végrehajtási mód által okozott teljesítőképesség-növekedést korlátozza az az időtöredék, ameddig a gyorsabb módot használni tudjuk. Képletben fogalmazva, az egész feladat végrehajtására vonatkozó teljes gyorsítást a következőképpen számíthatjuk ki:

$$\text{Gyorsítás}_{\text{teljes}} = \frac{1}{\left(1 - \text{Töredék}_{\text{javított}}\right) + \frac{\text{Töredék}_{\text{javított}}}{\text{Gyorsítás}_{\text{javított}}}}$$

Ez azt jelenti, hogy például ha a feldolgozás 5%-át **nem tudjuk párhuzamosítani**, az elérhető gyorsítás nem lehet nagyobb, mint 20, még ha a fennmaradó 95%-ban végtelen gyorsítást is érnénk el.

A képletben a $\text{Töredék}_{\text{javított}}$ azt az időt jelenti az összidőhöz viszonyítva, amelyre a feljavítás vonatkozik. Például, ha az eredeti gépen a végrehajtás 8

másodpercet vesz igénybe, és ebből 6 másodpercnyi részt tudunk felgyorsítani, akkor a töredék értéke 6/8=0,75 lesz. A $\text{Gyorsítás}_{\text{javított}}$ a feljavított részre vonatkozó gyorsítást jelenti, vagyis az eredeti végrehajtási idő arányát a gyorsabb végrehajtási módban elérhető időhöz. Az előző példánál maradva, ha a felgyorsított rész az új gépen 2 másodperc alatt fut le, akkor a gyorsítás mértéke 6/2=3. Végeredményben az egész feladatra elért teljes gyorsítás:

$$\text{Gyorsítás}_{\text{teljes}} = \frac{1}{0{,}25 + \frac{0{,}75}{3}} = \frac{1}{0{,}5} = 2$$

Belátható, hogy az eredeti 8 másodperc helyett, a felgyorsított gépen a feldolgozás 4 másodpercig fog tartani, tehát a teljes gyorsítás 2 – amennyit kiszámítottunk.

Amdahl törvényét megfogalmazhatjuk egy általánosított alakban is. Tegyük fel, hogy egy alkalmazás végrehajtása m szakaszból áll, amelyeknek az fi aránya ismert az egész végrehajtási időből:

$$f1 + f2 + \ldots + fm = 1$$

Belátható, hogy az eredeti 8 másodperc helyett a felgyorsított gépen a feldolgozás 4 másodpercig fog tartani, tehát a teljes gyorsítás 2 – amennyit kiszámítottunk.

Ha mindegyik i szakaszt sikerül felgyorsítani p_i tényezővel, akkor a teljes gyorsítást a következő képlet adja meg:

$$\text{Gyorsítás}_{\text{teljes}} = \frac{1}{\frac{f_1}{p_1} + \frac{f_2}{p_2} + \ldots + \frac{f_m}{p_m}}$$

1.5. Multiprocesszoros rendszerek teljesítőképessége

A több processzorból álló számítógép-rendszerek teljesítőképességének a kiértékelése komplex feladat, mivel különböző alkalmazások más-más lehetőséget nyújtanak a párhuzamosításra. Ahhoz, hogy a feladatot gyorsabban tudjuk elvégezni, mint az egyprocesszoros gépen, szükséges a programot olyan részfeladatokra bontani, amelyeket szét lehet osztani a processzorok között, majd egyszerre végrehajtani. Feltételezve, hogy a rendszerben n processzor van, a gyorsítást (Sn) a következőképpen fejezhetjük ki:

$$S_n = \frac{\text{Végrehajtási idő az egyprocesszoros rendszeren}}{\text{Végrehajtási idő a multiprocesszoros rendszeren}}$$

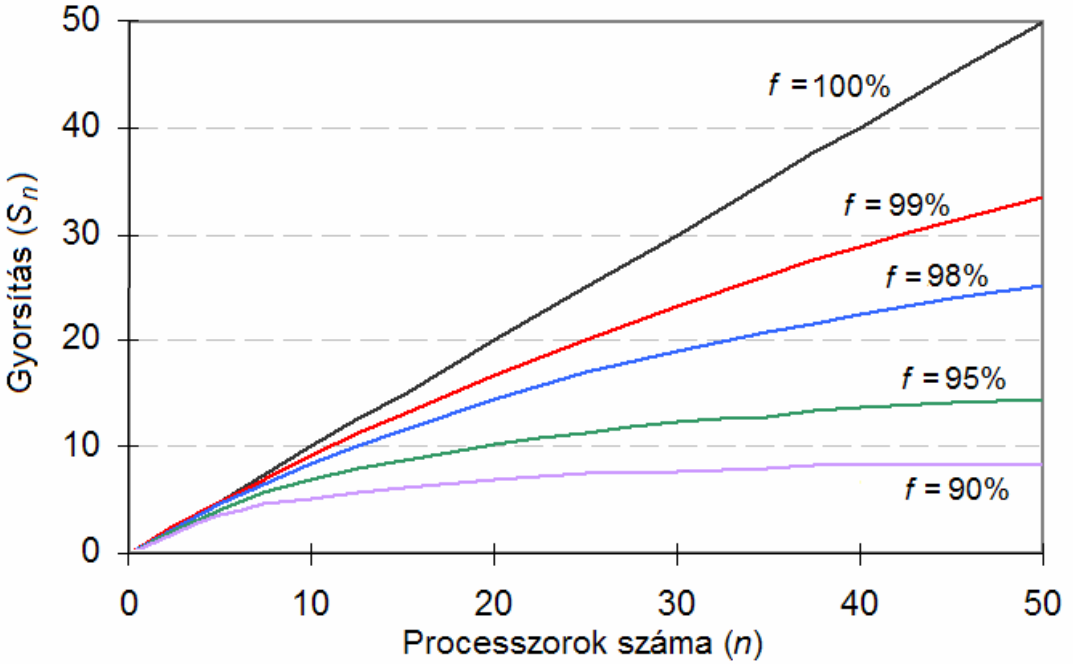
Az ideális esetet az jelentené, ha a programot pontosan n, egyforma végrehajtási idővel rendelkező részre lehet felbontani. Ekkor kapjuk a maximális gyorsítást, amely

$$S_n = \frac{t}{t/n} = n$$

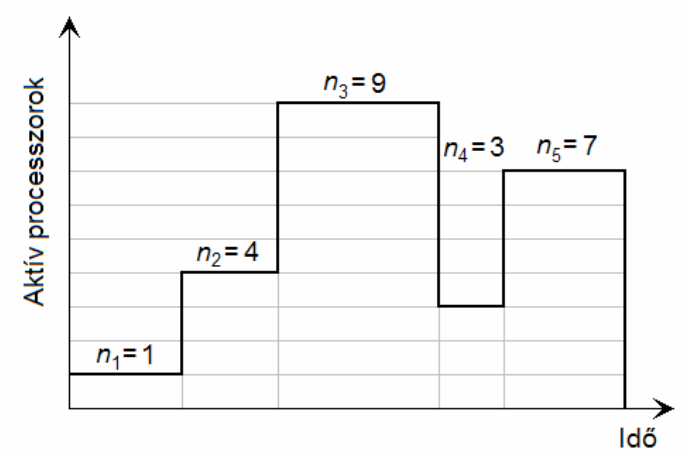
ahol t az egyprocesszoros rendszeren való végrehajtási idő. Ha viszont a számításoknak csak egy részét (legyen ennek az aránya f) lehet párhuzamos feladatokra bontani, akkor az egész felhasználási idő $(1-f) \times t + f \times t/n$ értékre csökken, és a gyorsítási tényezőt az alábbi képlet adja meg:

$$S_n = \frac{t}{(1-f)t + f\frac{t}{n}} = \frac{n}{n(1-f)+f}$$

Ez nem más, mint Amdahl törvénye, amelyben a párhuzamosítás ideje alatt a gyorsítási tényező n. Az alábbi ábra S_n grafikonját mutatja a processzorok számának függvényében. Látható, hogy a multiprocesszoros rendszer alkalmazása a feladat végrehajtásának gyorsítását eredményezi, de hogy ez jelentős legyen, szükséges a párhuzamosítható rész arányának nagyon magas értéke.



A gyakorlatban viszont a párhuzamos feldolgozás időben változó számú processzor igénybevételével történik, amint azt az alábbi ábra mutatja.



Ebben az esetben nincs minden processzor folyamatosan leterhelve, egyesek dolgoznak, mások nem. A gyorsítás kiszámítására Amdahl törvényének általánosított alakját használhatjuk:

A képletben n_i az i időintervallumban aktív processzorok számát jelenti, f_i pedig az i intervallum arányát az összidőből.

A valóságban az eddigi elméleti számításoknál rosszabb a helyzet, ugyanis az együttműködő processzoroknak kommunikálniuk kell egymással, ami további időráfordítást jelent. Ez a többletidő nem csak a kommunikációs hálózat és az operációs rendszer kommunikációs szoftvere által okozott késleltetésétől függ, hanem magától az alkalmazói programtól is. Ha a program részfeladatait sikerült úgy elosztani a processzorok között, hogy ritkán van szükségük részeredményeket közölni egymásnak, akkor a kommunikációs többletidő kicsi lesz, ellenkező esetben viszont jelentősen megnőhet. Általában feltételezhetjük, hogy a kommunikációs ráfordítás aránya függ a kommunikációs hálózat komplexitásától, és így a processzorok számának valamilyen függvénye, mondjuk *c(n)*. Visszatérve a párhuzamosítás idejére egyenletes munkamegosztást feltételező esetre, a kommunikációra fordított többletidő c(n) × f × t/n lesz. A gyorsítást a következő egyenlettel számíthatjuk ki:

$$S_n = \frac{t}{(1-f)t + f\frac{t}{n} + c(n)f\frac{t}{n}} = \frac{n}{n(1-f) + f + fc(n)}$$

Ideális esetben, ha az egész feldolgozás párhuzamosítható (f=1), valamint a processzorok közötti kommunikációk teljesen átfedhetik egymást, az előbbi egyenletből azt kapjuk, hogy:

$$S_n = \frac{n}{1 + c(n)}$$

A multiprocesszoros rendszer fontos feladata – amint láttuk – a számítások gyorsítása a párhuzamos végrehajtás által. A rendszer **hatékonyságát** (*efficiency*) a következőképpen határozhatjuk meg:

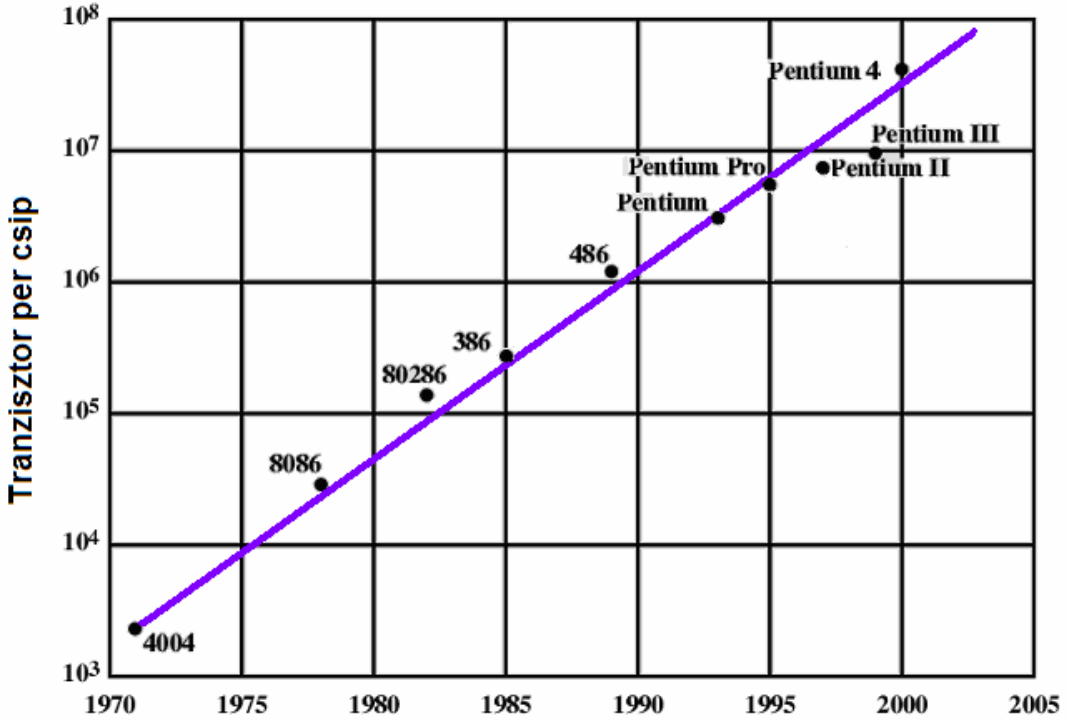
$$E = \frac{S_n}{n} \times 100\%$$

A maximális, 100%-os hatékonyság akkor érhető el, ha n-szeres a gyorsítás, de ez az ideális eset. A gyakorlatban E<1, és a rendszertervezőnek a feladata, hogy a feladatok jó elosztásával a processzorok között, valamint megfelelő multiprocesszoros architektúra kiválasztásával, a hatékonyság minél inkább megközelítse az egységnyi értéket. Mindenképpen el kell kerülni azt az esetet, hogy a kommunikációs ráfordítás domináljon, mert akkor megtörténhet, hogy az egyprocesszoros rendszer hatékonyabb lesz, mint a több processzoros. A rendszer hatékonyságát kedvezően befolyásolja a processzorok minél egyenletesebb terhelése.

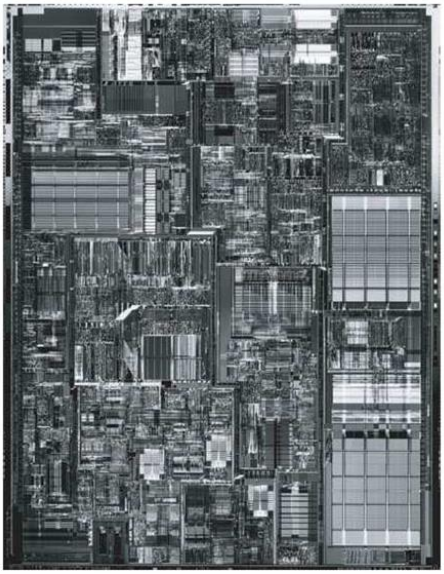
1.6. A technológia szerepe a teljesítőképességben

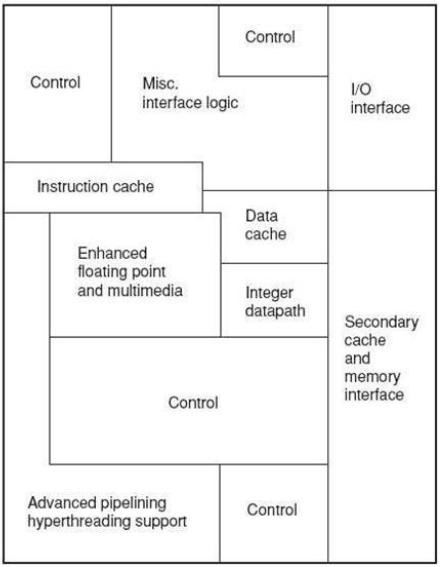
Az integrált áramkörök gyártási technológiájának meghatározó szerepe van ezek sebességében, méretében, fogyasztásában és árában. Egy processzor gyártójának választania kell a gyorsabb, de kevésbé sűrűn integrálható és nagyobb teljesítményt felvevő bipoláris tranzisztor alapú technológia (ECL vagy TTL), illetve a lassúbb, de sűrűbben integrálható és kevesebbet fogyasztó unipoláris tranzisztorokat használó technológia (MOS vagy CMOS) között. A legkisebb áramfelvétele a CMOS tranzisztornak van, mivel ez lényegében egy komplementáris tranzisztorpárból áll, amelyből az egyik mindig zárva van. A mai gyakorlatban a processzorgyártók a két technológia kombinációját, az ún. BiCMOS technológiát alkalmazzák, amelyben a nagyobb sebességet igénylő belső részegységeket bipoláris tranzisztorokból, a többit pedig CMOS-ból állítják elő.

Az integrált áramkörbe lévő tranzisztorok számának növekedési ütemét figyelve fogalmazódott meg 1965-ben a **Moore-féle szabály**: ***a csipbe integrálható tranzisztorok száma nagyjából évente megduplázódik, amit manapság inkább úgy érvényesítenek, hogy három évenként megkétszereződik***. Ezt a tendenciát követhetjük figyelemmel az Intel processzorok esetében az alábbi ábrán:

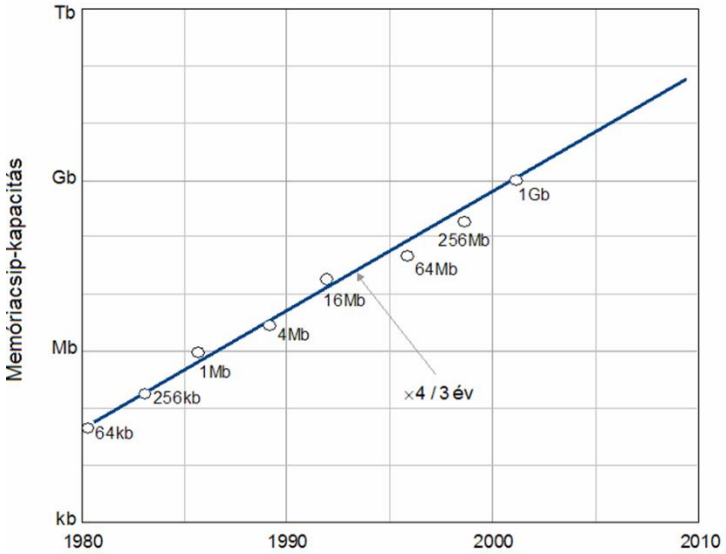


Manapság már ULSI (*Ultra Large Scale Integration*) áramkörökről beszélünk, amelyekben az integráltság foka elérte a 100 millió tranzisztort. A magas integráltság azért fontos, mert így egyetlen csipbe sok architekturális egységet lehet beépíteni, amire szükség van a hyperpipeline-os szuperskalár, esetleg többszálas vagy kétmagos processzoroknál, valamint a 64 bites feldolgozásnál. Például a Pentium 4 Extreme Edition processzor 178 millió tranzisztor tartalmaz. A következő ábra egy Pentium 4-es processzor integrált áramkörét, valamint a belső egységek elhelyezkedését mutatja:





Moore szabályának megfelelően folyamatos növekedést észlelhetünk a DRAM memóriacsipek kapacitásának esetében is (lásd az alábbi ábrát). Ez ugyancsak fontos mutatója egy számítógép teljesítőképességének, és első sorban a technológiai fejlődés eredménye. Az utóbbi években a kapacitás-növekedés lassulni látszik, kb. kétévente duplázódik meg.



Íme, a korszerű integrált áramkör-technológia néhány fontos jellemzője:

- **Sűrűség**. A csipben lévő tranzisztorok sűrűsége folyamatosan növekszik, mivel a gyártási technológia lehetővé tette a tranzisztorok és vezetékek méretének csökkentését. Egy tengely irányában kifejezve: 130 nm (2002) → 90 nm (2006) → 65 nm (2007) → 22 nm.
- **Sebesség**. A pipeline fokozatainak a késleltetése korlátozza a maximális órajel-frekvenciát. Ez a késleltetés függ egyrészt a logikai kapukat alkotó tranzisztorok kapcsolási sebességétől, másrészt az összekötő huzalok okozta késleltetéstől. A tranzisztorok szempontjából kedvezően hat a sebességre a kisebb méret, de ez nem mondható el a huzalokról is, az egységnyi hosszra eső ellenállás és kapacitás miatt. Ennek következményeként, az órajelciklus mind nagyobb részét kell felhasználni a jelek huzalokon való terjedése okozta késleltetések lefedésére. Az ilyen késleltetések kiegyenlítésére például a Pentium 4- be két plusz fokozatot kellett beiktatni.
- **Fogyasztás**. A tápegységtől felvett villamos teljesítmény kibocsátott hő formájában távozik az integrált áramkörből, és ezt el kell vezetni (ventilátorok és hűtőbordák segítségével). A teljesítmény-felvétel egyenesen arányos az áramkörben lévő tranzisztorok számával és ezek kapcsolási frekvenciájával. A CMOS csipekben a fogyasztás elsődleges forrása a dinamikus teljesítmény, amelyet egy tranzisztornál a következő egyenlet ad meg:

$$\text{Teljesítmény}_{\text{dinamikus}} = \text{Kapacitív terhelés} \times \text{Feszültség}^2 \times \text{Kapcsolási frekvencia}$$

Egy tranzisztor kapacitív terhelése a kimenetére kötött tranzisztorok számának, a tranzisztorok kapacitásának és a huzalok kapacitásának a függvénye. Az első tényezőt a tervrajz, az utóbbi kettőt az alkalmazott technológia határozza meg. A dinamikus teljesítményt a tápfeszültség csökkentésével lehet jelentősen redukálni, ezért ezt az értéket az utóbbi évtizedekben 5 V-ról 1,5 V-ra csökkentették. A kapcsolási frekvencia jelentős tényezővé lépett elő napjainkban a magas órajellel dolgozó processzoroknál. Például egy 3,2 GHz-es Pentium 4 Extreme Edition processzor 135 W-ot fogyaszt, ennek megfelelő hőáramot kell elvezetni a kb. 2 cm^2-es

csip felületéről, ennek túlmelegedése elkerülésével. Mivel a túlmelegedés veszélye kritikussá vált, a legtöbb processzorban leállítják az éppen inaktív belső egységeknek az órajelét.
A fogyasztás másik összetevője a statikus teljesítmény, amelyet a tranzisztorok zárt állapotában is elszivárgó áram okoz:

$$\text{Teljesítmény}_{\text{statikus}} = \text{Áram}_{\text{elszivárgó}} \times \text{Feszültség}$$

Nagyszámú tranzisztort tartalmazó processzorok esetében az elszivárgó áram olyan jelentős, hogy ez felel a fogyasztás egyharmadáért (2007-es adat).

A jövőben az órajel-frekvencia növekedésének a lassulása várható egyrészt a hőkibocsátás korlátai miatt, másrészt, mert a pipeline fokozataira manapság is kevés részfeladat jut, ennek működését nehezen lehet tovább gyorsítani. Egy megoldást jelenthet az egy csipbe beépített több processzormag, amelyek alacsonyabb feszültségen és frekvencián dolgoznak, így csökkentve a kibocsátott hőt.

1.7. A számítógépek szolgáltatásbiztonsága

A szolgáltatásbiztonság (**dependability**) egy gyűjtőfogalom, amely a számítógép által nyújtott **szolgáltatás iránti bizalmat** fejezi ki. A rendszer két állapotban lehet:

– **Teljesíti** a szolgáltatást a megadott specifikációnak megfelelően

– **Megszakítja** a szolgáltatást, eltérve a specifikációtól

A szolgáltatás megszakítása a számítógép **meghibásodásának** (*failure*) a következménye, aminek az oka egy alkotóelemben létrejött **hiba** (*fault*). A hibás alkotóelem lehet hardver (például egy memóriacsip), vagy szoftver (például a programozó által tévesen beiktatott utasítás). A hiba latens módon (latent error) tartózkodik a gépben, mindaddig, amíg nem aktiválódik, vagyis az általa okozott hibás adat vagy utasítás nem kerül felhasználásra. Amikor ez oda

vezet, hogy a számítógép külső viselkedése eltér az elvárttól, akkor beszélünk a gép meghibásodásáról. A hiba kiiktatásával, kijavításával érjük el a számítógép helyreállítását.

Figyelembe véve a két alapállapot, a szolgáltatásbiztonságot több számszerű mennyiséggel jellemezhetjük:

- A megbízhatóság (reliability) annak a valószínűsége, hogy egy adott időtartam alatt a számítógép ne hibásodjon meg. A gyakorlatban az átlagos meghibásodási időt (MTTF – Mean Time To Failure) szokták használni:

$$\mathrm{MTTF} = \int_0^t R(t)dt$$

A képletben R(t) jelöli a megbízhatóságot, amely mindig $0 \le R(t) \le 1$, t pedig az időt. Az MTTF fordítottja a meghibásodási ráta (failure rate)

$$\lambda = \frac{1}{\mathrm{MTTF}}$$

amit meghibásodás/óra mértékegységben fejezünk ki. Több elektronikai alkotóelemből álló, nem redundáns rendszer esetében az egész rendszernek a meghibásodási rátája az alkotóelemek meghibásodási rátájának az összege.

- A helyreállíthatóság (maintainability) annak a valószínűsége, hogy egy meghibásodott számítógépet adott időtartam alatt meg lehessen javítani. Itt is a gyakorlatban az átlagos helyreállítási időt szokás használni, aminek a jelölése MTTR (Mean Time To Repair).

- A rendelkezésre állás (availability) annak a valószínűsége, hogy egy adott időpillanatban a számítógép működőképes legyen, függetlenül attól, hogy előzőleg hányszor romlott el és javították meg. Ennek kifejezésére a gyakorlatban a rendelkezésre állási tényezőt szokták használni:

$$K_A = \frac{MTTF}{MTTF + MTTR}$$

A fenti egyenlet nevezője a meghibásodások közötti átlagos időt (**MTBF – Mean Time Between Failures**) jelenti:

$$MTBF = MTTF + MTTR$$

Ha megfigyeljük a számítógép eddigi működési–helyreállítási ciklusait, akkor a rendelkezésre állási tényezőt a következőképpen becsülhetjük meg:

$$K_A = \frac{\text{Teljes működési idő}}{\text{Teljes működési idő} + \text{Teljes helyreállítási idő}}$$

TÉTELEK, KÉRDÉSEK

1.1. Mi a végrehajtási idő és az átbocsátóképesség? Mondjon példákat az alkalmazásukra.

1.2. Mik a benchmark programok? Milyen célú benchmarkokat ismer?

1.3. Mit értünk MIPS és MFLOPS alatt?

1.4. Adja meg a processzor teljesítőképesség-egyenletét.

1.5. Mutassa be ábrán a hagyományos (Neumann-elvű) és pipeline utasítás- feldolgozás elvét.

1.6. Fogalmazza meg Amdahl törvényét szavakban és képletben. Mondjon egy példát.

1.7. Adja meg és magyarázza el a multiprocesszoros rendszerek gyorsítási tényezőjének egyenletét, ideális és valós esetben.

1.8. Jellemezze e korszerű integrált áramköröket sűrűség, sebesség és fogyasztás szempontjából.

1.9. Határozza meg a megbízhatóságot, a meghibásodási rátát és a helyreállíthatóságot.

1.10. Határozza meg a rendelkezésre állást és adja meg a rendelkezésre állási tényező kiszámításának két képletét.

2. MODELLSZÁMÍTÓGÉP TERVEZÉSI KÉRDÉSEI

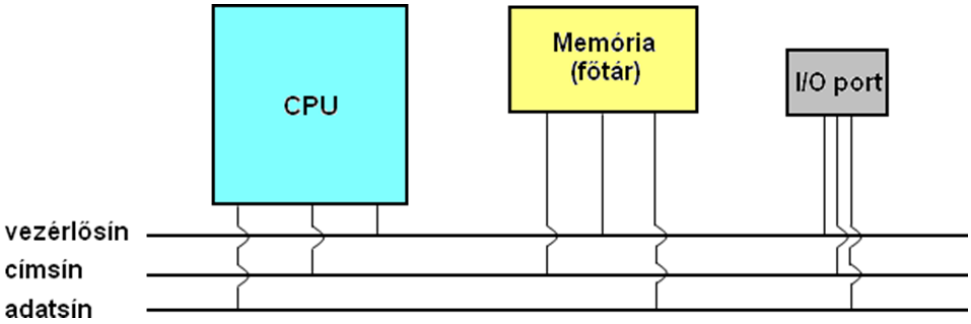
2.1. A modellszámítógép felépítése

[Kovács M., Knapp G., Ágoston Gy., Budai A. – Bevezetés a számítástechnikába, LSI, 2002, VI. fejezete nyomán]

Célunk egy mikroprocesszor-alapú számítógép általános felépítéséből kiindulva ismertetni a jelentősebb tervezési feladatokat, valamint – a következő fejezetben – megtervezni a modellprocesszor utasításkészletét, amely segítségével assembly nyelvű programozási feladatokat tudunk megoldani.

Az elképzelt mikroszámítógépünk a következő részegységekből épül fel (lásd az ábrát):

- **Processzor** (CPU) – lehívja, dekódolja és végrehajtja a memóriában lévő utasításokat;
- **Főtár** vagy operatív memória (RAM, ROM) – a végrehajtás alatt álló programokat és ezek adatait tárolja;
- **I/O portok** – a külső eszközökkel (beviteli és kiviteli perifériák, háttértárolók) biztosítják az adatátvitelt;
- **Sínrendszer** – a processzor és a többi részegység közötti kommunikációt szolgálja.



Egy konkrét processzor utasításszerkezetének meghatározásához és az utasításkészletének felépítéséhez azonban ez az általános architektúra önmagában még nem ad elegendő információt. Néhány fontosabb, eldöntendő kérdés:

- Milyen feladatokra akarjuk a regisztereket felhasználni, összesen hány regisztert tartalmazzon a CPU, egy regiszter mérete mekkora legyen (azaz hány bitből álljon)?
- Összesen hány bájtot akarunk megcímezni a számítógép memóriájában, vagy másképp fogalmazva: mekkora legyen a főtár maximális kapacitása bájtban?
- A CPU milyen állapotjellemzőit („flag”-eket) akarjuk rögzíteni az állapotregiszterben?
- Egy lépésben hány bitből álló címeket kell átvinni a címsínen?
- Egy lépésben hány bitből álló adategységeket kell mozgatni az adatsínen a processzor–memória, illetve a processzor és az I/O portok között?
- Milyen puffer és állapotregisztereket tartalmaznak az I/O portok?

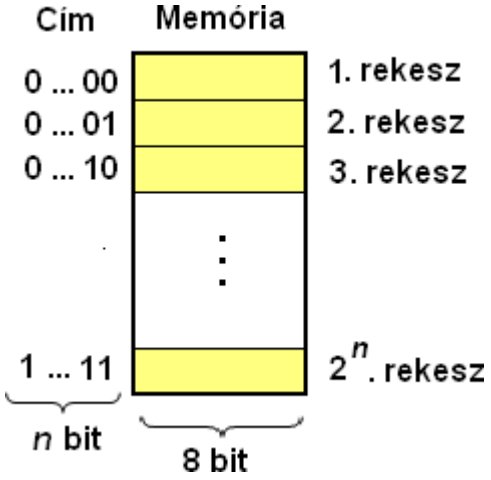
Mint látni fogjuk, ezekre a kérdésekre adható válaszok nem mindig függetlenek egymástól. Az elkövetkezendőkben modellünk konkretizálásával megadjuk a választ az előbbiekben feltett összes kérdésre.

Memória

Ha már rögzítettük a főtár maximális kapacitását bájtban, akkor ezzel egyúttal azt is eldöntöttük, hogy

- hány bitesek a címeket tároló regiszterek,
- egy lépésben hány bites címeket kell átvinni a címsínen.

Tegyük fel ugyanis, hogy számítógépünk lehetséges maximális tároló kiépítése esetén a főtár összesen 2^n darab bájtot tartalmaz. Ekkor ezeket a bájtokat n bitet tartalmazó bináris számokkal tudjuk címezni, 0-tól 2^n–1-ig (lásd az ábrát). Ha viszont egy bájt megcímzéséhez n-bit szükséges, akkor egy lépésben ennyi bitet kell átvinni a címsínen és a címeket tároló regisztereknek is képesnek kell lenniük n-bites számokat befogadni.



A napjainkban használatos Intel processzor alapú számítógépek címsínje egy lépésben 32 bites címek átvitelére képes (32 címvonalat tartalmaz). Ez egyúttal előző gondolatmenetünk alapján azt is jelenti, hogy ezekben a számítógépekben a maximális memóriakapacitás 2^{32} bájt = 4 Gigabájt lehet.

Regiszterkészlet

Egy **processzor regiszterkészletének megtervezése** során egymással ellentétes irányba ható tényezőket kell figyelembe venni:

- Mivel a regiszterek a leggyorsabb tárolók, ezért célszerű, ha a processzor minél több regisztert tartalmaz, mert nyilvánvalóan ekkor lesz a leggyorsabb.

- Minden egyes regiszter a chip áramköri egységeiből meghatározott mennyiségűt lefoglal. Emiatt a regiszterek a legdrágább tárolóeszközök, így a sok regiszter processzor árát is jelentősen megnöveli

Erre figyelemmel a processzorok tervezése során gondosan mérlegelni kell, hogy milyen feladatokra hány darab regisztert építünk be a CPU-ba.

Az elkövetkezendőkben modellprocesszorunk regiszterkészletéből az **általános regisztereket** fogjuk definiálni, azaz azokat a regisztereket, melyeket a gépi utasítások közvetlenül felhasználhatnak. Elméletileg egy processzor már akkor is működőképes, ha csak három regisztert, egy akkumulátort, egy utasításszámlálót és egy állapotregisztert tartalmaz. Ebben az esetben viszont például a verem címének kezelése, vagy a tömbök feldolgozásához szükséges indexelés csak a memóriában lenne megoldható. Ez nagyon bonyolulttá tenné a programokat és a processzort is lelassítaná. (A regiszterek hozzáférési ideje kb. tizedrésze a memóriának.)

Erre is figyelemmel a modell **processzorunk regiszterkészletét** a következőkben határozzuk meg:

AX Akkumulátor (*Accumulator*)

BX Bázisregiszter (*Basis register*)

CX Számláló regiszter (*Counter register*)

DX Adatregiszter (*Data register*)

SP Veremmutató (*Stack pointer*)

BP Bázismutató (*Basis pointer*)

SI S-Indexregiszter (*Source index*)

DI D-Indexregiszter (*Destination index*)

IP Utasításszámláló (*Instruction pointer*)

FLAGS Állapotjelző

Itt az AX, BX, CX, DX, SP, BP, SI, DI, IP a regiszterek szimbolikus rövid nevét jelöli. Az egyes regiszterek felhasználását a programokban a későbbiek során fogjuk bemutatni.

Az egyszerűség és a jobb szemléltethetőség érdekében a modellünkben szereplő **regiszterek hosszát egységesen 16 bit-ben határozzuk meg**. Ha 16-bites regisztereink vannak, akkor célszerű biztosítani a memóriából **16-bites (2 bájtos) szavak kiolvasását illetve visszaírását**. Ebből néhány dolog azonnal következik:

- A processzor maximum 16-bites, illetve 15-bites számokkal képes előjel nélküli, illetve előjeles műveleteket elvégezni;
- Számítógépünk maximális memóriakapacitása 2^{16} bájt = 64 Kilobájt;
- Számítógépünk címsínje 16 címvonalat tartalmaz;
- Számítógépünk adatsínje 16 adatvonalat tartalmaz.

Feltételezzük, hogy a főtár memóriavezérlő áramkörei a szavas hozzáférés mellett az **1 bájtos memóriaműveleteket** is lehetővé teszik. Ebben az esetben a sínrendszer úgy viszi át, a processzor pedig úgy fogadja a 8-bites bájtokat, hogy azok a regiszterek alsó legkisebb helyi értékű 8 bit-jén kerülnek letárolásra. Ezeket a bájtokat az AX, BX, CX, DX regiszterekben AL, BL, CL, DL-lel fogjuk jelölni, és az 1 bájtos memóriaműveleteket csak e regiszterekre engedjük meg.

I/O-portok

Mikroszámítógép modellünk nagyon hiányos lenne, ha nem biztosítaná az **input/output műveletek** végrehajthatóságát. Ahhoz, hogy az utasításkészletben input/output utasításokat is szerepeltetni tudjunk, konkrétan specifikálni kell azokat az I/O portokat, melyekhez a perifériák kapcsolódhatnak. Modellünkben **két I/O portot** definiálunk.

- *0-s port*, mely input adatok beolvasására szolgál,
- *1-es port*, mely output adatok továbbítására szolgál.

Az egyszerűség kedvéért képzeljük azt, hogy a 0-s port a billentyűzetről érkező adatokat fogadja, az 1-es port pedig az adatokat a monitor képernyőjére történő kiíráshoz továbbítja.

Minden port modellünkben **két 8-bites regisztert tartalmaz.** Ezek:

- a *pufferregiszter*, mely az adatok átmeneti tárolására szolgál;
- az *állapotregiszter*, mely a port állapotinformációit tárolja.

A következőkben megadjuk az állapotregiszterek egyes bitjeinek értelmezését:

A 0-s port állapotregiszterében:

2^0 bit = 1, ha a port pufferregiszterébe új adat érkezett az input eszközről
= 0, a port pufferregisztere változatlan, azaz nem érkezett új adat

2^1 bit = 1, ha a port pufferregiszterét a CPU még nem olvasta ki
= 0, ha a port pufferregiszterét a CPU már kiolvasta

Az 1-es port állapotregiszterében

2^0 bit = 1, ha nincs az output eszközhöz továbbítandó adat az 1-es port pufferében
= 0, ha az output eszközhöz továbbítandó új adat van az 1-es port pufferében

21 bit = 1, ha az output eszközhöz való adattovábbítás rendben befejeződött, a port felkészült új adat fogadására
= 0, ha az output eszközhöz való adattovábbítás még nem fejeződött be

Az állapotregiszterek 0-s értékű bitjeit a CPU, az 1-es értékű bitjeit pedig a külső eszközök állítják be. Nyilvánvaló, hogy az input/output-művelet végrehajthatóságának a CPU oldaláról az a feltétele, hogy az állapotregiszter 2^0 és 2^1 helyi értékű bitjei 1-es értékűek legyenek.

Az előző specifikációk alapján most már felvázolhatjuk **modellszámítógépünk pontos felépítését:**

A korábbiakban tárgyalt regisztereken kívül az ábrán még három **rendszerregisztert**

is feltüntettünk. Ezek:

- az IR utasításregiszter (*Instruction Register*), mely a memóriából kiolvasott programutasítás átmeneti tárolására szolgál a CPU-ban

- az MAR memória címregiszter (*Memory Adress Register*), mely a memóriából kiolvasandó adatok címét tárolja;
- az MDR memória adatregiszter (*Memory Data Register*), mely a memóriába beírandó, illetve onnan kiolvasandó adatok átmeneti tárolója.

Mivel 1, illetve 2 bájtos adategységeket akarunk mozgatni a CPU és a főtár között, nyilvánvaló, hogy MDR-nek 16-bites regiszternek kell lennie. Címeink 16 bitesek, ezért MAR is 16-bites regiszter. IR méretét nyilván az határozza meg, hogy mekkora egy gépi utasítás maximális hossza. Mint ezt a következő fejezetben le fogjuk vezetni, ez 32 bit, ennek megfelelően IR a modellünkben legyen 32-bites regiszter.

Az IR, MAR, MDR rendszerregiszterek, melyek egy gépi utasítás végrehajtásán belüli elemi lépésekben játszanak fontos szerepet. Ezek a programutasításokkal nem elérhetők, nem címezhetők és tartalmuk sem változtatható meg.

2.2 A buszciklusok

A program utasításainak a feldolgozása során a processzornak gyakran kell használnia a külső sínrendszert adatátvitel érdekében. Ezek az úgynevezett **buszciklusok** (*bus cycles*) sok utasítás végrehajtásához szükségesek, és hasonló módon zajlanak. Ez lehetővé teszi a processzor vezérlő egységének egyszerűbb megtervezését, mert azonos műveletsorozatot kell végrehajtania minden utasításnál ugyanannak a típusú buszciklusnak az esetében. A legfontosabb buszciklusok a következők:

- utasítás kiolvasása (lehívása) a memóriából (*fetch cycle*);
- olvasás a memóriából (*read cycle*);
- írás a memóriába (*write cycle*);
- bevitel (olvasás) egy port regiszteréből (*in cycle*);
- kivitel (írás) egy port regiszterébe (*out cycle*);

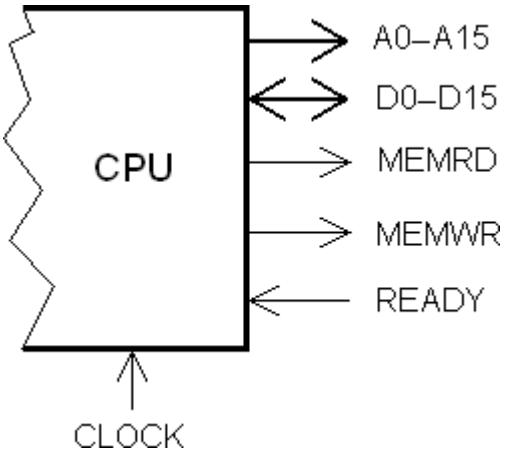
- megszakítási ciklus (*interrupt cycle*).

Az első három ciklus a leggyakoribb, ezek az alapciklusok. Az utasításlehívás is megegyezik a külső lebonyolítása szempontjából egy memória-olvasással, de a processzoron belül két sajátossággal rendelkezik:

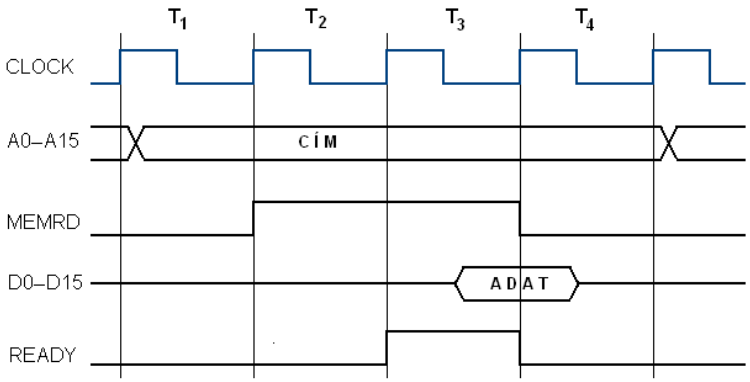
- az olvasás az utasításszámlálóban (IP) lévő címről történik;
- a beolvasott adat, ami ebben az esetben az utasítás kódja, az utasításregiszterbe (IR) kerül.

Egy **memória-olvasási ciklus** a processzor külső buszrendszere szintjén a következő tevékenységeket feltételezi (lásd az ábrát is):

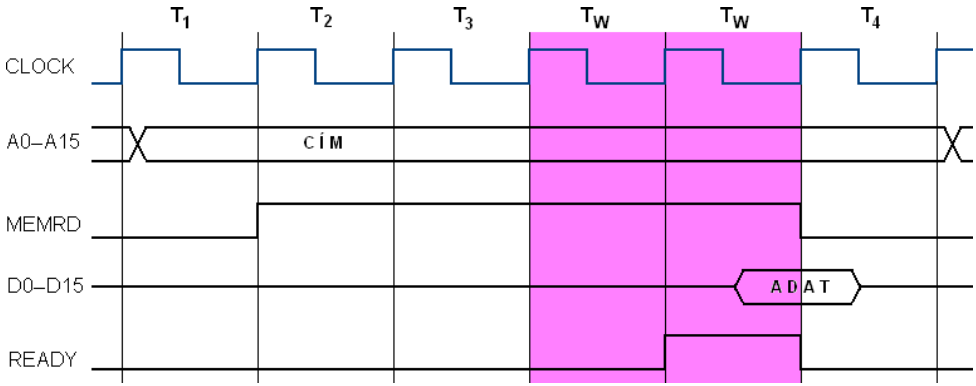
1. Az olvasni kívánt memóriarekesz címének kihelyezése a cím-buszra (A0– A15).
2. A memóriából való olvasást jelző vezérlőjel (MEMRD) aktiválása.
3. Várakozó állapotok beiktatása (a READ bemenet „0”-ra állításával) ha szükséges, abban az esetben ha a memória túl lassú a proceszszorhoz képest.
4. A memóriából érkező adat beolvasása az adatbuszról (D0–D15)



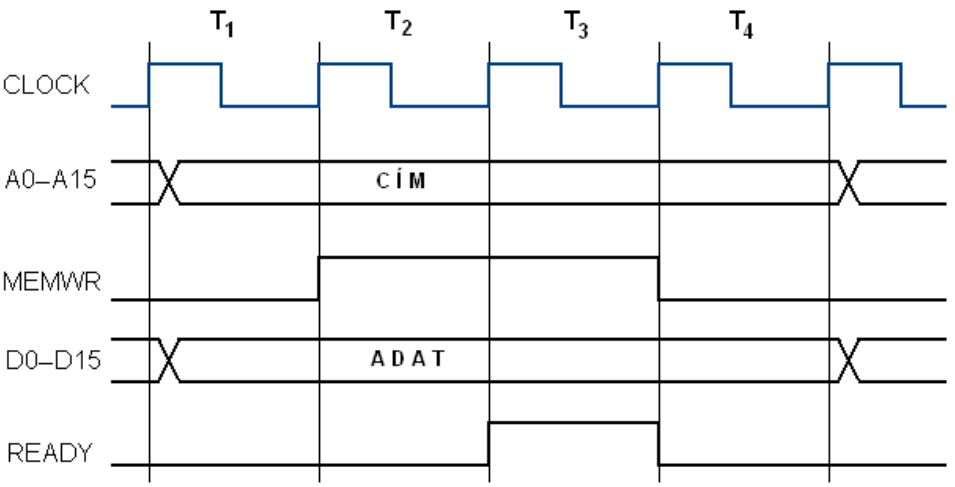
Mindezeket a műveleteket a processzor az órajelhez szinkronizálva hajtja végre, amint az a következő idődiagramból is látszik:



Egy olvasási ciklusban a kiolvasott szót a memóriának idejében az adatbuszra kell helyeznie, hogy azt a processzor helyesen átvehesse. Abban pillanatban, amikor a processzor mintavételezi az adatbuszt (a mi esetünkben a T_4 órajelciklus kezdetén), az adatvonalaknak már stabil állapotban kell lenniük. Ha viszont ez nem lehetséges, mert a memória-áramköröknek a hozzáférési ideje nagyobb, mint a válaszra rendelkezésre álló órajel-periódus, akkor T_W várakozó állapotot (vagy állapotokat) kell beiktatni a processzor buszciklusába. Ezt a READY bemenet megfelelő ideig tartó alacsony szintre („0”-ra) hozásával lehet megtenni. A következő ábra az olvasási ciklust két várakozó állapot beiktatásával mutatja:



Az írási ciklus az olvasásnak a fordítottja, ekkor a processzor helyezi az adatot az adatbuszra, majd a memóriába írást vezérlő jelet (MEMWR) aktiválja:

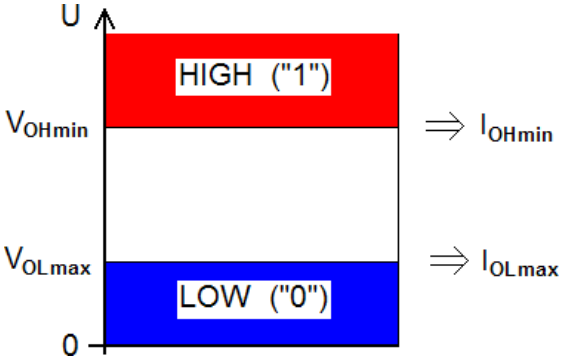


A kiviteli-beviteli buszciklusok lefolyása megegyezik a memóriaciklusokéval, azzal a különbséggel, hogy a processzor az I/O részegységekre vonatkozó vezérlőjeleket aktiválja (IORD, illetve IOWR).

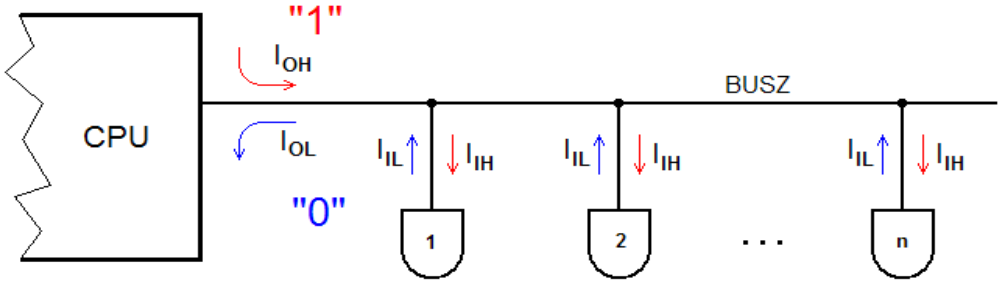
A ciklusok idődiagramjait a processzor gyártója mindig megadja, mert csak ennek betartásával lehet a rendszer többi egységét a buszhoz csatlakoztatni.

2.3 Külső sínek kialakítása

A külső busz teszi lehetővé a kapcsolatot a processzor és a többi részegység között. A bináris adatok helyes átvitele érdekében szükséges, hogy a buszvonalak feszültségszintjei a logikai „0”, valamint „1” értékeknek megfelelő intervallumban állapodjanak meg:



Mivel egy buszvonalra több vezérelt kapu van kapcsolva, Kirchhoff első törvényének megfelelően, a processzor kimenetének biztosítania kell a kapuk által igényelt áramok összegét, úgy az alacsony (*low*, azaz „0”), mint a magas (*high*, azaz „1”) állapotban:



(Az ábrán látható, hogy a „0” állapotban a processzor felveszi az áramot, az „1”-es állapotban pedig kiadja.) Minden logikai áramkör, így a processzor kimenetének is van egy névleges (katalógusban megadott) áramerőssége, amit biztosítani tud. Ezt az értéket a két feszültségtartomány szélső értékeire (V_{OLmax} és V_{OHmin}) adják meg, ami az első ábra alapján I_{OLmax} és I_{OHmin}. Feltételezve, hogy a processzor egy kimenetének a buszon keresztül *n* darab – a részegységekben lévő – kaput kell vezérelnie, a helyes logikai szintek biztosítása érdekében az alábbi feltételeknek kell teljesülniük:

$$I_{OLmax} \geq \sum_{1}^{n} I_{ILmax}$$

$$I_{OHmin} \geq \sum_{1}^{n} I_{IHmin}$$

A használt jelölések a következők:

O – output (kimenet)

I – input (bemenet)

L – low (alacsony szint)

H – high (magas szint)

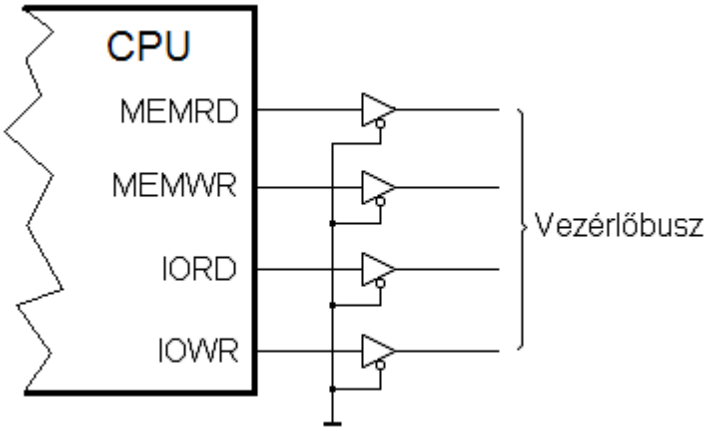
max – a V_{OLmax} határértéknek megfelelő

min – a V_{OHmin} határértéknek megfelelő

Az előző képleteket egyszerűbben tudjuk alkalmazni, ha az áramokat egy tipikus áramkörcsalád standard bemeneti áramaihoz viszonyítjuk. Ekkor a kimeneti, illetve bemeneti **terhelhetőségi tényezőkkel** (FO – *fan-out*, FI – *fan-in*) dolgozhatunk:

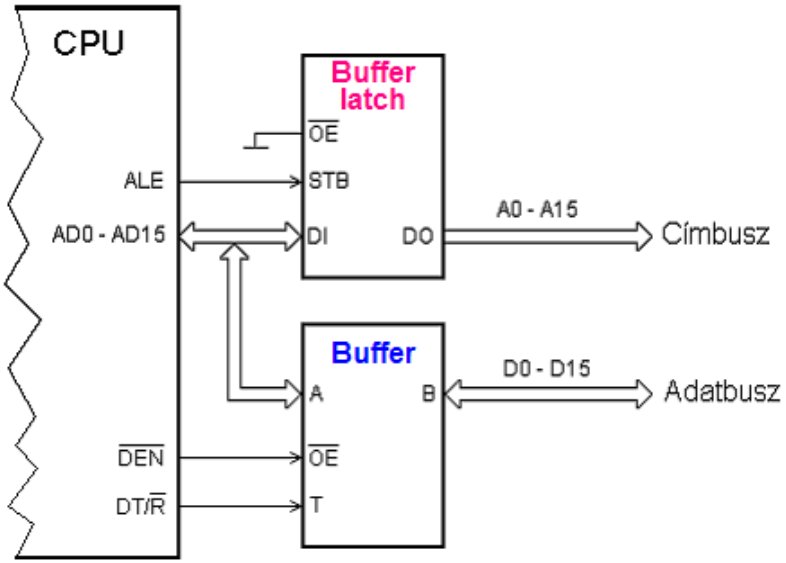
$$FO \geq \sum_{1}^{n} FI$$

Általában a mikroprocesszorok kimeneteinek kicsi a terhelhetőségi tényezője, ezért áramerősítést végző **meghajtó** (*driver*) kapukat kell beiktatni a processzor kimenete és a rendszersín közé. Célszerű ezt a meghajtót **háromállapotú kapukból** (*tri-state*) kialakítani, mert így a harmadik, magas impedanciájú állapotban át lehet engedni a buszvezérlést például egy DMA-vezérlőnek. Egy ilyen áramkör alkalmazására mutat példát a következő ábra, a vezérlőbusz kialakítása esetében:

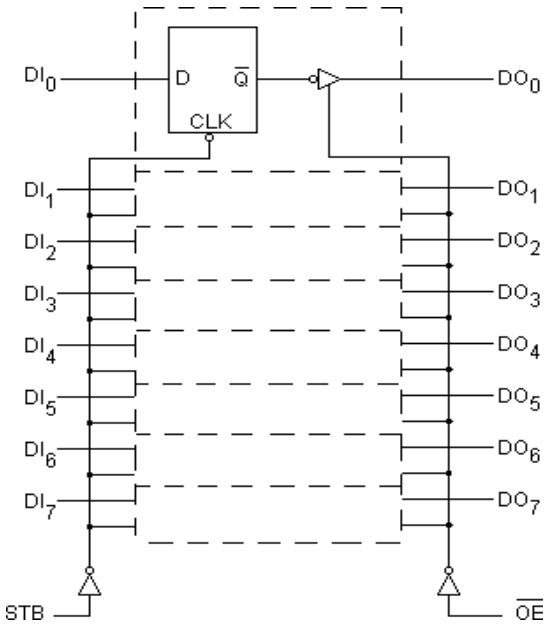


Mivel az adatbusz kétirányú, itt kétirányú **puffer meghajtókra** (*buffer driver*) van szükség. Egy ilyen 8 bites áramkörre mutat példát az alábbi ábra (OE = *Output Enable* ha aktív, megnyitja a kimeneteket, T = *Transmit* az adatátvitel irányát határozza meg):

Sok processzor **multiplexált cím- és adatbuszt** használ, így csökkentve a tok lábacskáinak a számát. Ebben az esetben a közös buszt demultiplexálni kell, azaz a különálló buszokat a processzoron kívül kell kialakítani:



Minden buszciklus (írás vagy olvasás) a cím kiadásával kezdődik. Multiplexált busz esetében ezt a címet egy külső regiszterben kell eltárolni, hogy a ciklus további részében is a címbuszon megmaradjon. A tárolás az ALE (*Address Latch Enable*) impulzussal történik. A használt *buffer latch* áramkör belső kapcsolási rajzát az alábbi ábra mutatja:



Amikor a közös buszon az adatok haladnak, a processzor egy DEN (*Data Enable*) jelet aktivál – ezzel nyitjuk meg az adatpuffert. Az adatátvitel irányát a DT/R- (*Data Transmit/Receive*) jel vezérli.

Természetesen a buszvonalak megfelelő vezérlését meg kell oldani minden olyan részegység esetében is, amely adatokat küld a buszon (memória, interfész). Ezeknek a buszra kapcsolódó kimeneteit is három állapotú puffer meghajtókkal kell ellátni úgy, hogy ha ez egység nem küld adatot, a kimenetei magas impedanciájú állapotban legyenek. Ekkor egy másik egység használhatja a buszt, elkerülve az adatok ütközését.

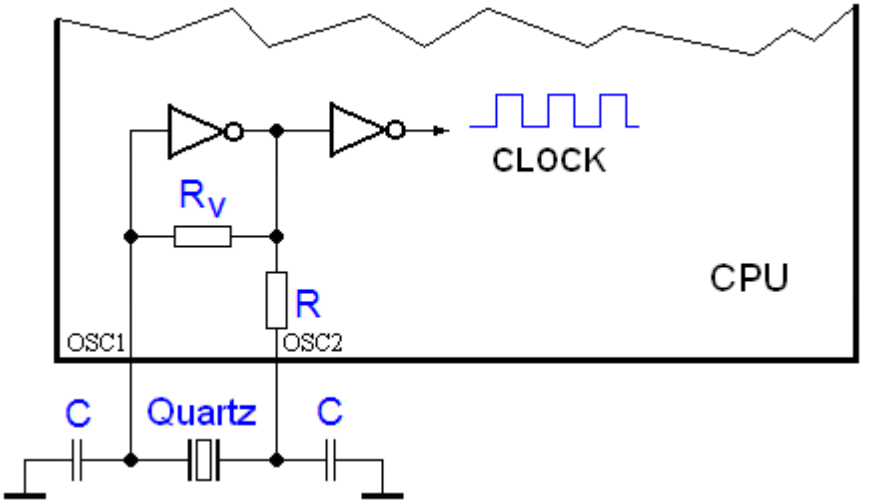
2.4 Az órajel-generátor

Az órajel (*clock*) alkalmazására minden processzor esetében szükség van, mert ez biztosítja a belső áramkörök szekvenciális működését. A hőmérséklet- és feszültségváltozásokra is stabil frekvencia biztosítása érdekében egy **kristályoszcillátort** használnak, mivel a kvarckristály a piezoelektromos tulajdonsága miatt a sajátrezgését nagyon pontosan tartja. A kristály úgy viselkedik, mint egy RLC-áramkör (ellenállás-induktivitás-kapacitás), amelynek a rezgési frekvenciája a méretétől és alakjától függ.

Az processzor típusától függően, az órajel-generátornak két kialakítási módja terjedt el:

- külsőáramkör, amelynek a jelét a processzor megfelelő (CLOCK) bemenetére kell alkalmazni
- a processzorba beépített oszcillátor, amelyre kívülről kell rákapcsolni a kvarckristályt (ezt a konstrukciót tipikusan a mikrokontrollereknél használják)

Az utóbbi megoldásra mutat példát a következő ábra:



Elektromos feszültség hatására a kristály deformálódik, majd a feszültség megszűnése után visszaveszi az eredeti alakját, és ő generál feszültséget. A rezgést az oszcillátor úgy tartja fenn, hogy az áramkör a kvarctól vett jelet felerősíti, és az inverter kimenetéről visszajuttatja a kristálynak. Amikor az így létrejött váltakozó feszültség megegyezik a kristály saját frekvenciájával, akkor rezonál. A második kapu csak erősítési célokat szolgál, a meghajtott további kapuk számára. Az ábrán látható *R, C* paramétereket a következő egyenletből számítjuk ki (*C* értéke kicsi, általában10-30 pF-os):

$$f = \frac{1}{2\pi RC}$$

Ebben *f* a frekvenciát jelöli.

A processzor névleges működési frekvenciáját a műszaki adatai között találjuk meg, de figyelmet kell fordítani az órajel kitöltési tényezőjére, valamint az impulzusok éleinek időtartamára is.

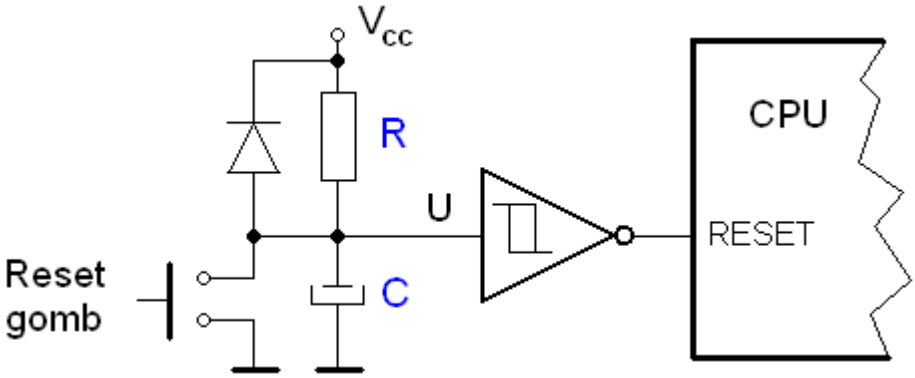
2.5 A RESET áramkör

Minden processzor rendelkezik egy RESET bemenettel, amely lehetővé teszi a processzor inicializálását, azaz kezdeti állapotba hozását. Erre egyrészt a számítógép indításánál van szükség, másrészt akkor, ha a kezelő a *Reset gomb* megnyomásával újra akarja indítani a gépet.

A RESET impulzus alkalmazásának általában két hatása van a mikroprocesszorban:

- az utasításszámláló kezdőértékre állítása (ez legtöbb processzornál nulla);
- a maszkolható megszakítások elfogadásának letiltása (ez mindaddig fennmarad, amíg a futó program egy engedélyező utasítással a tiltást fel nem oldja).

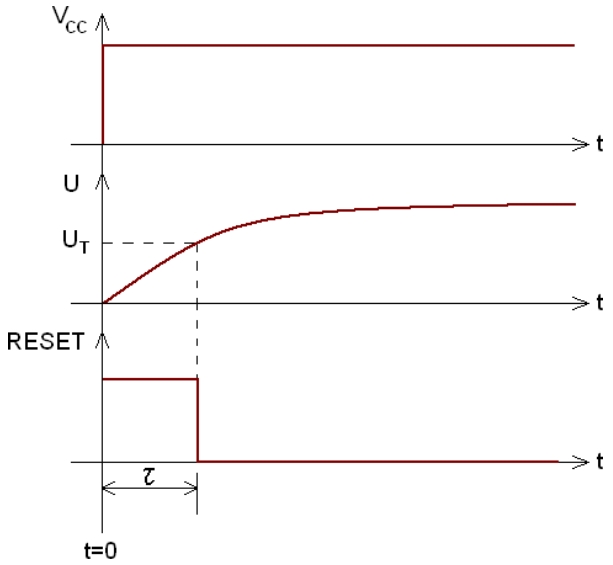
Egy RESET áramkör egyszerű kapcsolási rajzát mutatja az alábbi ábra:



A RESET gomb megnyomására egyértelmű, hogy az inverteren keresztül pozitív impulzus kerül a processzor bemenetére, ami ezt alaphelyzetbe állítja. Az R-C páros a gép bekapcsolásánál biztosítja az automatikus inicializálást. Az inverter egy *Schmitt-trigger*, amely pontos küszöbfeszültségen (legyen ez U_T) kapcsolja át a kimenetet logikai „1”-ből „0”-ba. A tápfeszültség (V_{CC}) megjelenésekor, a kondenzátor elkezd exponenciálisan töltődni az ellenálláson keresztül:

$$U = V_{CC}\left(1 - e^{\frac{t}{RC}}\right)$$

Amíg ez a feszültség el nem éri az U_T értéket, az inverter kimenetén logikai „1” van, ami alaphelyzetbe hozza a processzort. A küszöbfeszültség elérésének pillanata jelenti a pozitív impulzus időtartamának a végét, ekkor a RESET jel beáll „0”-ra. Az aktív időtartam az RC szorzattól függ (ezt az áramkör *időállandójának* is nevezik), annál hosszabb, minél nagyobb ennek értéke. A dióda szerepe az, hogy a gép kikapcsolásánál süsse ki a feltöltött állapotban lévő kondenzátort. A jelek időbeli alakulását a következő idődiagramon lehet követni:



A RESET impulzus időtartamának (τ) van minimális értéke, ami alatt a jelnek nincs hatása. Ezt az értéket a gyártó adja meg a processzor műszaki adataiban.

TÉTELEK, KÉRDÉSEK

2.1. Mi az összefüggés egy memória kapacitása és a címzéséhez szükséges bitek száma között? Mondjon egy példát.

2.2. Rajzolja le a modellszámítógép felépítését!

2.3. Vázolja fel egy memória-olvasási ciklus idődiagramját és magyarázza el a lefolyását!

2.4. Mikor és miként kell várakozó állapotokat beiktatni az olvasási ciklusba?

2.5. Vázolja fel egy memória-írási ciklus idődiagramját és magyarázza el a lefolyását!

2.6. Ismertesse a processzor és a buszra kapcsolt többi egység áramerősségei közötti feltételt, a helyes logikai szintek biztosításának érdekében!

2.7. Vázolja fel és magyarázza meg a cím- és adatbusz kialakításának elvét multiplexált buszt használó processzor esetében!
2.8. Ismertesse a kristályoszcillátor előnyét és működési elvét!
2.9. Vázolja fel egy processzor RESET áramkörének kapcsolási rajzát és ismertesse a működési elvét

3. MODELLPROCESSZOR. UTASÍTÁSKÉSZLET TERVEZÉSE. PROGRAMOZÁS.

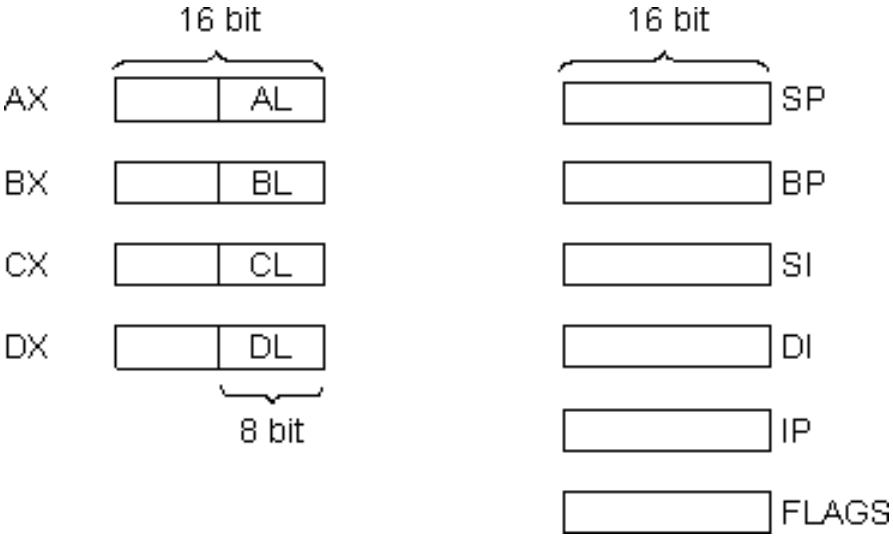
[Kovács M., Knapp G., Ágoston Gy., Budai A. – Bevezetés a számítástechnikába, LSI, 2002, VI. fejezete nyomán]

3.1. Regiszterkészlet

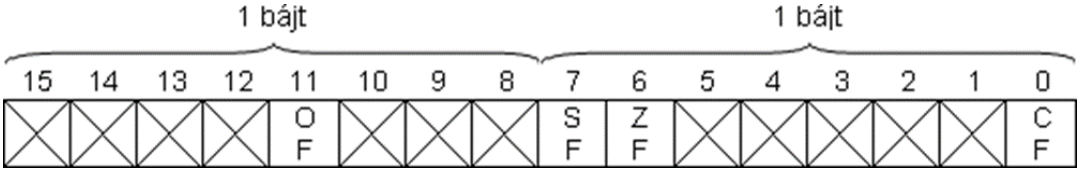
A modellprocesszorunk regiszterkészletét a következőképpen határozzuk meg:

AX	Akkumulátor (Accumulator)
BX	Bázisregiszter (Basis register)
CX	Számláló regiszter (Counter register)
DX	Adatregiszter (Data register)
SP	Veremmutató (Stack pointer)
BP	Bázismutató (Basis pointer)
SI	S-Indexregiszter (Source index)
DI	D-Indexregiszter (Destination index)
IP	Utasításszámláló (Instruction pointer)
FLAGS	Állapotjelző

Mindegyik regiszter 16 bites, de az első négy regiszter alsó felét (Al. BL, CL, DL) külön is lehet használni bájtműveletekre. A regiszterkészletet az alábbi ábra mutatja:



A FLAGS állapotjelző regiszter bitjei különösen a feltételes ugróutasítások kivitelezése szempontjából fontosak. Ez a regiszter modellünkben szintén 16 bitet fog tartalmazni a következő ábra szerint (az X-szel jelölt bitek nem használatosak):



Az ábra jelöléseivel az egyes flag-ek jelentése a következő:

- **CF** (**Carry flag**) Értéke 1, ha egy művelet eredményének legmagasabb helyi értékű (2^{15}) bitjén átvitel keletkezik
- **ZF** (**Zero flag**) Értéke 1, ha egy művelet eredményének minden bitje 0

- **SF** (**Sign flag**) Megegyezik a művelet eredményének legmagasabb helyi értékű bitjével (azaz a kettes komplemens ábrázolás szerinti értéke 1, ha az eredmény negatív)
- **OF** (**Overflow flag**) Előjeles értékkel végzett műveleteknél a túlcsordulást jelzi, azaz OF = 1, ha az eredmény nem előjelhelyes

Természetesen egy valódi processzorban a FLAGS regiszter általunk „nem használt”-nak minősített bitjei további fontos információkat tartalmaznak a processzor állapotáról, illetve vezérlik a processzor működését. Ilyen például: a paritáshiba jelzése, a hardver megszakítások letiltása stb.

3.2 Címzési módok

A processzormodellünk megalkotásában fontos szempont volt, hogy a legelterjedtebben használatos Intel processzoroktól csak a legszükségesebb mértékben térjünk el. Ezért elképzelt processzorunk jó közelítéssel egy erősen egyszerűsített, *primitív Intel processzornak* tekinthető. (Az Assembly utasítások szimbolikus nevei is azonosak az IBM PC Assemblyben megfelelő mnemonikjaival.)

A **modellprocesszor címzési módjainak** kidolgozása során arra a kérdésre kell megkeresnünk a választ, hogy a gépi utasítás- végrehajtás különböző „szereplőit”:

- az *általános regisztereket,*
- a *memória 1, illetve 2 bájtos egységeit,*
- az *I/O portok regisztereit,*

milyen módon tudjuk azonosítani a programutasításokban. Ezekhez a számítógép bináris működése miatt nyilvánvalóan bináris kódokat kell hozzárendelni és egy kódnak egyértelműen azonosítani kell egy tárolóegységet.

A **regisztereket** egyszerűen 4-bites *bináris sorszámukkal címezzük* a következő felsorolás szerint:

Binárisan	Decimálisan	Hexadecimálisan	Szimbolikus nevük
0000	0	0	AX
0001	1	1	AL
0010	2	2	BX
0011	3	3	BL
0100	4	4	CX
0101	5	5	CL
0110	6	6	DX
0111	7	7	DL
1000	8	8	SP
1001	9	9	BP
1010	10	A	SI
1011	11	B	DI
1100	12	C	IP

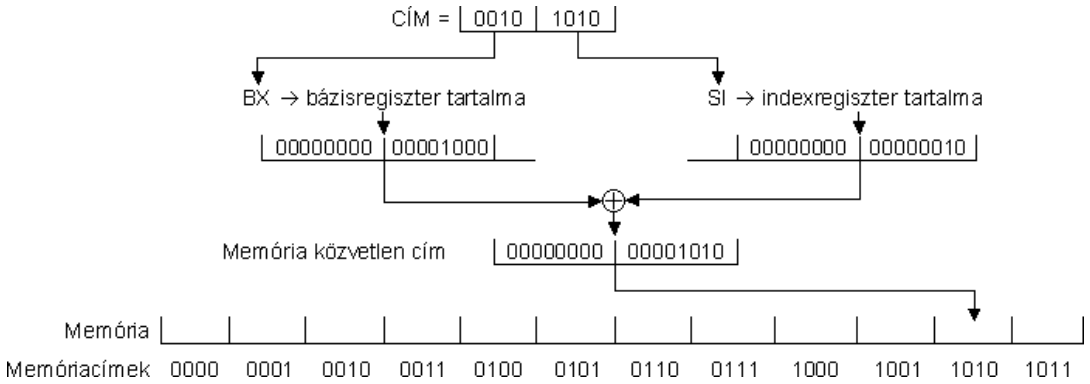
A regiszterek bináris kódjai

Felvetődhet a kérdés, hogy a FLAGS regisztert miért nem láttuk el egy bináris azonosítókóddal. Ennek az az oka, hogy ezt a regisztert közvetlenül a gépi utasításokban nem lehet megcímezni. Ugyanakkor külön utasításokkal a FLAGS regiszter tartalmához is hozzáférhetünk.

A **memória címzéséhez háromféle módot engedünk meg** processzor modellünkben. Ezek:

- a *regiszter indirekt címzése*, ekkor az utasításban szereplő regiszter tartalmazza azt a 16-bites memóriacímet, ahol az adat megtalálható;
- a *közvetlen adatcímzés*, amikor magában az utasításban található meg egy 8-, illetve 16-bites adat- vagy címérték;
- a *bázis-relatív és indexelt címzés*, amikor az utasítás két regiszter címét tartalmazza.

Ebben az esetben az adat tényleges 16-bites memóriacímét úgy számítjuk ki, hogy a két regiszterben található címet összeadjuk (lásd az ábrát).



Végezetül az **I/O portok címzését** határozzuk meg, melynél figyelembe kell venni, hogy a portban található két regisztert önállóan meg kell címeznünk az utasításokban. Ezért e portok regisztereinek a címeit a következőképpen definiáljuk:

0 port	pufferregiszter	0 0
0 port	állapotregiszter	0 1
1 port	pufferregiszter	1 0
1 port	állapotregiszter	1 1

Foglaljuk össze, hogy a programutasításokban szereplő tárolóelemeket milyen kódokkal azonosítjuk:

a/ Általános regiszterek	4-bites regiszterkód
b/ Memória indirekt címzése regiszterrel	4-bites regiszterkód
c/ Közvetlen adatcímzés	8- és 16-bites adatérték
d/ Memória bázis-relatív és -indexelt címzése	két 4-bites regiszterkód
e/ I/0 portok regisztereinek címzése	2-bites port kód

Vegyük észre a következő problémákat:

- Ha az utasításban egy operandus helyén egy regiszter kódja van, hogyan ismeri fel a processzor, hogy a regiszter egy adatot vagy egy címet tartalmaz (Másként fogalmazva, mi alapján lehet elkülöníteni az a/ és b/ esetet?);
- Ha az utasítás egy 8 bites bináris számot tartalmaz, mi alapján dönthető el, hogy ez egy közvetlen adatcímzéshez tartozó bájt vagy pedig két regiszter azonosítója (c/ és d/ eset).

A többértelműség kiküszöbölése érdekében az a/, b/, c/, d/ eseteknek megfelelő **címzési módokat binárisan fogjuk kódolni** a következőképpen:

- Regiszterben lévő adat 00 = R
- Regiszter indirekt címzése 01 = I
- Közvetlen adatcímzés 10 = D
- Bázis-relatív és -indexelt címzés 11 = B

Az elkövetkezendőkben az a/, b/, c/, d/ címzési módokat rendre R, I, D, B-vel fogjuk jelölni. Az I/O port regisztereinek a címzését külön egyedi esetként kezeljük, mivel ezt az I/O utasítás kódja egyértelműen azonosítja.

3.3 A gépi utasítás szerkezete

Mint később látni fogjuk, modellprocesszorunk egy gépi utasítása bináris formában és Assembly mnemonikkal a következő ábra szerinti formátumú:

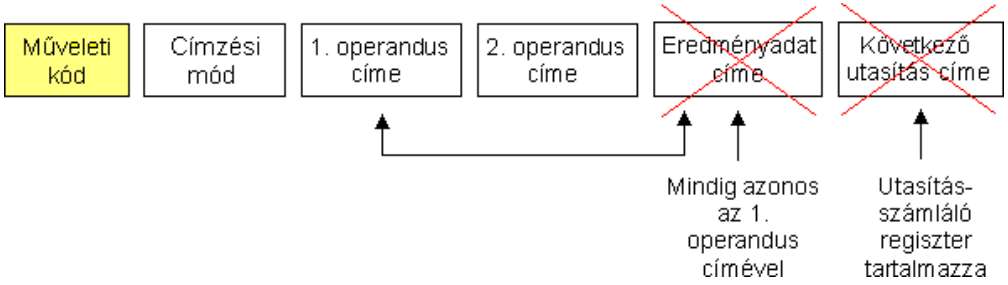
Assembly formátum:	ADD AX, [BX+SI]
Gépi kód:	0000 0000 0010 1010

Ez egy bázisindexelt címzésű két bájtot az akkumulátorhoz hozzáadó utasítás. Figyeljük meg az ábrán látható bitsorozatot és tegyük fel a kérdést: honnan állapítja meg a processzor, hogy a bitek melyik csoportja jelenti a műveleti kódot (mit kell elvégezni) és az utasításban résztvevő tárolók címét?

Nyilvánvalóan, ehhez egy egyértelmű és konzekvensen alkalmazott szabályrendszer kell, mely meghatározza, hogy az utasításokban lévő biteket a processzor hogyan értelmezze. Ezt határozza meg a **gépi utasítás szerkezete**, melyet most fogunk kidolgozni modellprocesszorunk számára.

Egy gépi utasítás funkcionális részekre való felbontását az határozza meg, hogy a processzornak milyen információra van szüksége az utasítás végrehajtásához.

Ha egy gépi utasításban maximum két adat vehet részt, akkor a művelet végrehajtásához a processzornak ismernie kell a művelet kódját, az 1. és 2. operandus címét, annak a tárolóhelynek a címét ahol az eredményt el kell helyezni, valamint azt a címet, ahol a programot folytatni kell, ha az utasítás végrehajtása befejeződött:



A következő utasítás címe és az eredményadat címe részeket kihagyhatjuk a gépi utasításból a következő két konvenció bevezetésével:

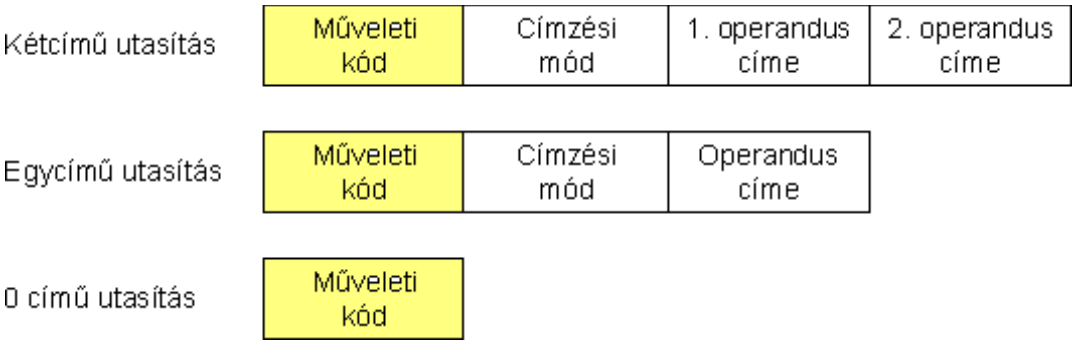
- *a következő utasítás címét mindig az IP utasításszámláló regiszter tartalmazza;*
- *két operandust tartalmazó gépi utasításoknál az eredményadatot mindig az első operandus címén* (annak korábbi állapotát felülírva) *kell elhelyezni.*

Ezt figyelembe véve modellszámítógépünk utasításkészletében maximum kétcímű utasításokat fogunk szerepeltetni.

Az utasításszerkezet specifikálása során az lesz a célunk, hogy a megtervezett utasítások minél rövidebbek legyenek, tehát ahol ez lehetséges takarékos-

kodni fogunk a bitekkel. Ennek az az értelme, hogy a rövidebb gépi utasításokból álló program egyrészt kevesebb tárolóhelyet igényel, másrészt végrehajtása is gyorsabb, mivel az utasítások kiolvasása során kisebb mennyiségű adatot kell mozgatni a főtár és a processzor között.

Az előbbieket figyelembe véve modellprocesszorunk gépi utasítása négy, három illetve két főrészből fog állni:



Próbálkozzunk most azzal, hogy a műveleti kód és a címzési mód mezőkre ne használjunk el többet 1 bájtnál. Mivel az R, I, D, B címzési módok 2 biten kódolhatóak, ezért a **címzési mód mező**

- kétcímű utasításnál 4 bit,
- egy című utasításnál 2 bit,
- 0 című utasításnál 0 bit

tárolóhelyet igényel. Ebben az esetben a műveleti kód kódolására

- kétcímű utasításnál 4 bit,
- egycímű utasításnál 6 bit,
- 0 című utasításnál 8 bit lehetőségünk marad.

A műveleti kódot OP-vel, a címzési módot AM jelölve ezt mutatja be a következő ábra:

Bitpozíció	7	6	5	4	3	2	1	0
Kétcímű utasítások	OP				AM			
Egycímű utasítások	OP						AM	
0 című utasítások	OP							

Ennek megfelelően az utasításkészletben például maximum 16 (2^4) kétcímű utasítást leszünk képesek kódolni. Ha ezekbe a keretekbe az utasításkészletbe kiválasztott konkrét utasítások belefférnek, akkor a műveleti kód és a címzési mód bináris kódolását 1 bájton meg tudjuk oldani.

Tekintsük most át utasításaink **címrészének felépítését**. Ehhez emlékeztetünk arra, hogy a címzési mód szerint:

- R 4 bit
- I 4 bit
- D 16 bit (8 bit)
- B 4 + 4 = 8 bit tárolóhelyre van szükségünk.

Itt beleütköztünk abba a problémába, hogy 8- és 16-bites adategységeket is szeretnénk mozgatni a főtár és a CPU között. Hogyan lehet ezt kifejezni az utasításszerkezetben, mivel a kódolásra felhasználható bitjeink már „elfogytak".

Ezt az ellentmondást a következő **konvencióval** küszöbölhetjük ki: *ha az 1-es operandusban 16- illetve 8-bites regiszter szerepel, akkor ennek megfelelően a 2-es operandus által meghatározott memóriacímről 2 illetve 1 bájtot kell átvinni a CPU-ba.*

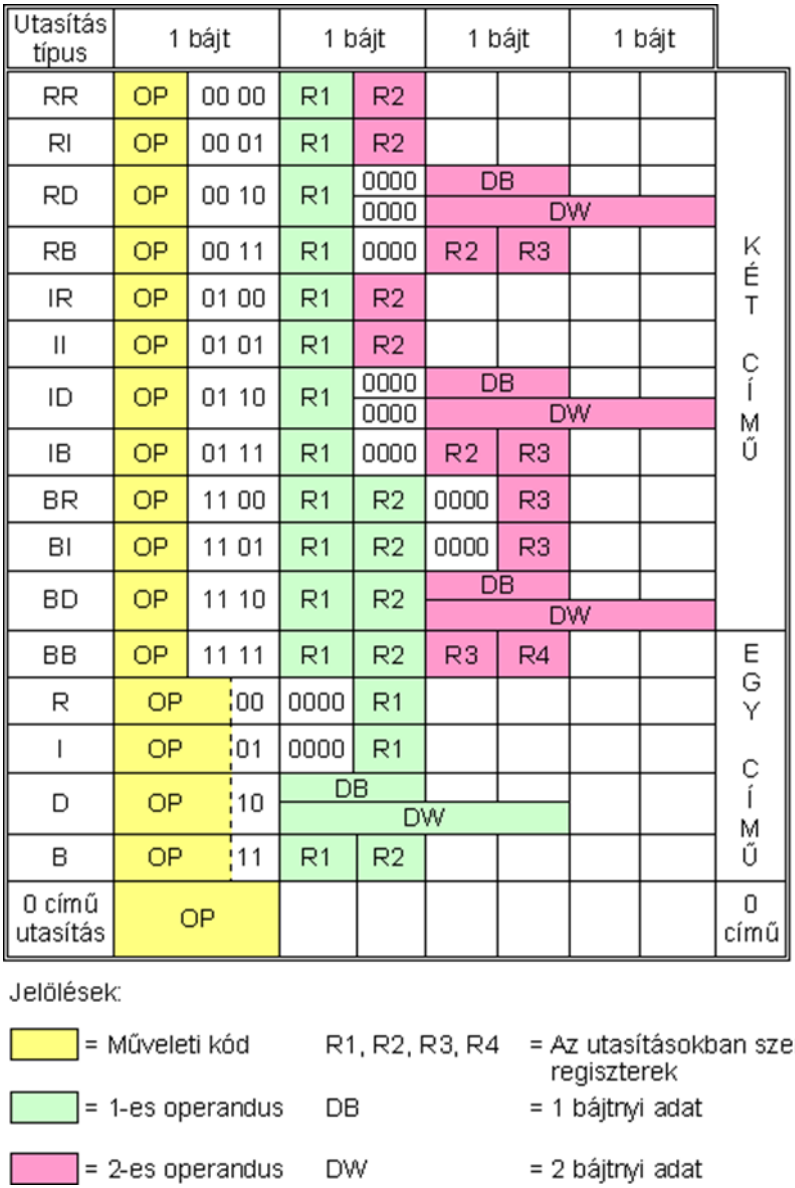
Az utasítások címrészének felépítése szerint elvileg a következő **utasítástípusok** lehetségesek:

- *Kétcímű utasítások:* RR, RI, RD, RB,
  IR, II, ID, IB, BR, BI, BD, BB
- *Egycímű utasítások:* R, I, D, B
- *0 című utasítások*

Ehhez két megjegyzést célszerű tenni:

- Az utasítástípusok most még csak „elméletileg” léteznek, ezeket a variációkat majd a konkrét utasítások definiálásakor töltjük fel tartalommal.
- A DR, DI stb. utasítástípusok nyilvánvalóan nem értelmesek, mert konvenciónk szerint az utasítás eredménye az 1-es operandusban keletkezik.

A modellprocesszorunk utasításszerkezetének teljeskörű definícióját az alábbi ábra tartalmazza.



A modellprocesszor utasításszerkezetének definíciója

Az ábra alapján látható, hogy

- a kétcímű utasítások 2, 3 és 4 bájt,
- az egycíműek 2, 3 bájt

- a 0 címűek 1 bájt hosszúak.

3.4 Az utasításkészlethez tartozó utasítások kiválasztása és kódolása

Az elkövetkezendőkben sorra vesszük a számítógép különböző utasításcsoportjait és a funkciókat értékelve meghatározzuk, hogy modellprocesszorunk utasításkészletében milyen utasításokat szeretnénk szerepeltetni. A kiválasztásnál arra törekszünk, hogy a leggyakrabban használt Assembly utasítások szerepeljenek az utasításkészletben és ezekből olyan választék legyen, mely programok megírását is lehetővé teszi.

Aritmetikai és logikai utasítások

Az **aritmetikai műveletek** közül csak a legszükségesebbeket választjuk ki. Ezek:

- ADD *összeadás* (fixpontos)
- SUB *kivonás* (fixpontos)
- CMP *összehasonlítás*
- INC *inkrementálás, azaz 1-gyel növelés*
- DEC *dekrementálás, azaz 1-gyel csökkentés*
- NEG *komplementálás, azaz (-1)-gyel való szorzás* (kettes komplemens).

A **logikai műveletek** közül négyet szerepeltetünk modellprocesszorunk utasításkészletében. Ezek:

- AND *logikai „ÉS”*
- OR *logikai „VAGY”*
- XOR *logikai kizáró „VAGY”*
- NOT *logikai „NEM”* (egyes komplemens)

Adatmozgató utasítások

Az adatmozgató utasításokat úgy fogjuk kiválasztani, hogy segítségükkel

- a memória és regiszter
- a regiszter és regiszter
- a regiszter és verem
- a regiszter és I/O portok

közötti adatátvitel végrehajtható legyen. Ezek:

- MOV adatmozgatás memória és regiszter, regiszter és regiszter között
- PUSH regiszter betöltése verembe
- POP verem tetejének betöltése regiszterbe
- PUSHF FLAGS regiszter mentése verembe
- POPF FLAGS regiszter betöltése veremből
- IN adatolvasás I/O portról
- OUT adatküldés I/O portra.

Vezérlésátadó utasítások

E körbe tartozó utasítások közül a ciklusképzéshez, a szubrutinokhoz, a feltételes és feltétel nélküli ugráshoz, valamint a programvezérelt megszakításhoz szükséges műveleteket választjuk ki. Ezek:

- LOOP *CX értékének dekrementálása és ugrás, ha CX□O*
- CALL *ugrás szubrutinba* (szubrutin felhívása)
- RET *visszatérés szubrutinból*
- JMP *feltétel nélküli ugrás* az utasításban megadott címre
- JZ/JE *ugrás, ha ZF = 1* / ugrás, ha egyenlő (CMP után)
- JNZ/JNE *ugrás, ha ZF = 0* / ugrás, ha nem egyenlő (CMP után)
- JC/JB *ugrás, ha CF = 1* / ugrás, ha kisebb

- JNC/JAE *ugrás ha CF = 0* / ugrás, ha nagyobb egyenlő
- INT programvezérelt megszakításkérés
- IRET visszatérés megszakítás-kiszolgáló rutinból.

Bitléptető és forgató utasítások

- SHL *Logikai eltolás balra*
- SHR *Logikai eltolás jobbra*
- RCL *Forgatás balra CF-fel*
- RCR *Forgatás jobbra CF-fel*

A bitléptető és forgató utasításokat a fejezet második részében bemutatott mintaprogramokban fogjuk felhasználni. Néhány ilyen utasítást azért szerepeltettünk az utasításkészletben, mert ezek nélkül utasításkészletünk nem reprezentálná a valódi processzorokat. Érdemes felhívni a figyelmet arra, hogy bár a szorzás és osztás művelete nincs benne modellparancsunk utasításkészletében, ezek is elvégezhetők ismételt összeadással és a léptető utasítások felhasználásával.

Processzorvezérlő utasítások

- CLC *CF beállítása 0-ra*
- STC *CF beállítása 1-re*
- HLT *Processzor leállítása egy hardver megszakításig*

Miután funkcionálisan rögzítettük modellprocesszorunk utasításkészletét, a következőkben meghatározzuk az **egyes utasítások műveleti kódját**. Ehhez az egyes utasításokban elvégzendő műveletek értékelésével először meghatározzuk, hogy utasításaink közül melyek a kétcímesek, egycímesek illetve 0-címűek. (A processzorvezérlő utasítások, a PUSHF és POPF, valamint a RET és IRET 0-című utasítások).

Soroljuk fel először a kétcímű, majd az egycíműeket, végül a 0 című utasításokat és mindegyikhez rendeljünk hozzá rendre a 4, 6, 8 bites műveleti kódokat (lásd az ábrát).

Utasításkód mnemonik	Műveleti kód binárisan			
	4 bit	2 bit	2 bit	
ADD	0000			KÉT-CÍMŰ UTASÍTÁSOK
SUB	0001			
CMP	0010			
AND	0011			
OR	0100			
XOR	0101			
MOV	0110			
SHL	0111			
SHR	1000			
RCL	1001			
RCR	1010			
INC	1011	00		EGY-CÍMŰ UTASÍTÁSOK
DEC	1011	01		
NEG	1011	10		
NOT	1011	11		
PUSH	1100	00		
POP	1100	01		
IN	1100	10		
OUT	1100	11		
LOOP	1101	00		
CALL	1101	01		
JMP	1101	10		
JZ/JE	1101	11		
JNZ/JNE	1110	00		
JC/JB	1110	01		
JNC/JAE	1110	10		
INT	1110	11		
RET	1111	00	00	0 CÍMŰ UTASÍTÁSOK
IRET	1111	00	01	
PUSHF	1111	00	10	
POPF	1111	00	11	
CLC	1111	01	00	
STC	1111	01	01	
HLT	1111	01	10	

3.3. táblázat. *A modellprocesszor utasításkódjai*

Vegyük észre, hogy a műveleti kód alapján a processzor egyértelműen el tudja dönteni, hogy kétcímű, egycímű vagy 0 című utasítást olvasott be:

$$\text{műveleti kód első 4 bitje} \leq 1010 \rightarrow \text{kétcímű utasítás}$$

$$1010 < \text{műveleti kód első 4 bitje} \leq 1110 \rightarrow \text{egycímű utasítás}$$

műveleti kód első 4 bitje = 1111 → 0 című utasítás

3.5 Utasításkészlet gépi kódban. Programkészítés.

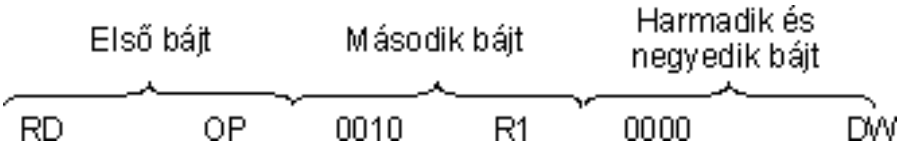
Tulajdonképpen már minden információval rendelkezünk ahhoz, hogy a gépi kódú utasításokat binárisan le tudjuk írni. Nézzünk erre példákat! Az egyes, konkrét utasítások meghatározásához semmi mást nem kell tennünk, csak a 3.1, 3.2, 3.3 táblázat adatait felhasználnunk.

1. Példa

Adjuk meg annak az utasításnak a gépi kódját, mely az AX akkumulátorban lévő értéket 5-tel megnöveli.

Megoldás

- A szükséges utasítás nyilvánvalóan az összeadás lesz.
- AX-hez egy konkrét számot kell hozzáadnunk, így az 1-es operandus egy regiszter, a 2-es operandus pedig a közvetlen adatcímzésű „5”-ös szám. Az utasítástípus tehát RD.
- A 3.1. számú táblázat szerint AX kódja: 0000.
- A 3.3. számú táblázatból kikeressük az összeadás (ADD) műveleti kódját, ez: 0000
- A 3.2. számú ábra táblázatból kikeressük az RD utasítástípust. A megfelelő sor ezt tartalmazza:

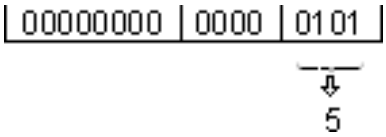


Itt OP az ADD műveleti kódja, tehát 0000

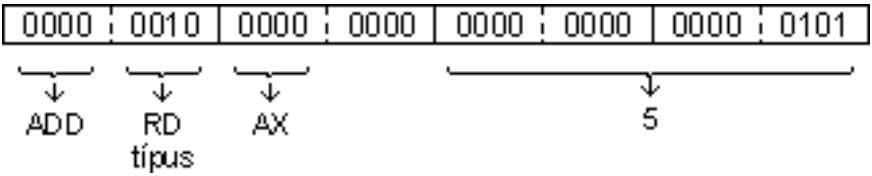
R1 az AX-t jelenti, tehát 000

DW az 5-ös számot jelenti 16 biten binárisan kódolva.

Ennek értéke tehát:



Behelyettesítve RD utasításszerkezetébe a keresett utasítás gépi kódját a következő ábra mutatja:



2. Példa

Töltsük fel bináris 0-val a CX regisztert! (Ezt a szakmai zsargonban legtöbbször úgy fejezik ki, hogy töröljük CX-et)

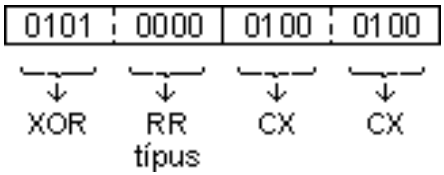
Megoldás

Ezt a feladatot az 1. példához hasonlóan a MOV utasítással is megoldhatjuk (ezt gyakorlásként ajánljuk az olvasónak), de „elegánsabb" és gyorsabb megoldás az XOR „kizáró vagy" utasítás alkalmazása. Az XOR CX, CX utasítás végrehajtása után ugyanis CX tartalma bináris 0 lesz.

Az XOR műveleti kódja: 0101

A CX regiszter kódja: 0100

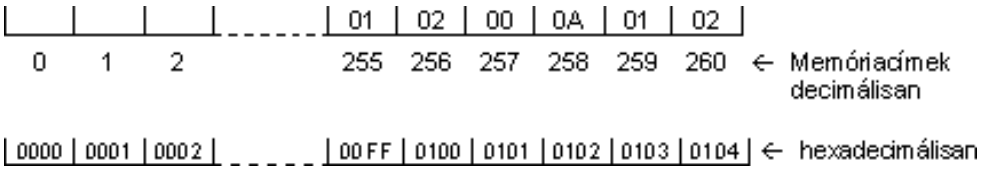
Utasításunk nyilván RR típusú, ezért a keresett gépi kód a következő:



3. Példa

Írjunk egy kis programrészt a következő feladatra:

A memória 255-ik bájtjától kezdődően 6 db 1 bájtos szám található, amint az alábbi ábra mutatja:



Adjuk össze ezt a 6 db számot úgy, hogy az eredmény AX-ben keletkezzen.

1. Megoldás

```
XOR     AX, AX      ;   Először AX-et és DX-et törölni kell,
XOR     DX, DX      ;   mert nem tudjuk mi a tartalma
MOV     BX, 255     ;   BX-be az első bájt címének elhelyezése
MOV     DL, [BX]    ;   Vigyél DX alsó 1 bájtjába a
                    ;   BX regiszterben található címről 1 bájtot
ADD     AX, DX      ;   Add össze AX-et és DX-et (az eredmény
                    ;   AX-ben keletkezik)
MOV     BX, 256     ;   Az előző 3 utasítást most még 5-ször ismé-
MOV     DL, [BX]    ;   telni kell a fennmaradó 5 bájt
ADD     AX, DX      ;   összeadásához
MOV     BX, 257
MOV     DL, [BX]
ADD     AX, DX
MOV     BX, 258
MOV     DL, [BX]
ADD     AX, DX
MOV     BX, 259
MOV     DL, [BX]
ADD     AX, DX
MOV     BX, 260
MOV     DL, [BX]
ADD     AX, DX
```

Tegyük fel most a kérdést: hogyan oldanánk meg ezt a feladatot, ha nem 6 db, hanem mondjuk 600 számot kellene összeadni? Nyilván nem szeretnénk a két MOV és az egy ADD utasítást 600-szor megismételni.

2. Megoldás

Vegyük észre, hogy az előző programrészletben a

```
MOV BX, ?
MOV DL, [BX]
ADD AX, DX
```

utasításokat úgy ismételtük, hogy csak a ? -lel jelölt értéket (címet) változtattuk. Ez a cím először 255 volt, azután 256, 257 stb., azaz mindig 1-gyel növekedett. BX értékének 1-gyel való megnövelését viszont megoldhatjuk az INC utasítással.

Ha BX értéke már 255, akkor a következő két utasítássorozat teljesen egyenértékű:

```
MOV BX, 256          INC BX
MOV DL, [BX]    =    MOV DL, [BX]
ADD AX, DX           ADD AX, DX
```

A jobboldali utasítássorozat már nem tartalmazza a 256, 257 stb. konstansokat. Ezért, ha azt mondhatnánk, hogy ezt a három utasítást ismételd meg hatszor, akkor célhoz jutnánk.

Ezt úgy fogjuk elérni, hogy a CX számlálóregiszter kezdeti értékét 6-tal (a számok darabszámával) feltöltjük, majd minden egyes szám hozzáadását követően CX értékét 1-gyel csökkentjük (DEC utasítás).

Végül egy JNZ (ugrás, ha ZF = 0, azaz CX ≠ 0) utasítással „visszaugrunk” az összeadást addig ismételni, amíg CX értéke 0 nem lesz. Ekkor már ZF = 1-re lesz állítva, tehát nem fog bekövetkezni visszaugrás.

Ezt figyelembe véve a feladatot megoldó programrészletünk a következő lesz:

```
1        XOR    AX, AX
2        XOR    DX, DX
3        MOV    BX, 255
4        MOV    CX, 6
```

```
5 ISMET:    MOV     DL, [BX]
6           ADD     AX, DX
7           INC     BX          ; BX értékének növelése 1-gyel, hogy a
                                ; következő bájtot címezze meg
8           DEC     CX          ; CX értékének csökkentése 1-gyel
9           JNZ     ISMET       ; ugrás, ha CS ≠ 0
```

E programrészletben az „ISMET:" az Assembly programnyelvben alkalmazható ún. **szimbolikus címke**, melyet a fordítóprogram a MOV DL, □BX□ utasítás memóriabeli kezdőcímének fog értelmezni.

A következő táblázatban feltüntetjük, hogy az előző programrészlet végrehajtása során az utasítások befejezésekor az AX, BX, CX, DX regiszterek tartalma milyen hexadecimális értéket vesz fel. (A 6 db 1 bájtos szám a 3. példa elején megadott ábra szerinti értékeket tartalmazza.)

Javasoljuk az olvasónak, hogy egy ilyen üres táblázatot először a programutasítások alapján próbáljon meg kitölteni, majd az eredményt a 3.4. számú táblázat alapján ellenőrizze.

	Utasítás sorszáma	AX	BX	CX	DX
	1	0000	?	?	?
	2	0000	?	?	0000
	3	0000	00FF	?	0000
	4	0000	00FF	0006	0000
1. ÁTFUTÁS	5	0000	00FF	0006	0001
	6	0001	00FF	0006	0001
	7	0001	0100	0006	0001
	8	0001	0100	0005	0001
	9	0001	0100	0005	0001
2. ÁTFUTÁS	5	0001	0100	0005	0002
	6	0003	0100	0005	0002
	7	0003	0101	0005	0002
	8	0003	0101	0004	0002
	9	0003	0101	0004	0002
3. ÁTFUTÁS	5	0003	0101	0004	0000
	6	0003	0101	0004	0000
	7	0003	0102	0004	0000
	8	0003	0102	0003	0000
	9	0003	0102	0003	0000
4. ÁTFUTÁS	5	0003	0102	0003	000A
	6	000D	0102	0003	000A
	7	000D	0103	0003	000A
	8	000D	0103	0002	000A
	9	000D	0103	0002	000A
5. ÁTFUTÁS	5	000D	0103	0002	0001
	6	000E	0103	0002	0001
	7	000E	0104	0002	0001
	8	000E	0104	0001	0001
	9	000E	0104	0001	0001
6. ÁTFUTÁS	5	000E	0104	0001	0002
	6	0010	0104	0001	0002
	7	0010	0105	0001	0002
	8	0010	0105	0000	0002
	9	0010	0105	0000	0002

Jelölés
= A megfelelő utasítás végrehajtása során az eredményadatok helye

A regisztertartalmak változása programrészletünk végrehajtása során

Látható, hogy programrészletünk végrehajtása során az 5, 6, 7, 8, 9 sz. utasítások 6-szor kerülnek végrehajtásra.

Ha egy utasítássorozatot egymás után többször, változatlan formában, de általában változó adatokkal hajtunk végre a program futtatása során, akkor ezt **ciklusnak** nevezzük.

Az egyik gyakori programhiba az úgynevezett „örök ciklus”. Az előző programpéldánkban ez az eset állt volna elő, ha a 4 sorszámú MOV CX, 6 utasítást az utasítássorból „kifelejtjük” és a CX regiszter tartalma a ciklusba való

belépéskor 0000 vagy egy negatív szám. Ebben az esetben CX értéke a dekrementálások hatására negatív lesz és ezáltal a ZF flag sohasem fogja felvenni az 1 értéket. Emiatt a program végrehajtásakor az 5, 6, 7, 8, 9 utasításokat a számítógép „végtelen sokszor” megismétli.

Az előző programrészletet felhasználjuk arra, hogy a gépi kódú programokat is bemutassuk, és kis betekintést nyerjünk az Assembly nyelvű programok fordítóprogramjának működésébe.

Tételezzük fel, hogy programrészletünk a memóriában közvetlenül a 6 db 1 bájtos szám után helyezkedik el. Ekkor kezdőcíme decimálisan 261, hexadecimálisan pedig 0105. A korábban már megismert módon állítsuk elő a 9 db utasítás gépi kódját:

		1 bájt		1 bájt		1 bájt		1 bájt	
	XOR AX, AX	0101	0000	0000	0000				
	XOR DX, DX	0101	0000	0110	0110				
	MOV BX, 255	0110	0010	0010	0000	0000	0000	1111	1111
	MOV CX, 6	0110	0010	0100	0000	0000	0000	0000	0110
ISMET:	MOV DL, [BX]	0110	0001	0111	0010				
	ADD AX, DX	0000	0000	0000	0110				
	INC BX	1011	0000	0000	0010				
	DEC CX	1011	0100	0000	0100				
	JNZ ISMET	1110	0010	0000	0001	0001	0001		

Programrészletünk gépi kódban

A 3.5. számú táblázatot a 3.1., 3.2. és 3.3. számú táblázatok alapján állítottuk össze. Ennek során a lényegét tekintve a **fordítóprogramot szimuláltuk**:

- A szimbolikus műveleti kód mnemonikok (pl. XOR, MOV stb.) alapján kikerestük a műveleti kód táblázatból a műveleti kód bináris értékét.
- „Felismertük” a regiszterek szimbolikus nevét (AX, BX, stb.) ez alapján a regiszterkódtáblázatból kikerestük a regiszterek bináris kódját.

- Az utasításokban található regiszter darabszám, konstansok, az indirekt címzést jelölő [ ] jel alapján meghatároztuk, hogy az utasításoknak milyen az utasításszerkezete. Ezt követően az utasításszerkezetre vonatkozó táblázatból kikerestük az utasítás formai felépítését.
- Ha mindezt megtettük, ezt követően már csak az utasításokban lévő konstansokat kellett binárisra átkonvertálni és a szimbolikus címeknek (ISMET:) a bináris értékét meghatározni. Ezt követően kitölthettük a 3.5. számú táblázatot gépi kódban.

A **szimbolikus címek bináris értékének meghatározása** még további magyarázatot igényel.

Mint már említettük az Assembly konvenciók szerint az „ISMET:" szimbolikus cím a MOV DL, [BX] utasítás gépi kódjának tárolóbeli címét jelenti. Ahhoz, hogy a JNZ utasításban lévő ugrási cím (mely szimbolikusan ISMET:) bináris értékét meghatározzuk, egy rövid számításra van szükség.

Ennek során először az ISMET: címkével jelölt MOV utasítás gépi kódjának a programrészlet elejétől való távolságát határozzuk meg bájtokban számolva.

A 3.5. számú táblázat szerint az

ISMET: MOV DL, [BX]

utasítást 2 darab 2 bájtos (2 db. XOR) és 2 darab 4 bájtos (2 db. MOV) utasítás előzi meg. Ez összesen

$$2 \times 2 + 2 \times 4 = 12$$

bájtot jelent. Ez azt jelenti, hogy az ISMET: szimbolikus címnek megfelelő cím a programrészlet kezdőcímétől 12 bájtra található. Mivel tudjuk, hogy programrészletünk kezdő címe a memóriában 261, ezért ISMET: címe:

$$261 + 12 = 273$$

vagy hexadecimálisan és binárisan:

$$0111_H = 0000\ 0001\ 0001\ 0001$$

Példánk alapján világos, hogy az Assembly nyelvű programokból egy jól felépített szabály és eljárás rendszer alapján előállíthatjuk az utasítások gépi kódját. Erre az algoritmusra tehát programot is írhatunk, amit **Assembler**-nek nevezünk. Az Assembler tehát *az a fordítóprogram, mely az Assembly nyelven megadott programutasításokból gépi kódú utasításokat állít elő.*

TÉTELEK, KÉRDÉSEK

3.1. Mi a szerepe a FLAGS regiszternek, és milyen biteket tartalmaz?

3.2. Mit kell megadni az utasítás címrészében, és hol található az operandus a következő címzési módoknál:

- regiszterben lévő adat
- regiszter indirekt címzése
- közvetlen adatcímzés
- bázis-relatív és -indexelt címzés

3.3. Vázolja a két operandust feldolgozó utasítás logikai szerkezetét! Mely mezőket lehet ebből kihagyni, és miért?

3.4. Adjon öt példát aritmetikai és logikai utasításokra!

3.5. Adjon öt példát adatmozgató utasításokra!

3.6. Adjon öt példát vezérlésátadó utasításokra!

3.7. Mi a szimbolikus címke, és hogy kezeli a fordítóprogram?

4. HUZALOZOTT ÉS MIKRO-PROGRAMOZOTT VEZÉRLŐEGY-SÉG TERVEZÉSI KÉRDÉSEI

4.1 A vezérlőegység feladata

Egy utasítás végrehajtása több ciklus végrehajtását jelenti, mint az utasítás lehívása, oparandus címszámítása, végrehajtás, megszakítás. Az utasításciklus minden ilyen kisebb alciklusa (fázisa) elemi lépések sorozatát feltételezi, amelyekben olyan egyszerű műveletek zajlanak, mint két regiszter közötti adatátvitel, egy regiszter és a külső busz közötti átvitel, vagy egyszerűen egy aritmetikai művelet. Megállapítható tehát, hogy *egy utasításciklus minden fázisát* ***elemi műveletek*** *sorozatára lehet lebontani, amelyeket egy-egy időegység alatt kell végrehajtani:*

PROGRAM VÉGREHAJTÁSA---UTASÍTÁSCIKLUSOK---ALCIKLUSOK (lehívás, címszámítás, stb.)---ELEMI MŰVELETEK

A processzor **vezérlőegysége** (*control unit*) felel minden utasítás elemi műveletének helyes sorrendben való végrehajtásáért, aminek érdekében a megfelelő időpillanatokban vezérlőjeleket bocsát ki. A vezérlőegység jellemzése több szempontot feltételez:

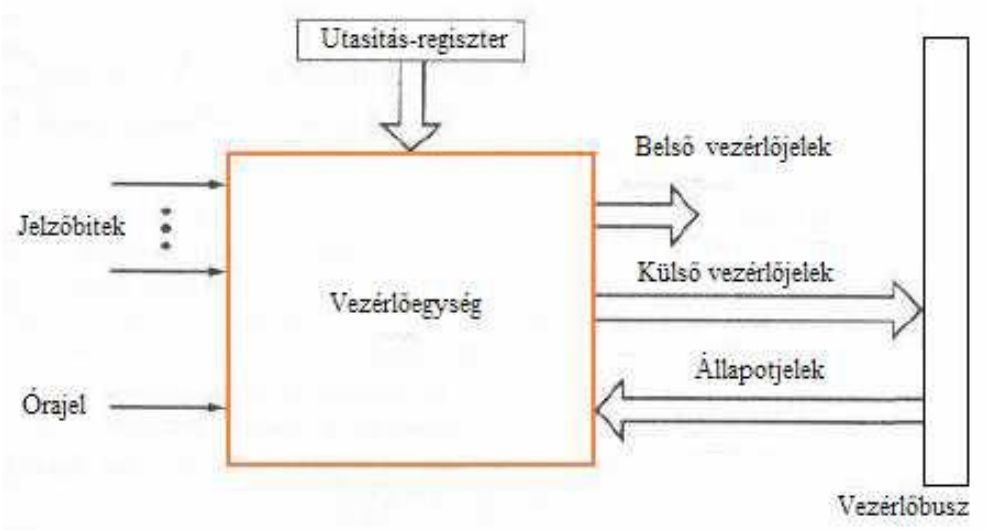
1. A processzor *alapegységeinek* a meghatározása (ALU, regiszterek, belső buszok, külső buszok, vezérlőegység).
2. A processzor által végrehajtandó *utasítások* specifikálása.

3. Az utasítások által feltételezett *elemi műveletek* meghatározása.
4. A vezérlőegységnek az elemi műveletek végrehajtására irányuló *feladatainak* a meghatározása.

A vezérlőegységnek két alapfeladata van:

- Az elemi lépések *megfelelő sorrendben* való végrehajtása, a feldolgozott programnak megfelelően;
- Az elemi műveletek végrehajtásához szükséges *vezérlőjelek* generálása.

A vezérlőegységnek egy általános modelljét mutatja a következő ábra:



A vezérlőegység bemenetei:

- **Órajel** – minden órajelütemben egy-egy elemi művelet vagy egyidejű müveletcsoport kerül végrehajtásra;
- **Utasításregiszter** – az aktuális utasítás műveleti kódja határozza meg, hogy milyen elemi lépéseket kell elvégezni a végrehajtási ciklusban;

- **Állapotjelzők** – megadják a processzor állapotát (hol tart az utasítás-cikluson belül) és az előző ALU-művelet kimenetét;
- **Vezérlőjelek a vezérlőbuszról** – külső egységek megszakításait és visszajelzéseit tartalmazzák.

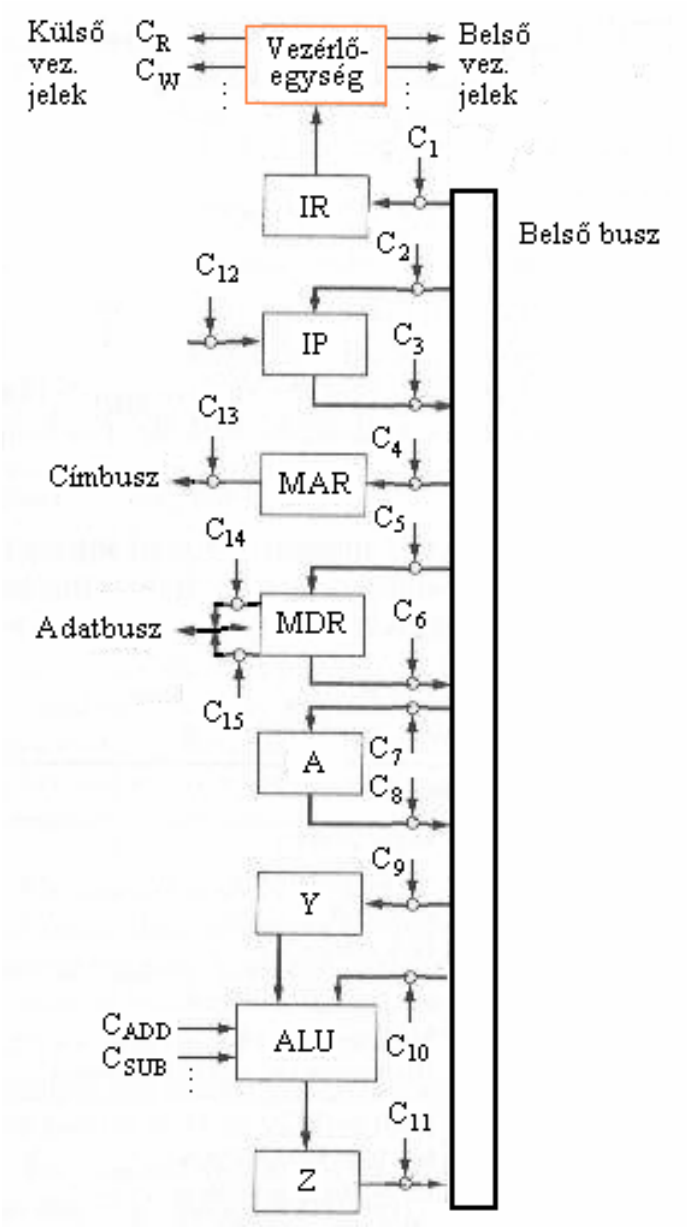
A vezérlőegység kimenetei:

- **Belső vezérlőjelek** – a regiszterek közötti adatátvitelt vezérlik, illetve az ALU műveletét határozzák meg;
- **Külső vezérlőjelek** – a memóriát és az I/O-portokat vezérlik.

Minden órajelütemben a vezérlőegység beolvassa a bemeneteket, és kiadja a megfelelő elemi művelet végrehajtását biztosító **vezérlőjeleket**. A vezérlőjelek három típusát különböztetjük meg:

- az ALU egyik műveletét aktiválja;
- egy belső adatútvonalat engedélyez;
- a rendszerbuszon küld ki egy vezérlést.

Végső soron mindegyik vezérlőjel egy-egy logikai kapu bemenetére kerül, amelynek a megnyitásával történik meg egy adat regiszterbe való beírása vagy onnan való kiolvasása, valamint egy ALU-művelet kiváltása. Az alábbi ábra egy belső busz köré szervezett processzor felépítését mutatja a vezérlési pontokkal együtt:



Az ALU mellett két – eddig nem tárgyalt – Y-nal és Z-vel jelölt regiszter jelenik meg. Egy kétoperandusú művelet esetében az Y regiszter tárolja ideiglenesen az egyik operandust (a másik a belső adatbuszról érkezik). Mivel az aritmetikai egység kombinációs áramkör, a kimenetét nem lehet közvetlenül a

buszra kapcsolni, mert ez visszacsatolást hozna létre. Ennek megakadályozása végett van szükség a Z regiszterre, amely az eredményt ideiglenesen tárolja. Ezek a regiszterek nem állnak a programozó rendelkezésére.

Példaként, az alábbi táblázat két alciklus elemi műveleteinek a leírását, valamint a megfelelő vezérlési jeleket adja meg:

Alciklus	**Időzítés**	**Aktív vezérlőjelek**
Lehívás	T_1: MAR ← (IP) T_2: Memória ← (MAR) T_2: MDR ← Memória IP ← (IP) +1 T_3: IR ← (MDR)	C_3, C_4 C_{13}, C_R C_{12}, C_{15} C_1, C_6
Összeadás	T_1: Y ← (MDR) T_2: Z ← (A) + (Y) T_3: A ← (Z)	C_6, C_9 C_8, C_{10}, C_{ADD} C_7, C_{11}

Az utasítás-lehívási ciklus mindig hasonló módon zajlik, de egy-egy utasításnak a végrehajtási ciklusa más-más elemi műveletsorozatot igényel, amit a végrehajtó egység az utasítás műveleti jelrészének a vizsgálata alapján határoz meg.

Az elemi műveletek sorozatának időegységekhez való hozzárendelésénél *két szabályt* kell figyelembe venni:

- az elemi lépéseknek az események helyes sorrendjét kell követniük (például a címküldés meg kell előzze a memóriából való adat beolvasását);
- a konfliktusokat el kell kerülni (például nem lehet egy időben ugyanabba a regiszterbe írni és onnan olvasni).

A vezérlőegység implementálására alapvetően két technikát alkalmaznak:

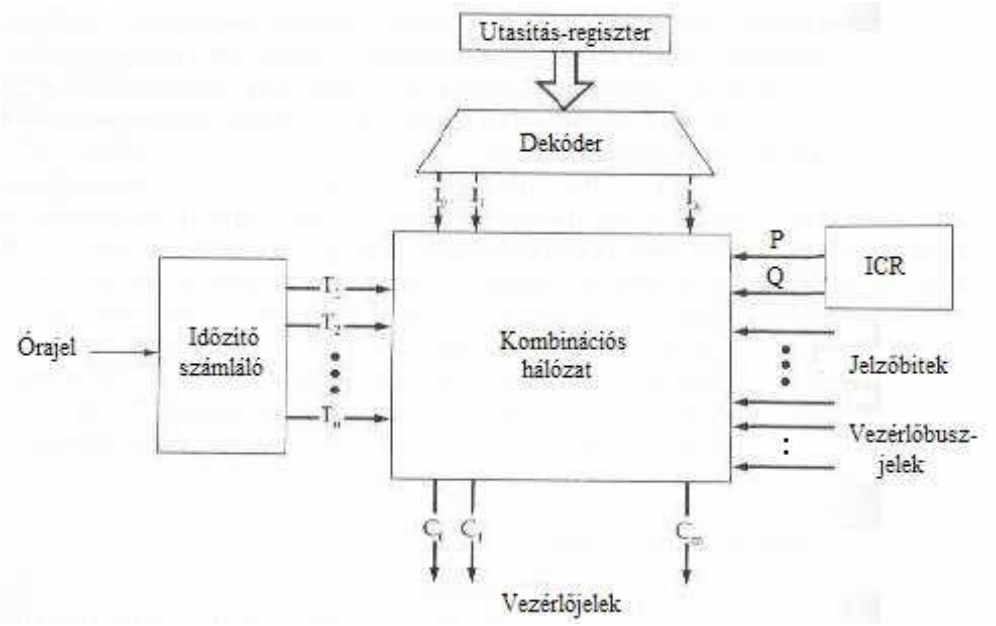
- **Huzalozott** vezérlőegység;
- **Mikroprogramozott** vezérlőegység.

A huzalozott vezérlőegység fő előnye a nagyobb sebességben van, de a mikroprogramozott vezérlőegységnek egyszerűbb az áramköri megvalósítása.

Ezért a CISC processzoroknál a mikroprogramozott megoldást alkalmazzák, szemben a RISC processzorok egyszerűbb utasításai számára alkalmasabb huzalozott technikával.

4.2 Huzalozott vezérlőegység

A huzalozott vezérlőegység központi eleme egy kombinációs hálózat, amely a végrehajtandó utasítás által meghatározott bemeneteket alakítja át vezérlőjelekké a kimenetein. Egy ilyen vezérlőegység felépítését mutatja az alábbi ábra:

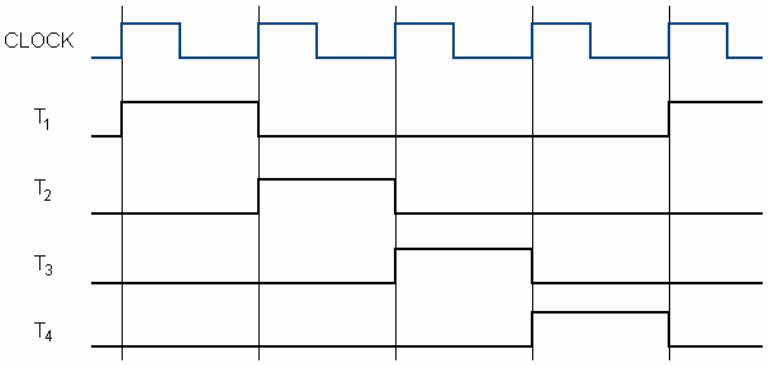


A vezérlőegység a végrehajtási ciklusban az **utasítás-regiszterben** lévő utasítás *műveleti kódja alapján* kell kibocsásson különböző vezérléseket. Az implementálás egyszerűsítése érdekében minden műveleti kód esetében a kombinációs hálózat egy-egy külön bemenetét kell aktiválni. Ezt a feladatot látja el a **dekóder** áramkör, amely minden bemeneti kód számára egy kimeneti jelet állít elő. Egy n bemenetű dekódernek 2^n kimenete van, amelyből csak az az egy aktív, amelyet a bemeneti bitkombinációval választottak ki.

Példaként egy 3 bemenetű és 8 kimenetű dekóder igazságtáblázatát mutatja a következő ábra:

I2	I1	I0	O0	O1	O2	O3	O4	O5	O6	O7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

A periodikus órajel alapján egy **számláló** állítja elő az elemi műveletek sorrendjét meghatározó T_1, T_2 stb. impulzusokat:

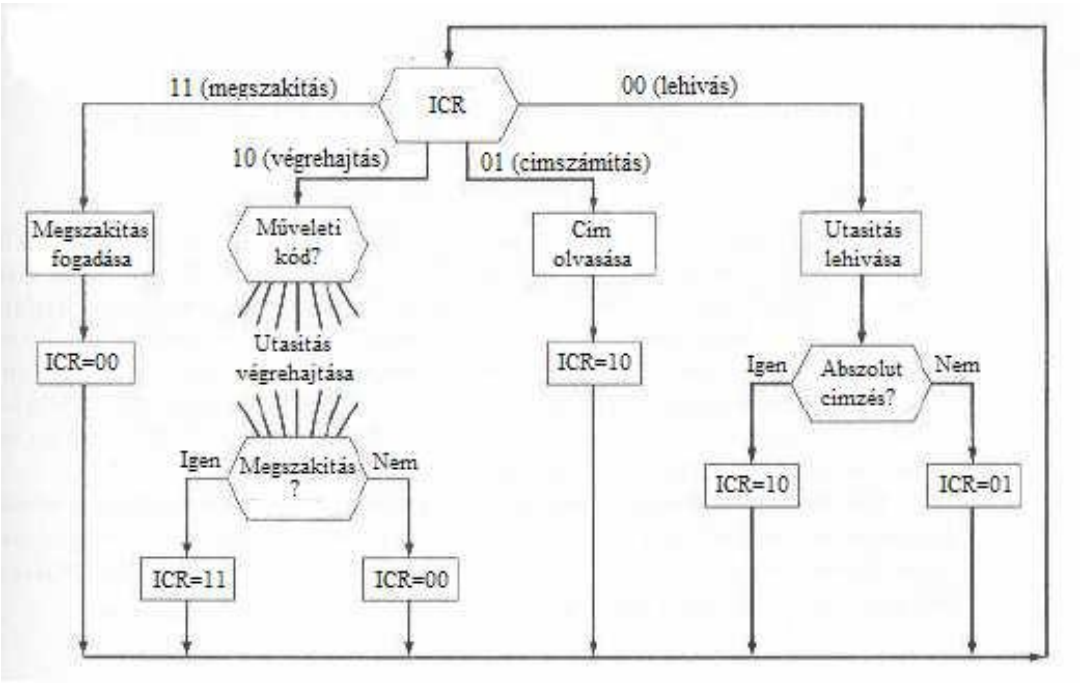


Minden utasításciklus végén a számlálót vissza kell állítani a kezdeti állapotba, hogy a T_1-et generálja.

Az utasítás lefolyása szempontjából helyes vezérlőjelek előállításához szükséges ismerni, hogy az utasításciklus melyik alciklusa következik. Ezt a célt szolgálja az ábrán az **utasításciklus fázisregisztere** (ICP- *Instruction Cycle Phase*). Az egy egyszerű esetben lehet egy kétbites (P és Q) regiszter, melynek állapotai például az alábbi alciklusoknak felelnek meg:

PQ = 00	Utasításlehívási ciklus
PQ = 01	Címszámítási ciklus
PQ = 10	Végrehajtási ciklus
PQ = 11	Megszakítási ciklus

A példaként használt ICP működését egy folyamatábrával lehet leírni:



Minden alciklus után a vezérlőegységnek elő kell állítani az ICP megfelelő állapotát, amely meghatározza a következő végrehajtandó alciklust.

Miután minden alciklus számára elkészül a szükséges elemi műveletek időrendi specifikálása, fel lehet írni a Boole-algebra segítségével a **vezérlőjelek egyenleteit**. Például, ha a C_3 vezérlőjelet aktiválni kell minden lehívási és a címszámítási ciklus második időegységében, valamint a MOV, ADD és AND

utasítások végrehajtási ciklusának harmadik időegységében, az egyenlete a következő lesz:

$$C3 = P \cdot Q \cdot T2 + P \cdot Q \cdot T2 + P \cdot \overline{Q} \cdot (MOV + ADD + AND) \cdot T3$$

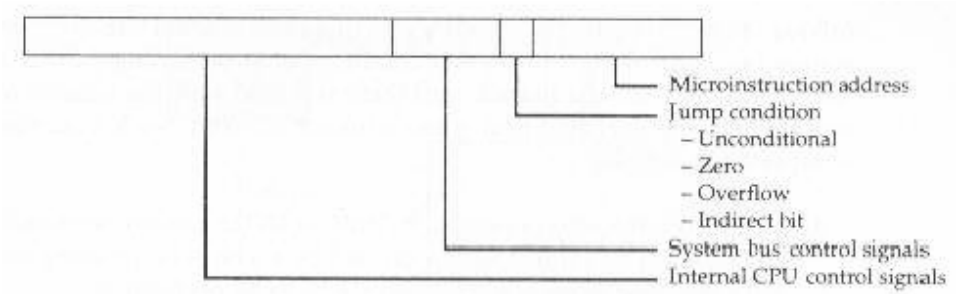
Miután az összes jel számára felírtuk a függvényeket, el lehet végezni ezek alapján a **kombinációs hálózat** logikai kapukkal való implementálását.

4.3 Mikroprogramozott vezérlőegység

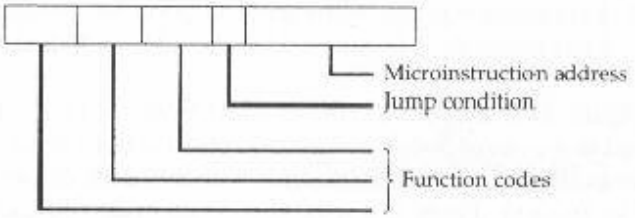
A mikroprogramozott vezérlőegység logikai működését egy ***mikproprogram*** *valósítja meg, amely mikroutasításokból áll. Minden* ***mikroutasítás*** *egy elemi műveletet specifikál*, így a végrehajtása a művelethez tartozó vezérlőjelek aktiválását eredményezi.

A mikroutasítások kétféleképpen lehet megvalósítani (lásd az ábrát):

- *Horizontális* mikroutasítás – a mikroutasítás mindegyik bitje egy-egy vezérlőjel bináris alakját jelenti (ha „1”, akkor a vezérlőjel aktív, ha „0”, akkor nem);
- *Vertikális* mikroutasítás – a mikroutasítás a végrehajtandó elemi művelet kódját tartalmazza.

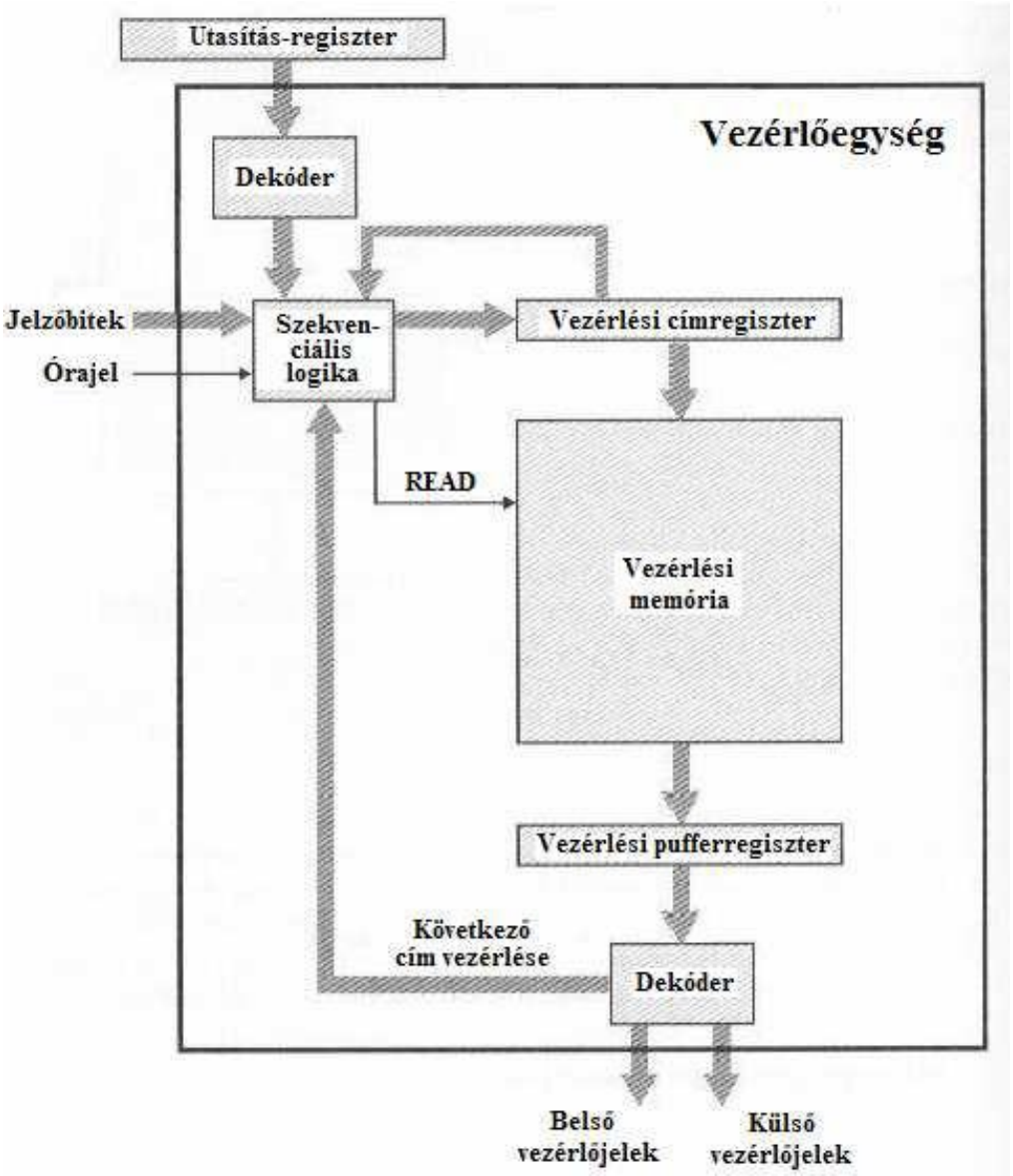


a) Horizontális mikroutasítás



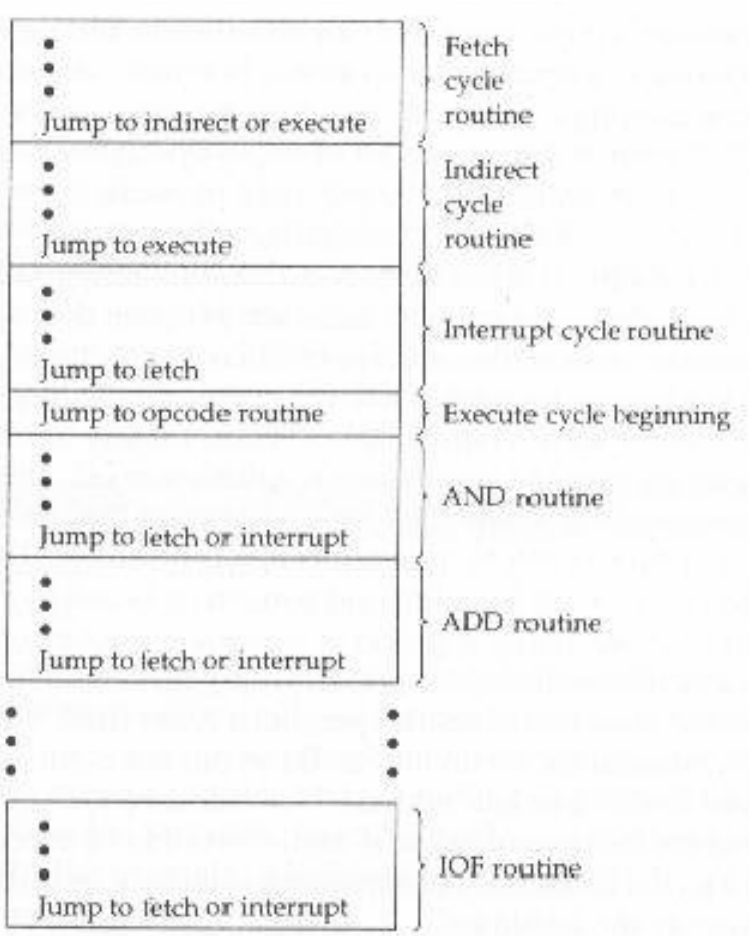
b) Vertikális mikroutasítás

A mikroprogramozott vezérlőegység elvi felépítése a következő:



A *vezérlési memória* egy ROM áramkör, amely a mikroutasítások sorozatát tárolja, szubrutinokba rendezve, amint az alábbi ábra mutatja. Mindegyik alciklusnak egy szubrutin felel meg, a végén egy ugró utasítással a következő végrehajtandó rutinra. A végrehajtási ciklus elején van egy speciális rutin,

amely az aktuális műveleti kód alapján választja ki a gépi szintű utasításnak megfelelő rutint (ADD, AND, MOV stb.).



A következő kiolvasandó mikroutasítás címét a *vezérlési címregiszter* tartalmazza. A kiolvasott mikroutasítás a *vezérlési pufferregiszterbe* kerül. A kimenetén lévő dekóderre csak a vertikális mikroutasítások esetében van szükség, mert a horizontális mikroutasítások már a vezérlőjeleknek megfelelő biteket tartalmazzák. A másik, az utasításregiszter kimenetén lévő dekóder, a gépi szintű utasítás műveleti kódját alakítja át vezérlési memóriabeli címmé.

A vezérlőegység működése *minden órajelütemben* a következő:

1. A szekvenciális logika kibocsátja az olvasási parancsot (READ).
2. A megcímzett mikroutasítás bekerül a pufferregisztrerbe.
3. A pufferregiszter tartalma előállítja a vezérlőjeleket és a következő mikroutasításra vonatkozó adatot.
4. A következő mikroutasításra vonatkozó adat és az FLAGS regiszter állapotjelzői alapján a szekvenciális logika meghatározza a következő mikroutasítás címét, a következő lehetőségekből:
    - a vezérlési címregiszter taralmát +1-gyel megnöveli,
    - az ugró (JUMP) mikroutasításban megadott címet tölti be, ezzel egy másik rutinra ugorva;
    - az utasításregiszterben lévő műveleti kódnak megfelelő rutin címét tölti be, elindítva ennek az utasításnak a végrehajtási alciklusát.

A mikroprogramozott vezérlőegységnek két alapfeladatot kell ellátni:

- A mikroutasítások *szekvenciálását*, azaz meghatározni a következő mikroutasítás címét;
- A mikroutasítások *végrehajtását*, vagyis generálni a szükséges vezérlőjeleket.

Mindkét feladat megoldása szoros összefüggésben van a mikroutasítások szerkezetével.

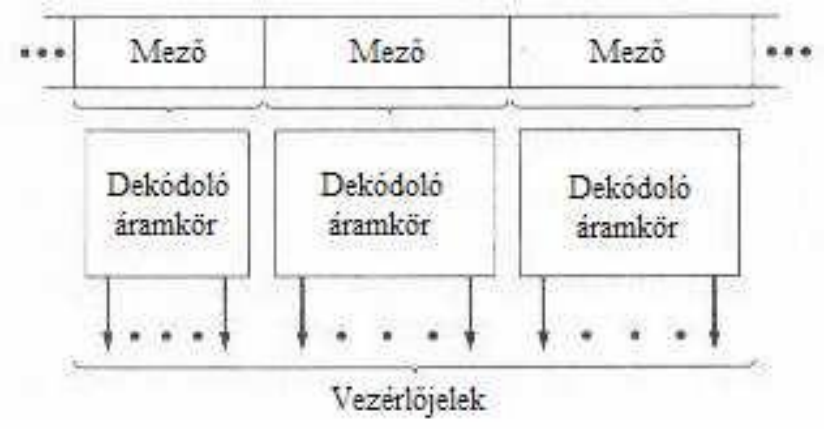
4.4 A mikroutasítások kódolása

A tiszta horizontális, illetve vertikális utasításszerkezet két végletet jelent: ez egyiknél minden vezérlőjelet egy külön bit jellemez, a másikra egy magas fokon becsomagolt formátum a jellemző. A gyakorlatban minden vezérlőegység tervezésénél valamilyen fokig szokás a mikroutasításokat kódolni, hogy csökkenjen a vezérlőmemória szélessége, és egyszerűsödjön a mikroprogramozás feladata.

A kódolt mikroutasítások tervezésének menete a következő:

1. Szervezzük meg a szerkezetet mezőkre bontva úgy, hogy a mezőkre jellemző műveletek vezérlőjelei egyszerre is aktiválódhassanak. Így N mezővel N egyidejű eseményt lehet meghatározni.
2. Határozzuk meg minden mezőben a specifikálható, egymást kizáró műveleteket úgy, hogy azokból egyszerre csak egy lehessen aktív. Ha egy mező M bitet tartalmaz, akkor ezzel maximum 2^M egymást kizáró eseményt lehet kódolni.

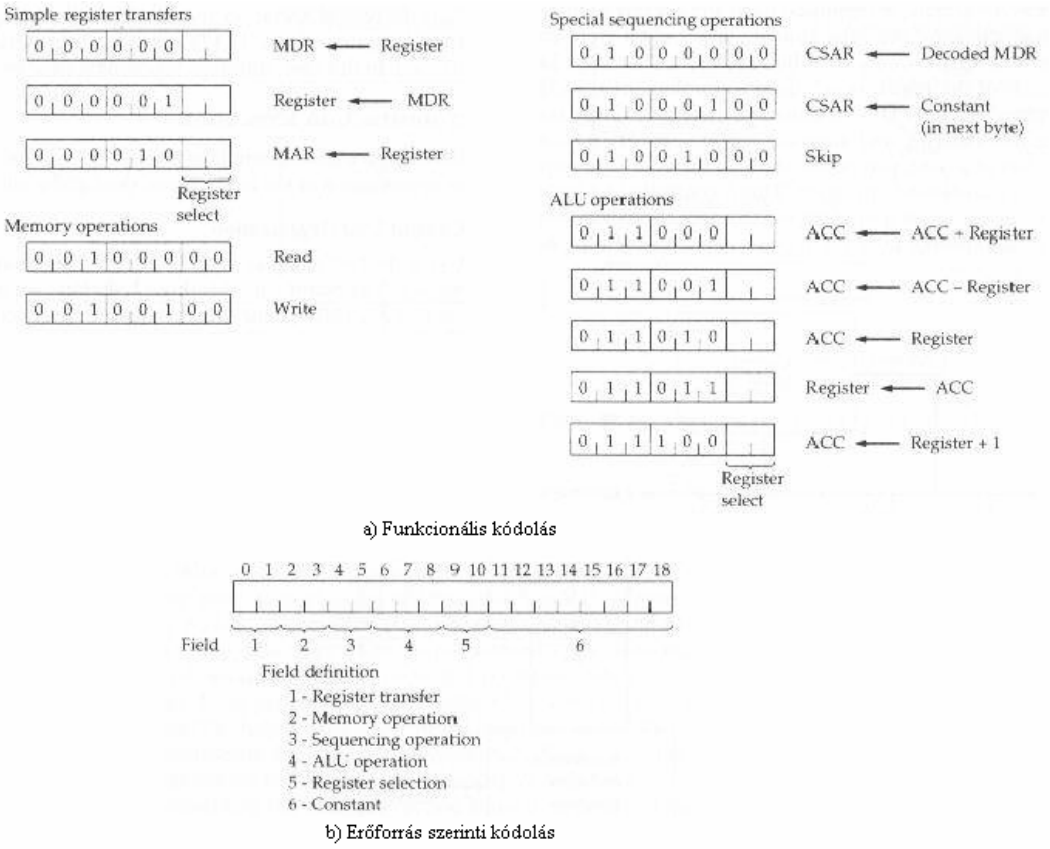
A létrejött mikroutasítások mindegyik mezője tartalmaz egy kódot, amelyet dekódolva létrejönnek a vezérlőjelek:



A mikroutasítások kódolásának két technikája van:

- **Funkcionális kódolás** – a gép minden egyes funkciójának egy-egy mezőt tartunk fel;
- **Erőforrás szerinti kódolás** – a gép minden erőforrásához rendelünk egy-egy mezőt.

Erre a két technikára mutat példákat az alábbi ábra:

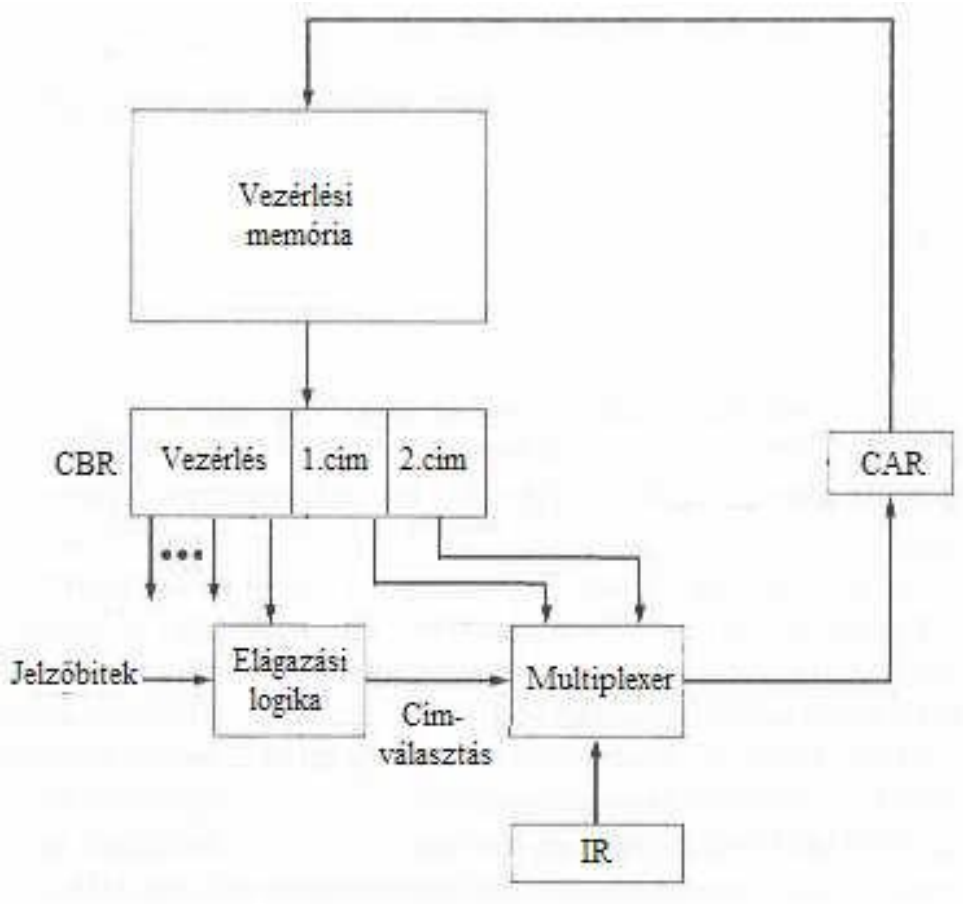


4.5 A mikroutasítások szekvenciálása

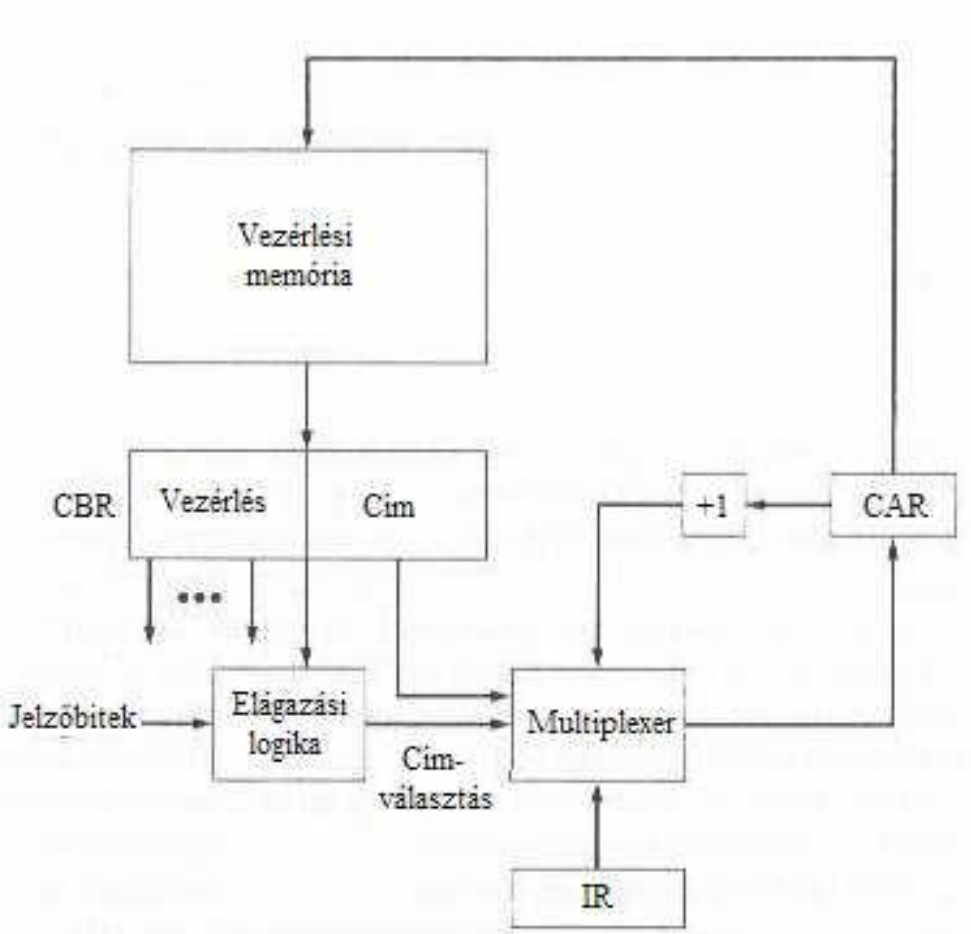
A következő végrehajtandó mikroutasítás számára egy vezérlőmemóriabeli címet kell előállítani az aktuális mikroutasítás, a jelzőbitek és az utasítás-regiszter tartalma alapján. A mikroutasítás szerkezetében lévő címinformációk alapján, az erre használt technikák három csoportba foglalhatók:

- Két címmezős szerkezet;
- Egy címmezős szerkezet;
- Változó szerkezet.

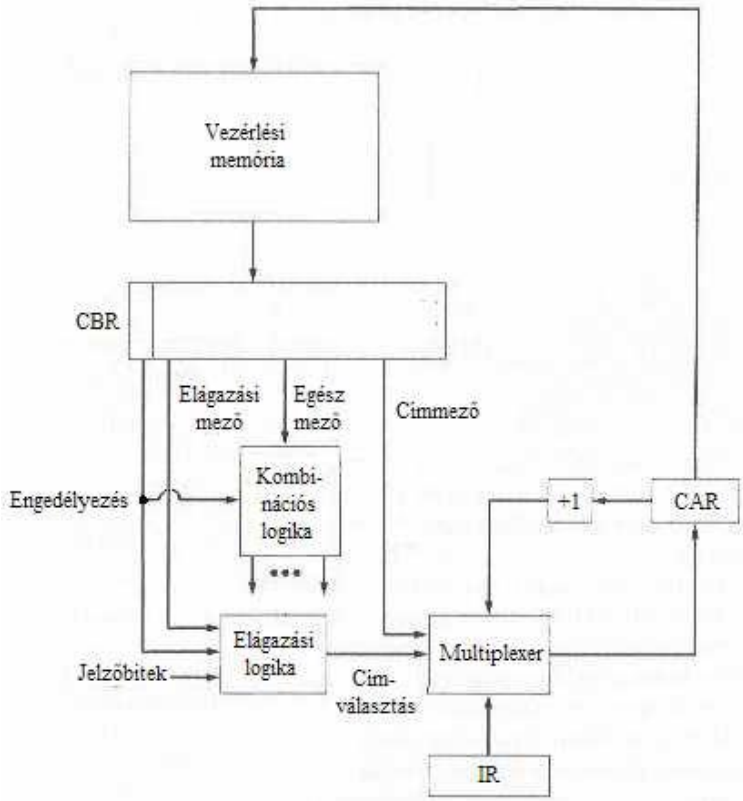
A legegyszerűbb megoldás a **kétcímes szerkezet** (lásd az alábbi ábrát), de hátránya a hosszú mikroutasítás. Az egyik cím a következő mikroutasítás címe, a másik a feltételes elágazásnál érvényes cím. Egy multiplexáló áramkör továbbítja a vezérlési címregiszterbe a műveleti kódot vagy a két cím egyikét. A kiválasztást az elágazási logika végzi el, a mikroutasítás vezérlőrésze és a jelzőbitek alapján.



A takarékosabb, **egycímes szerkezetnél**, a soron következő utasítás címét – ugrás hiányában – az aktuális utasítás címének inkrementálásával kapjuk meg:



Az előző megoldás sem a leghatékonyabb, mert a mikroutasításban fenntartott címmezőre csak az ugrási utasításokban van szükség. A **változó szerkezet** esetében, a mikroutasitás egy külön bitje határozza meg, hogy melyik formátumról van szó: az egyik formátumban a fennmaradó bitek a vezérlőjeleket aktiválják, a másikban egy részük az elágazási logikát vezérli, a többi az ugrási címet tartalmazza (ez utóbbi esetben egy órajelütemre kimarad a vezérlőjelek generálása):



Az eddig bemutatott **explicit** technikákon kívül léteznek **implicit** címgenerálási technikák is:

- A gépi utasítás kódja jelenti a mikroutasítás címét;
- A mikroutasítás címmezőjének részein végrehajtott művelettel (például összeadással) állítjuk elő a következő mikroutasítás címét;
- Egy mikroszubrutin végén a visszatérési címet egy – a vezérlőegységen belüli – regiszterből vagy veremből olvassuk ki.

A mikroprogramozott módon implementált vezérlőegység lehetővé teszi az **emulációt**. Ez alatt azt értjük, hogy egy számítógépen – megfelelő mikroporgram segítségével –, *lehetővé tesszük egy másig gépre írt, annak utasításait használó program végrehajtását.*

KÉRDÉSEK, TÉTELEK

4.1. Melyek a vezérlőegység alapfeladatai?

4.2. Vázolja fel a huzalozott vezérlőegység felépítését, és magyarázza el a működését.

4.3. Mi a különbség a horizontális és a vertikális mikroutasítás között? Mutassa be egy ábrán!

4.4. Vázolja fel a mikroprogramozott vezérlőegység felépítését, és magyarázza el a működését.

4.5. Mi a kódolt mikroutasítások tervezésének a menete? Milyen fajta kódolást alkalmazhatunk?

4.6. Hogy határozzuk meg a következő mikroutasítás címét különböző szerkezetű mikroutasítások (egy címmezős, két címmezős, változó formátumú) esetében?

5. MEMÓRIA-HIERARCHIA TERVEZÉSI KÉRDÉSEI.

5.1 A központi tár címzése

A központi tárhoz való hozzáférés érdekében a processzor az illető rekesz címét küldi ki a címbuszon. A címbusz szélessége meghatározza a közvetlenül megcímezhető rekeszek számát, azaz a **memória címtartományát** (m bites cím esetében a címtartomány mérete 2^m). A mikroszámítógépekben minden rekesz **8 bitet (1 bájtot)** tartalmaz.

A címtartomány egy részét – nem feltétlenül az egészet – foglalják el a rendszerbe fizikailag bekötött memória-áramkörök. Ebben a fizikailag létező tárban egy memóriarekesz megcímzése két feladatot feltételez:

- az illető rekeszt tartalmazó ***csip*** *kiválasztását*
- a ***rekesz*** *kiválasztását* a csipen belül.

A csipen belüli rekesz kiválasztása a címbusz legkisebb helyiértékű vonalaival történik, amelyeket az áramkör címbemeneteire kell kötni. Ezeknek a számát az illető memória-áramkör kapacitása határozza meg. Például egy 1 KB-os áramkörnek 10 címbemenete van (2^{10} = 1 K).

A címbusz többi vonalát a memóriacsipek kiválasztására használjuk. Ezek határozzák meg az áramkörök által elfoglalt fizikai memória helyét a címtartományon belül.

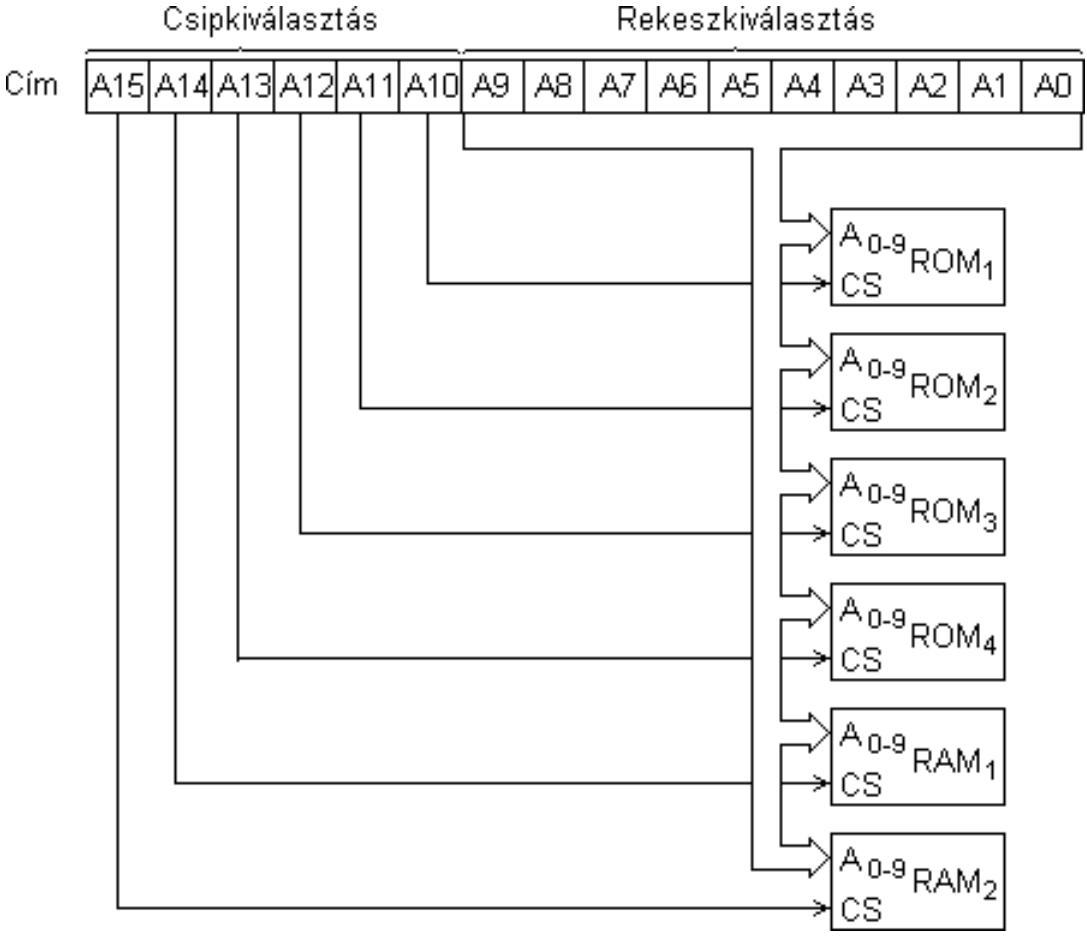
A csipkiválasztásnak két technikája van:

- **lineáris** szelekció
- **dekódolt** szelekció

a) Lineáris szelekció

Ebben az esetben ***minden áramkör kiválasztására egy-egy külön címvonalat használunk fel***. Ezt az egyszerű módszert csak kevés memóriaáramkört tartalmazó rendszerekben tudjuk használni, ahol a megmaradt magasabb helyiértékű címvonalak száma nem kisebb az áramkörök számánál.

Az alábbi példa egy 16 bites címbusszal rendelkező rendszert mutat be, amelyben 1 KB-os memóriacsipek vannak. Ezeknek az A_0–A_9-es címbemenetei szolgálnak az áramkörön belül egy rekesz kiválasztására, míg a CS (*Chip Select*) bemenetnek a feladata az áramkör kiválasztása:



A fizikai memóriaterület elhelyezkedését a processzor címtartományában **memóriatérképnek** (*memory map*) hívjuk. A fenti példa esetében a memóriatérkép az alábbi ábrán látható (a címek **16-os** számrendszerben vannak megadva):

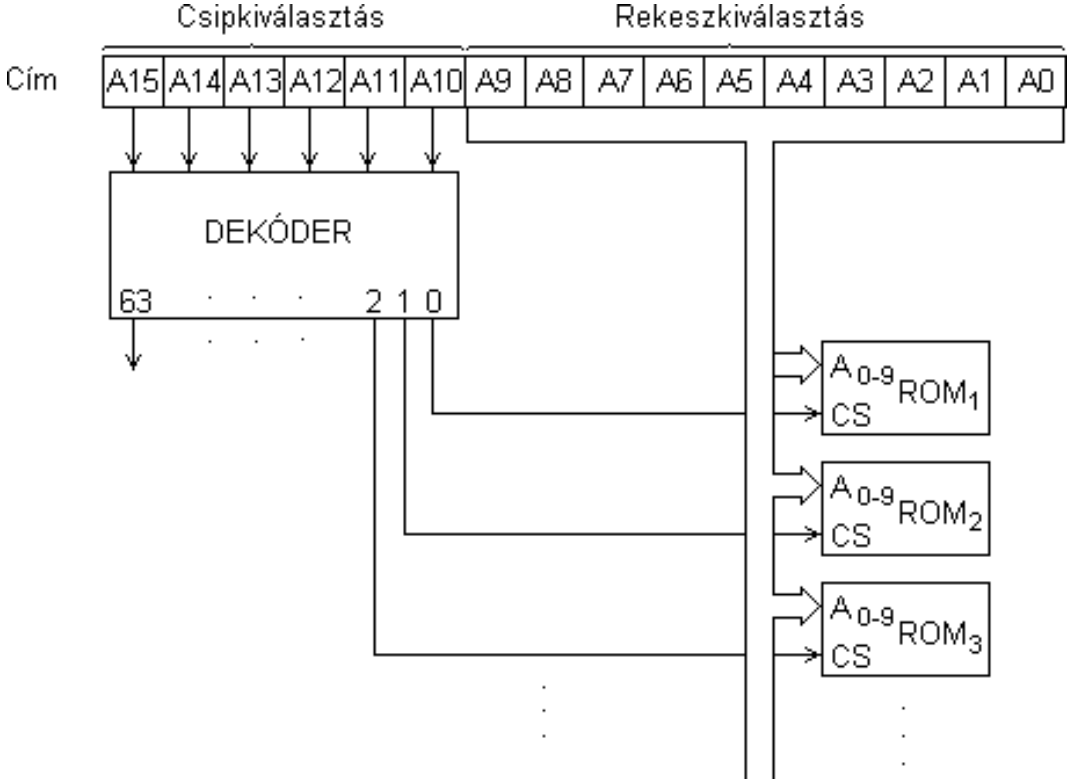
Cím	**Főtár**
$0000_H - 03FF_H$	1 K szabad
$0400_H - 07FF_H$	1 K ROM_1
$0800_H - 0BFF_H$	1 K ROM_2
$0C00_H - 0FFF_H$	1 K szabad
$1000_H - 13FF_H$	1 K ROM_3
$1400_H - 1FFF_H$	3 K szabad
$2000_H - 23FF_H$	1 K ROM_4
$2400_H - 3FFF_H$	7 K szabad
$4000_H - 43FF_H$	1 K RAM_1
$4400_H - 7FFF_H$	15 K szabad
$8000_H - 83FF_H$	1 K RAM_2
$8400_H - FFFF_H$	31 K szabad

Látható, hogy lineáris szelekcióval nem lehet kitölteni az egész címterületet, és töredezett memóriaterületet eredményez, ami ennek a módszernek a hátránya.

b) Dekódolt szelekció

Ebben az esetben a csipkiválasztás számára *megmaradt címvonalak dekódolásával* állítjuk elő a memória-áramkörök szelekciós jeleit. Ez a módszer lehetővé teszi az egész címtartomány kihasználását, valamint ezen belül folyamatos memóriaterületeket lehet létrehozni.

Az alábbi ábra az előző példát mutatja be dekódolt szelekció alkalmazásával. Az A10–A15 címbitek dekódolásával létrejön $2^6 = 64$ jel a csipek kiválasztására. Ezek a jelek a 64 KB-os címtartományt 1 KB-os zónákra bontják, amint az a memóriatérképen látható.



Cím	**Főtár**
$0000_H - 03FF_H$	1 K ROM1
$0400_H - 07FF_H$	1 K ROM2
$0800_H - 0BFF_H$	1 K ROM3
.	.
.	.
.	.

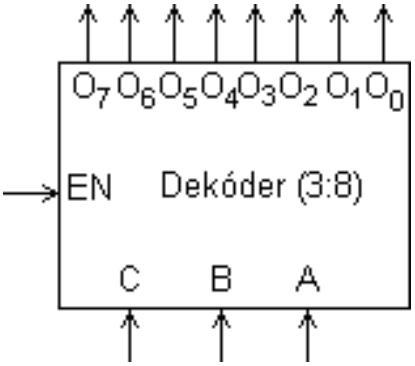
A dekódoló logikát meg lehet építeni különálló kapukból, vagy dekóder áramkör alkalmazásával. A *csipkiválasztó jelek egyenleteit*– az előbbi példánál maradva –a következőképpen írhatjuk fel:

$$CS_1 = \overline{A15} \cdot \overline{A14} \cdot \overline{A13} \cdot \overline{A12} \cdot \overline{A11} \cdot \overline{A10}$$

$$CS_2 = \overline{A15} \cdot \overline{A14} \cdot \overline{A13} \cdot \overline{A12} \cdot \overline{A11} \cdot A10$$

$$CS_3 = \overline{A15} \cdot \overline{A14} \cdot \overline{A13} \cdot \overline{A12} \cdot A11 \cdot \overline{A10}$$

Ezen egyenletek alapján készíthetjük el a dekódoló logikát, vagy használhatunk kész **dekódoló áramkört**. Egy ilyen 3 bemenetű és $2^3 = 8$ kimenetű áramkörnek a logikai vázlatát és a működési táblázatát tartalmazza az alábbi ábra. Az **EN** (*Enable*) bemenet a dekódolási funkció engedélyezését jelenti.



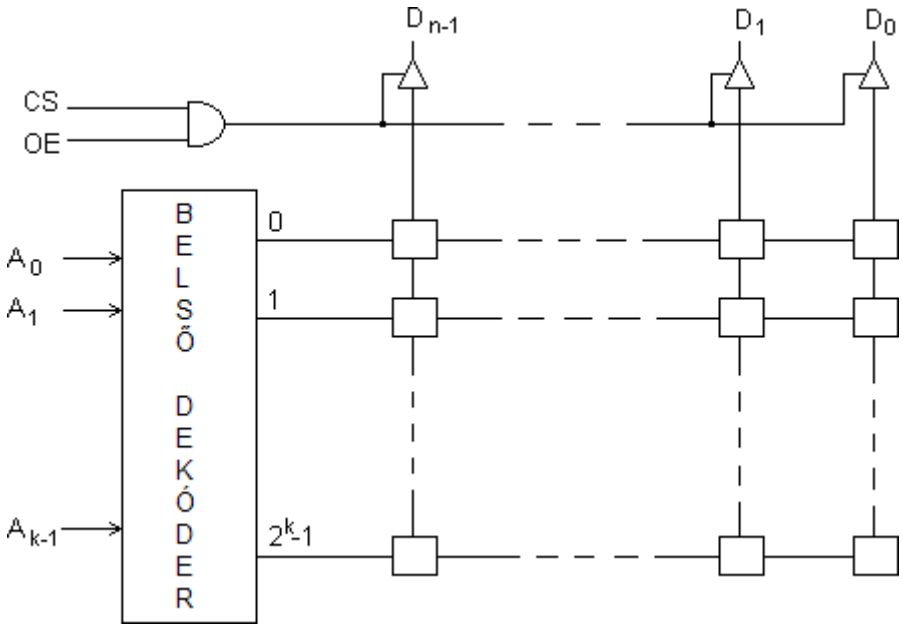
EN	**C**	**B**	**A**	**O_7**	**O_6**	**O_5**	**O_4**	**O_3**	**O_2**	**O_1**	**O_0**
0	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0

1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

5.2 A memória-áramkörök bekötésének elve

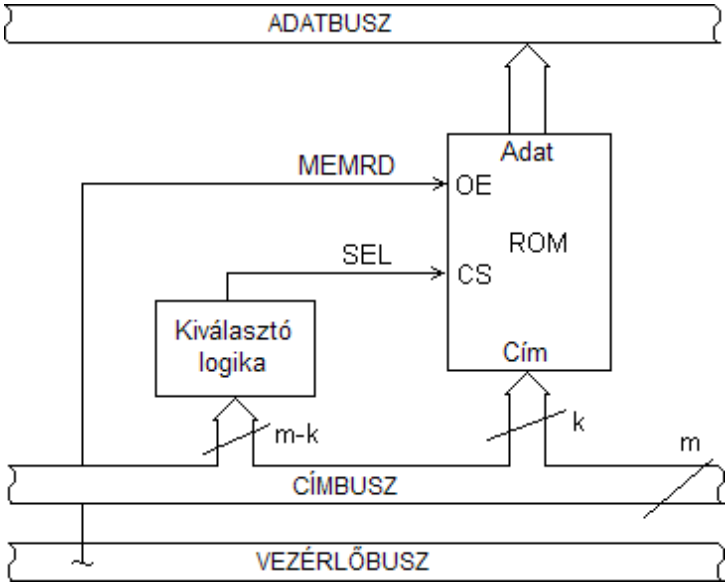
ROM memóriák

Egy ROM áramkör memóriacellái mátrix alakban vannak elrendezve a csip belsejében (lásd az ábrát). Minden egyes cella egy bitet tárol. A k a címbemenettel ($A_0 – A_{k-1}$) lehet aktiválni a mátrix egyik sorát, és így kiválasztani az egyiket az összesen 2^k n-bites rekesz közül. Az áramkör kapacitása tehát 2^k x n bit.



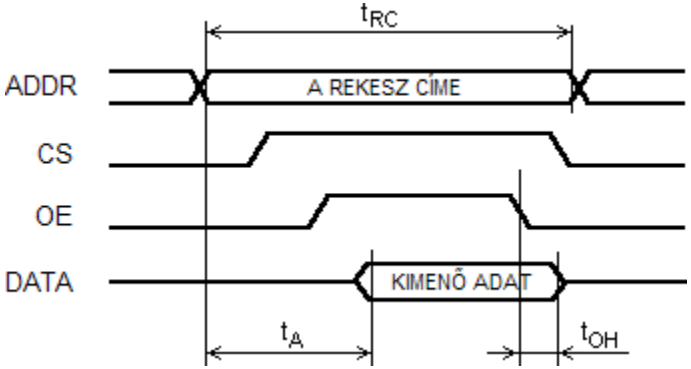
A mátrix oszlopai 3-állapotú puffer meghajtókon keresztül csatlakoznak az adatkimenetekre (D_0 – D_{n-1}). Ezáltal lehetővé válik a kimenetek közvetlen csatlakoztatása az adatbuszra. Az adat továbbítása a kimeneteket engedélyező OE (*Output Enable*) jel hatására történik, feltéve, hogy az áramkör szelektátva van (CS = 1).

Egy ROM áramkör elvi bekötését a rendszer buszaira a következő ábra szemlélteti:



Miután a processzor a címbuszra helyezte a kiolvasandó rekesz *m* bites címét, ez a cím kettéválik: az alsó *k* bit közvetlenül a 2^k kapacitású csip címbemeneteire kerül, míg a megmaradó bitekből a kiválasztó logika generálja a csip szelekciós jelét (SEL), a már ismert technikák egyikével (lineáris vagy dekódolt szelekció). A kiolvasott adatok a memóriaolvasási vezérlőjel (MEMRD) hatására kerülnek az adatbuszra.

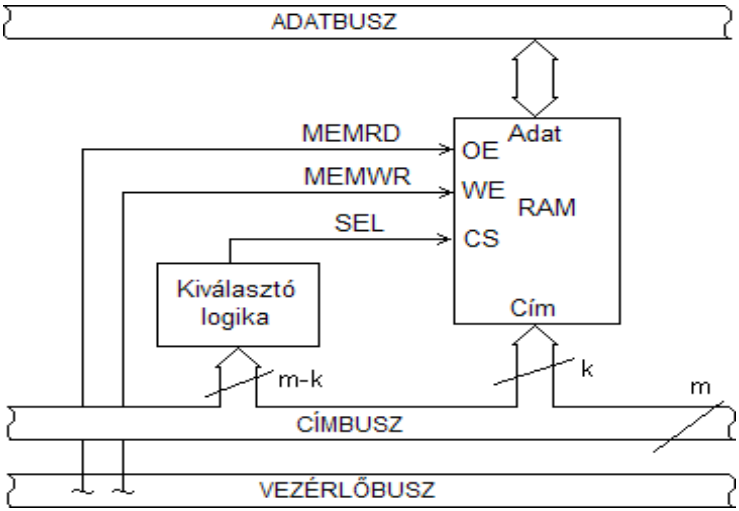
Minden memória-áramkör fontos jellemzője a **hozzáférési idő** (t_A – *access time*), amely a címnek a csip bemeneteire való helyezésének pillanatától az adatok kimeneteken való megjelenéséig tart. A hozzáférési időt az áramkör olvasási ciklusát leíró idődiagram mutatja:



Az olvasási ciklus ideje alatt (t_{RC} – *Read Cycle*) a címnek stabilan meg kell maradni a memória bemenetein. A kimenetek letiltása után az adat még egy tartási ideig (t_{OH} – *Output Hold*) elérhető, és így átvehető a processzor által az adatbuszról. A processzor működését egy olvasási ciklusban össze kell hangolni a használt memóriára jellemző olvasási ciklussal.

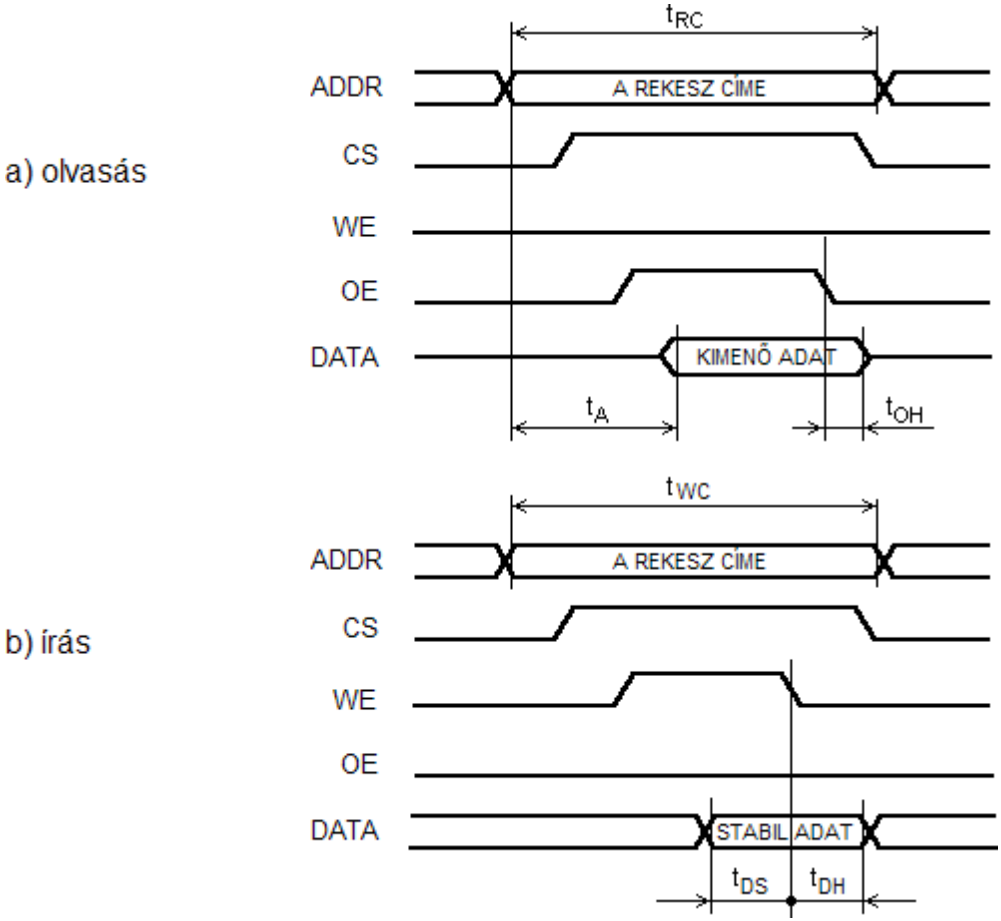
RAM memóriák

A statikus RAM áramkör buszrendszerhez kapcsolását mutatja a következő ábra:



A bekötés módja hasonló a ROM áramköréhez, de itt van még egy vezérlőbemenet, amely az írást teszi lehetővé (**WE** – *Write Enable*). Erre a bemenetre a processzortól jövő memóriaírási vezérlőjel (MEMWR) kerül, míg a kimenetek megnyitása az olvasási ciklusban történik a MEMRD jellel.

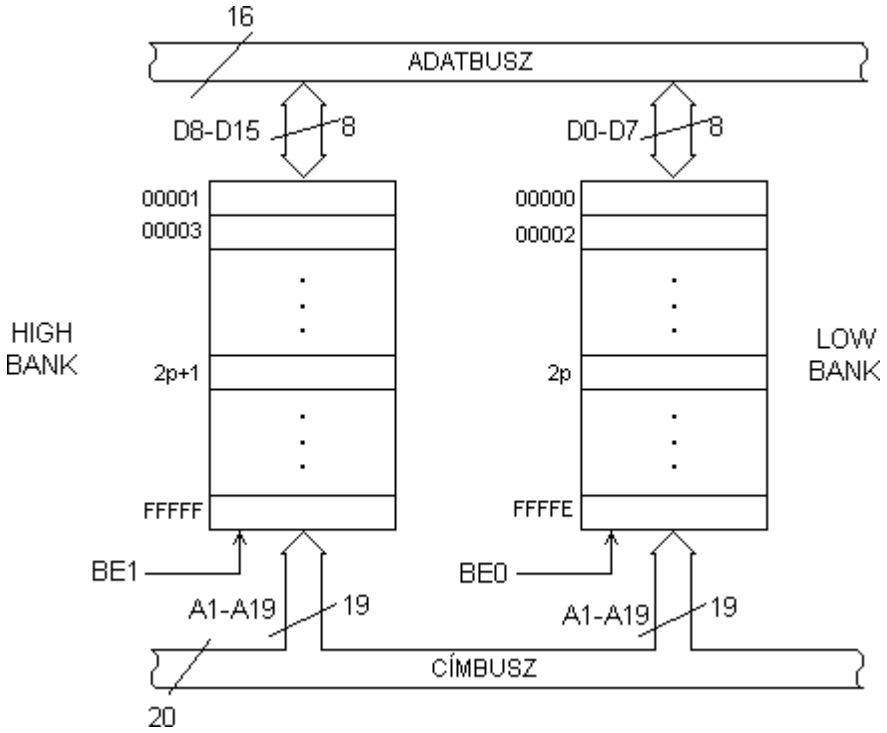
A helyes működést a memória-áramkör idődiagramjai mutatják:



Az írási ciklusban az adat a processzortól érkezik az adatbuszon. Ez stabil kell maradjon a WE jel hátsó éle körül, mert ekkor történik az adat beírása a memóriába. A csip katalógusában megadják az **előkészítési idő** (t_{DS} – *Data Setup*) és a **tartási idő** (t_{DH} – *Data Hold*) minimálisan szükséges értékeit.

A memória-áramkörök bekötésének sajátosságai széles adatbusszal rendelkező rendszerek esetében

Egy 16 bites adatbusszal rendelkező processzor esetében lehetőség van egyszerre két bájt átvitelére a memóriából. Ennek feltétele, hogy a memóriát két tömbben szervezzük meg, amelyek mindegyike az adatbusz egy-egy 8 bites részéhez kapcsolódik:

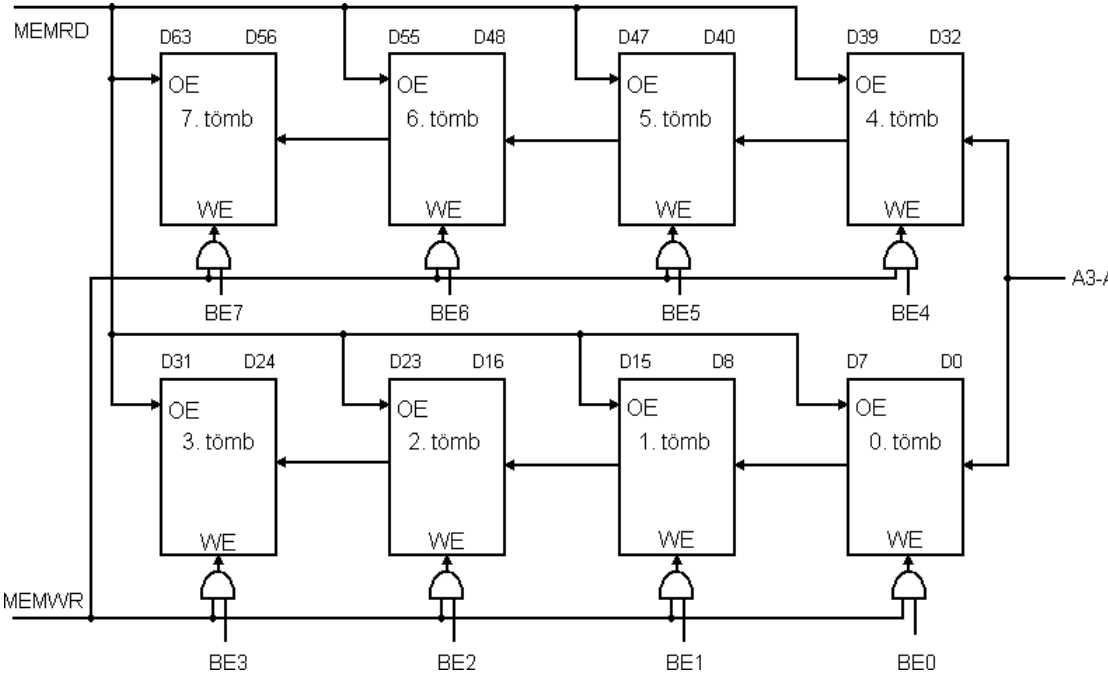


Az alsó tömb (*low bank*) tartalmazza a páros című rekeszeket (A0 = 0), és az adatbusz D0–D7 vonalaihoz csatlakozik. A páratlan című rekeszek (A0 = 1) a felső tömbben (*high bank*) vannak, amelyik az adatbusz D8–D15 vonalaihoz csatlakozik. A tömbök kiválasztására a processzor két **bájtkiválasztó jelet** (BE – *Byte Enable*) küld, ezért az A0 címvonalra nincs is szükség. Három fajta adatátvitel lehetséges:

- 1 bájt átvitele páros címről a D0–D7-en – ekkor BE1 = 0, BE0 = 1
- 1 bájt átvitele páratlan címről a D8–D15-ön – ekkor BE1 = 1, BE0 = 0
- 2 bájt átvitele páros címtől kezdődően a D0–D15-ön – ekkor BE1 = 1, BE0 = 1

Az előbbi elvet lehet általánosítani 32, illetve 64 bites adatbuszok esetére is. A 64 bites adatbusszal rendelkező rendszerben 8 tömböt kell kialakítani,

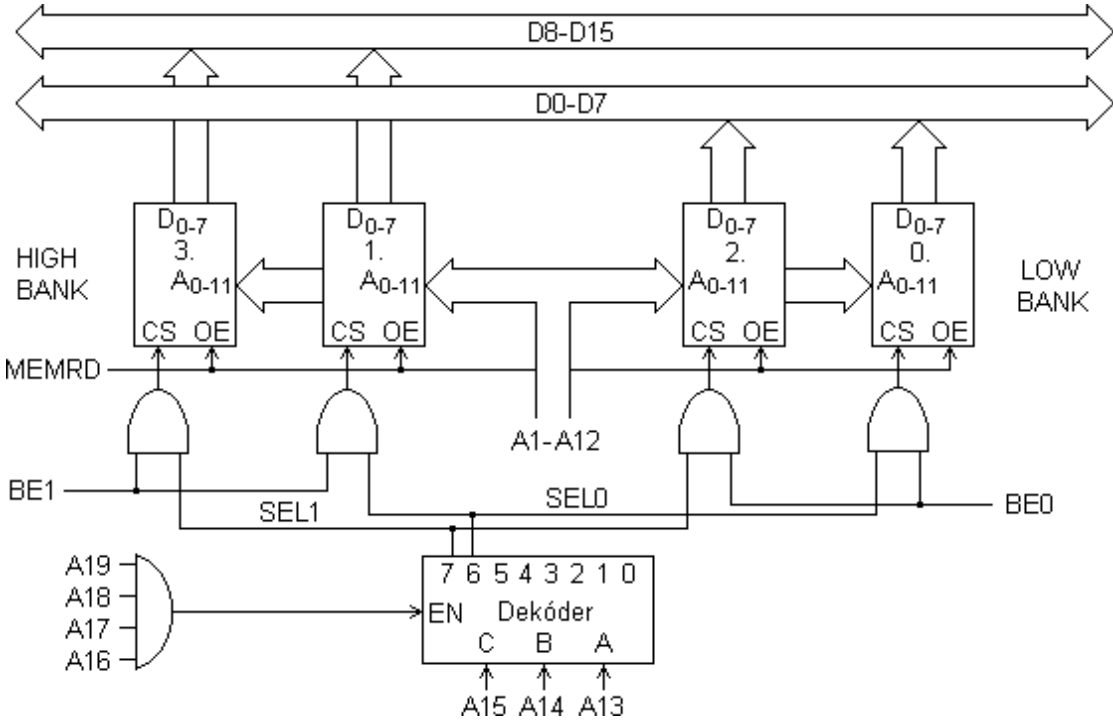
amint az alábbi ábra mutatja. Itt 8 bájtkiválasztó jel van, ezért az A0–A3 címbitekre nincs szükség (ezeket a processzor belsőleg használja a bájtkiválasztó jelek generálására).



Az előző ábrán figyeljük meg, hogy a bájtkiválasztó jelek az írás engedélyezésében vesznek részt, mivel az olvasás nem destruktív (egyszerre ki lehet olvasni a címterületen szomszédos címen lévő, de mégis külön tömbben helyet foglaló adatokat, még ha a processzornak nincs is mind a 8 bájtra szüksége – írásnál viszont ezt már nem szabadna megtenni).

ROM memóriablokk

Az alábbi példa egy 16 KB-os ROM memóriablokkot mutat be, egy 16 bites adatbusszal és 20 bites címbusszal rendelkező rendszer számára. A felhasznált csipek szervezése 4K × 8 bit.



A memóriablokk két tömbbe van szervezve, az egyik a busz D0–D7, a másik a D8– D15 vonalaira van kötve. A tömbök kiválasztását a BE0 és BE1 jelek végzik. A memóriablokk által elfoglalt címterületet az alábbi táblázat mutatja, amely az áramkör párok címhatárait tartalmazza:

A19	A18	A17	A16	A15	A14	A13	A12	...	A1	A0	Csip	Szelekció	Címterület
1	1	1	1	1	1	0	0	...	0	0	0. és 1.	SEL0=1	$FC000_H$÷$FDFF_H$
1	1	1	1	1	1	0	1	...	1	1			
1	1	1	1	1	1	1	0	...	0	0	2. és 3.	SEL1=1	$FE000_H$÷$FFFF_H$
1	1	1	1	1	1	1	1	...	1	1			

A páros címeket (A0=0) tartalmazó csipek az alsó tömbben, a páratlant (A0=1) tartalmazók a felsőben vannak. A csipeken belüli rekeszek kiválasztására az A1–A12 címbiteket használjuk, a megmaradt A13–A19 biteket dekódolva állítjuk elő a táblázat sorainak megfelelő SEL0 és SEL1 jeleket. A csipkiválasztó jeleket a következő egyenletek adják meg:

$$CS_0 = SEL0 \cdot BE0$$

$$CS_1 = SEL0 \cdot BE1$$

$$CS_2 = SEL1 \cdot BE0$$

$$CS_3 = SEL1 \cdot BE1$$

(Meg kell jegyezni, hogy mivel csak olvasható memóriáról van szó, nem feltétlenül lenne szükséges a bájtkiválasztó jelek használata.)

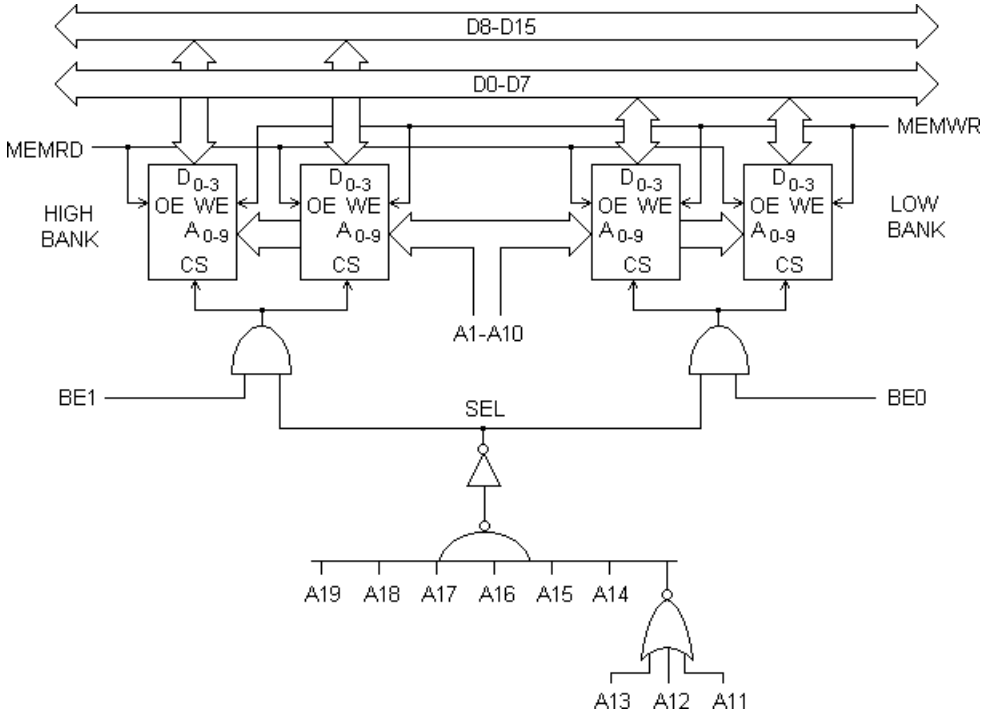
A dekóder bemeneteire kerülnek az A13–A15 vonalak, a további bitek (amelyeknek az értéke változatlan) biztosítják a dekóder engedélyezését:

$$EN = A19 \cdot A18 \cdot A17 \cdot A16$$

A kimeneteket az olvasási ciklusban aktiválódó MEMRD (*Memory Read*) vezérlőjel nyitja meg.

RAM memóriablokk

A következő példa egy 2 KB-os statikus RAM memóriablokkot ábrázol, 1K × 4 bites áramkörökből. Az adatbusz 16 bites, a címbusz pedig 20.



Megfigyelhető, hogy 1 KB-nyi memória két áramkörből áll, amelyek párhuzamosan dolgoznak, csak az adatvonalaik vannak úgy az adatbuszhoz kötve, hogy kijöjjön a 8 vezetékes csoport. A blokk által elfoglalt memóriatartomány $FC000_H$–$FC7FF_H$, ami csak egy szelekciós jel generálását teszi szükségessé, amint az a címhatárok táblázatából is látszik:

A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	...	A1	A0	Szelekció	Címterület
1	1	1	1	1	1	0	0	0	0	...	0	0	SEL=1	$FC000_H$–$FC7FF_H$
1	1	1	1	1	1	0	0	0	1	...	1	1		

A memória-áramkörök 10 címbemenettel rendelkeznek, ezekre az A1–A10 címvezetékeket vezetjük. A SEL szelekciójel egyenletét könnyen felírhatjuk a táblázat alapján:

$$SEL = A19 \cdot A18 \cdot A17 \cdot A16 \cdot A15 \cdot A14 \cdot \overline{\overline{A13} \cdot \overline{A12} \cdot \overline{A11}} =$$

$$= A19 \cdot A18 \cdot A17 \cdot A16 \cdot A15 \cdot A14 \cdot \overline{(A13 + A12 + A11)}$$

A bájtkiválasztást is figyelembe véve megkapjuk a csipkiválasztó jelek egyenletét:

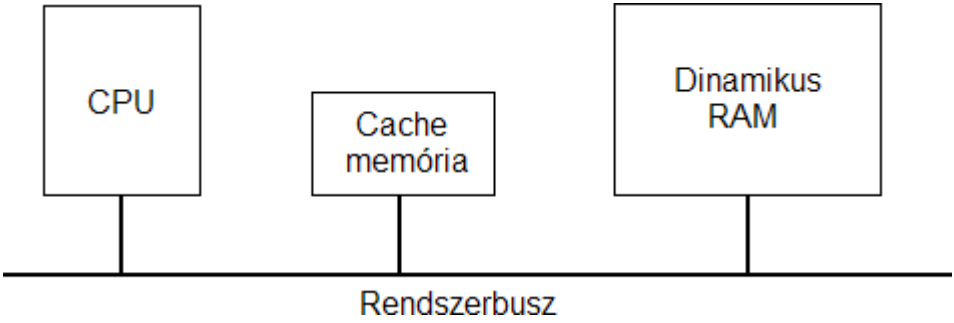
$$CS_{LOW} = SEL \cdot BE0$$

$$CS_{HIGH} = SEL \cdot BE1$$

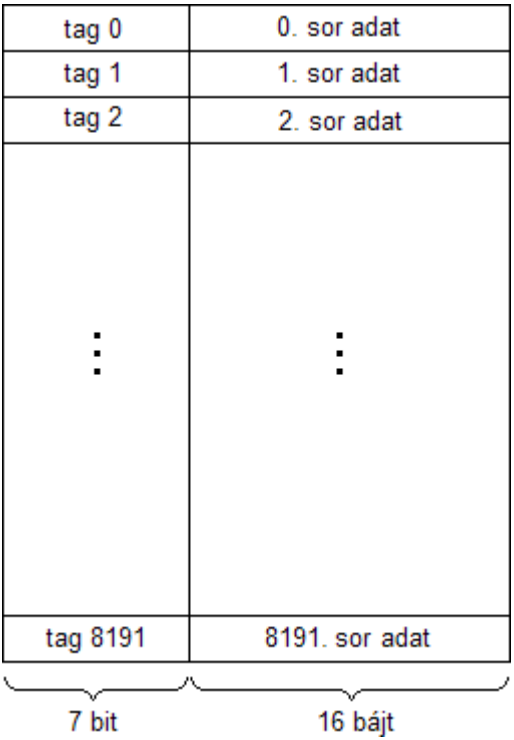
A többi vezérlőjel (olvasás, írás) alkalmazása könnyen megérthető az általános elv alapján.

Cache memória

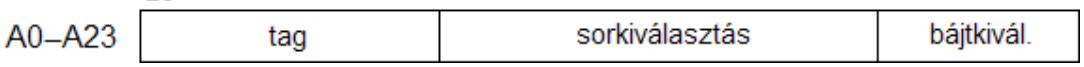
Egy **gyorsítótárat** (*cache*) tartalmazó tárhierarchia blokkvázlatát mutatja az alábbi ábra. A processzor a buszon kiküldött címről kezdeményezi a memória-hozzáférést. Ha az adat benne van a cache-ben (*cache hit*), akkor ezt gyorsan megkapja, és folytathatja a feldolgozást. Ha az adat nincs a cache-ben (*cache miss*), azt a dinamikus RAM-ból kell kiolvasni, ami hosszabb ideig tart.



Példaként egy 128 KB-os, közvetlen leképezésű, átírásos gyorsítótárat tárgyalunk. A közvetlen leképezés azt jelenti, hogy egy, a cache-be bemásolt adat mindig ugyanarra a helyre kerül. Az átírásos (*write-through*) működés esetében az összes CPU-írás közvetlenül a főtárba kerül, de ha a hivatkozott cím megtalálható a cache- ben is, akkor az adataktualizálás ott is megtörténik. Minden sor 16 bájtos, így 32 bites adatbusz esetében egy sor feltöltéséhez 4 szó átvitelére van szükség (ezt *burst* átvitelben szokás megvalósítani). Kiszámítható, hogy a cache 128 K / 16 = 8 K = 8192 sort tartalmaz. A cache vázlatos szervezése a következő:



Feltételezve, hogy a címbusz 24 bites (A0–A23), a címszó szerkezete a következőképpen alakul:

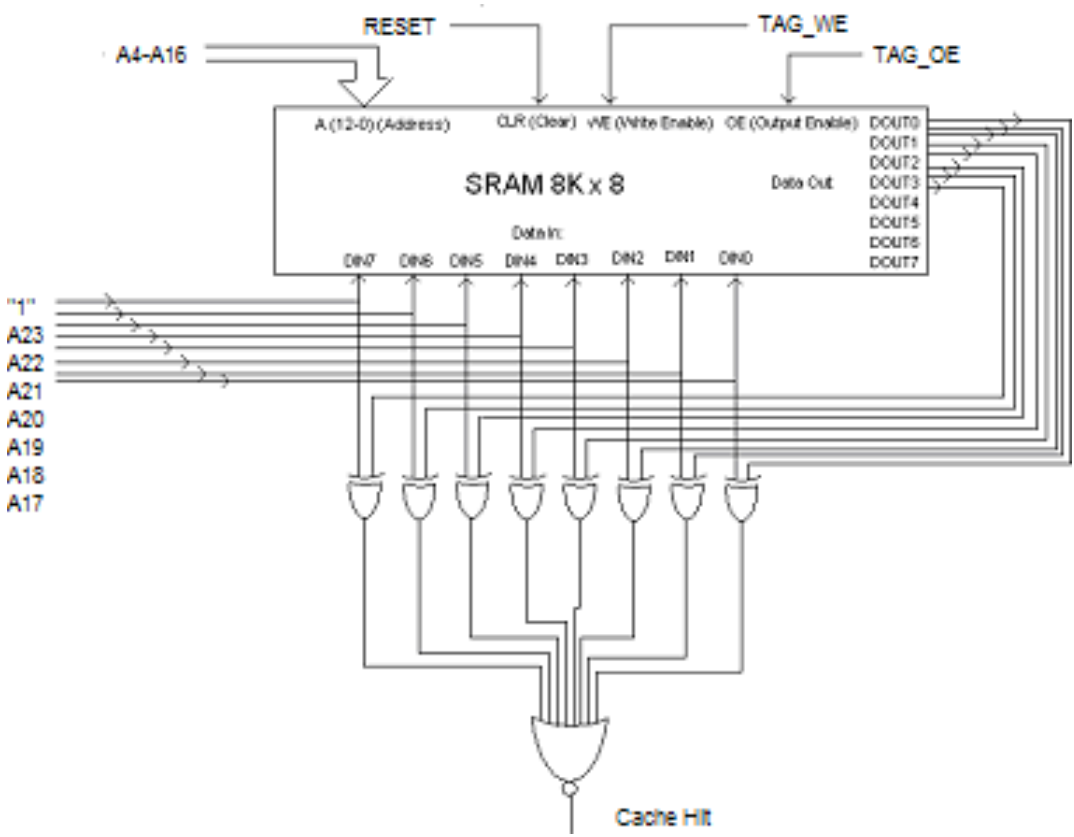


A legkisebb helyiértékű 4 bit azonosítja a bájtot egy soron belül, a következő 13 bit a sort választja ki, míg a fennmaradt 7 bit beazonosítja a keresett sort a lehetséges 128 közül. A címnek ez utóbbi része kerül összehasonlításra a kiválasztott sorban lévő taggel, és egyezőség esetén cache-hitről beszélünk, azaz a keresett adat a cache-ben van.

A gyorsítótár „tag” részének implementálásához két elemre van szükségünk:

- egy 8 K × 8 bites statikus RAM memóriára
- egy 8 bites összehasonlító áramkörre

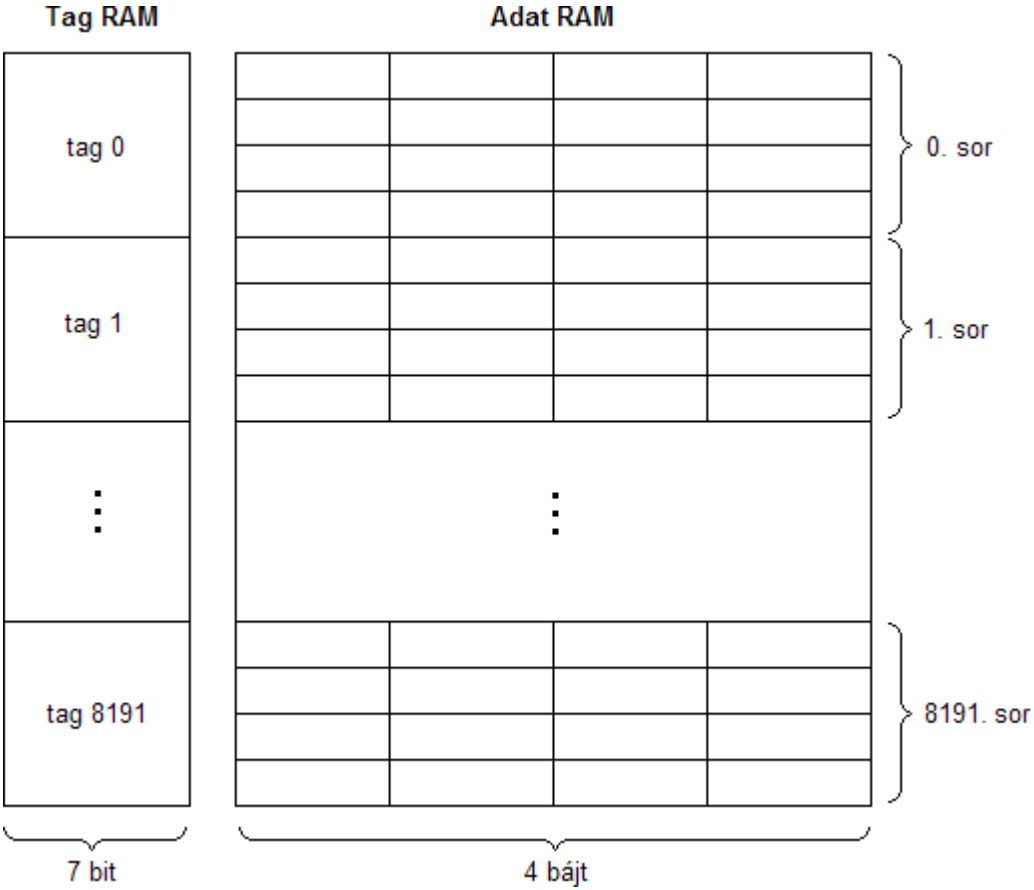
A **„tag RAM”** kapcsolási rajza a következő lesz:



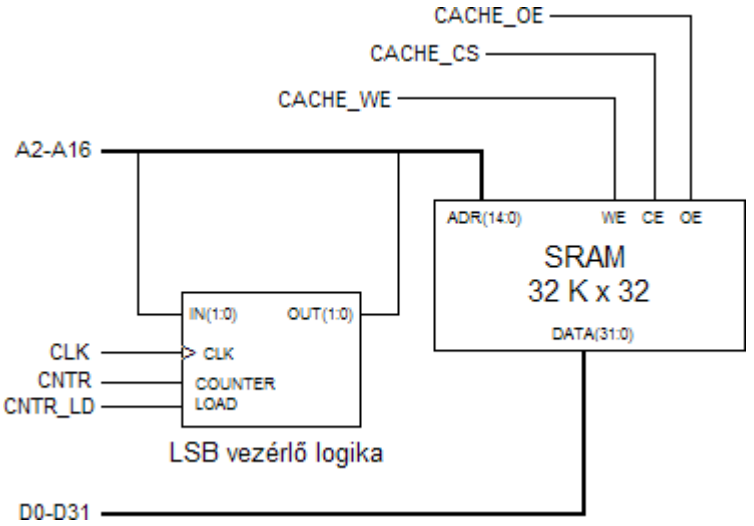
A tag memória adatbemeneteire az A17-A23 címvonalak kerülnek, amelyek egy sor beírásánál a tag részt tartalmazzák. Egy CPU-hivatkozásnál, a memória-áramkör címbemenetein azonosított rekeszben lévő adat kerül bitenként összehasonlításra (XOR kapusor) a processzor által hivatkozott cím magas helyiértékű bitjeivel. Egyezőség esetén a keresett taggel rendelkező sor benne van a cache-ben, ekkor a ***Cache Hit*** jel aktiválódik. Figyeljük meg, hogy a DIN7 bemenet logikai „1”-re van kötve, ezáltal lehet megkülönböztetni az érvényes adatokat tartalmazó rekeszeket az üresektől (mivel RESET után ez egész áramkör tartalma törlődik). Az írásnál és olvasásnál használt vezérlőjeleket a cache memória-rendszer vezérlő logikája állítja elő.

A rendszer „adat” részének minden sora – amint említettük – 16 bájtot tartalmaz. Mivel az adatbusz 32 bites, az adatmemóriát is ilyen méretű egységekbe kell szervezni, ami azt jelenti, hogy az **„adat RAM”** 32 K × 32 bites lesz

(mert a meghatározásban megadott 128 K × 8 bit = 32 K × 32 bit). Ez azt jelenti, hogy minden tagnek 4 darab 32 bites rekesz felel meg az adatrészben:



Az előző gondolatmenetnek megfelelően, az adatészt implementáló statikus RAM áramkör bekötési sémája a következő lesz:



A memóriablokk adatvonalai az adatbuszhoz csatlakoznak, ezen az útvonalon történik a cache feltöltése és kiürítése. A címszerkezetből láttuk, hogy egy-egy cache-sor azonosításához 13 címbitet használunk (A4–A16), mégis az adat RAM címbemeneteire 15 bites cím került (A2–A16). Ez azért van, mert 2 bit (A2 és A3) szükséges beazonosítani a soron belül, hogy melyik 32 bites rekeszben található a keresett egy bájtos adat. Ebből kifolyólag, a címbemenetek alsó két bitjének a kezelésére szükség van az ábrán látható LSB vezérlő logikára, amely egy multiplexert és egy számlálót tartalmaz. Cache hit esetén ezt a két bitet a multiplexer szolgáltatja a CPU-tól érkező címből (A2 és A3). Akkor viszont, amikor cache hitet követően egy sor feltöltése történik a dinamikus RAM-ból, a két címbit a számlálótól származik, amely négy rekeszen át inkrementálja a címet, a négy 32 bites szó feltöltésének ütemében.

A cache megfelelően időzített vezérlőjeleit egy, a gyorsítótárhoz tartozó vezérlő logika állítja elő. Ezt a lehetséges négy esetben (*read-hit, read-miss, write-hit, write- miss*) szükséges idődiagramok alapján, mint véges állapotú automatát lehet implementálni.

KÉRDÉSEK, TÉTELEK

5.1. Mekkora a központi memória címtartománya?

5.2. Mi a memória-áramkörök lineáris szelekciójának elve?

5.3. Mit értünk memóriatérkép alatt?

5.4. Mi a memória-áramkörök dekódolt szelekciójának elve?

5.5. Vázolja fel és magyarázza el egy ROM memória-áramkör belső felépítését.

5.6. Adja meg a ROM memóriacsip buszrendszerhez csatlakoztatásának elvi vázlatát.

5.7. Jelölje meg egy memória-áramkör jellemző idődiagramján a hozzáférési időt.

5.8. Adja meg a statikus RAM memóriacsip buszrendszerhez csatlakoztatásának elvi vázlatát.

5.9. Mi célt szolgálnak a bájtkiválasztó jelek?

5.10. Milyen két tárolórészből áll egy cache-rendszer? E tárolórészek címzéséhez a címszó mely mezőit kell használni?

5.11. Mit jelent, és hogy állapítható meg a cache-hit?

6. INTERFÉSZEK, HÁTTÉRTÁROLÓK TERVEZÉSI KÉRDÉSEI

6.1 A portok bekötésének elve

A számítógép külső egységei (perifériái) nem közvetlenül csatlakoznak a buszhoz, hanem illesztő áramkörökön keresztül. Ezeket **interfészeknek** (*interface*) nevezik, amelyeknek fő feladata egy egységes hardver felületet biztosítani a processzor számára a perifériák felé az adatok átviteléhez. A busz felöli oldalon mindegyik interfész azonos módon kell csatlakozzon, betartva a buszprotokollt, a másik oldalon viszont a csatlakozás a külső egység (monitor, billentyűzet, nyomtató, merevlemez stb.) sajátosságainak megfelelően. Egyes esetekben az interfész komplex áramkör, akár dedikált processzort is tartalmazhat, ezért sokszor **vezérlőnek** is nevezik (monitorvezérlő, lemezvezérlő stb.). Az interfészben az adatok úgynevezett **portokon** keresztül kerülnek átvitelre. Az átvitel irányának függvényében megkülönböztethető

- bemeneti port (*input port*)
- kimeneti port (*output port*)
- bemeneti/kimeneti port *(I/O port*)

Az adatátvitel a processzor és egy periféria között általában két lépésben történik: a processzor elküldi a sínen az adatot a portnak, majd onnan az interfész logikája továbbítja a perifériának. Visszafelé, az interfész logikája fogadja a perifériától érkező adatot, majd a processzor kiolvassa az interfész portjából. Ebből következik, hogy a port általában az átvitt adatok ideiglenes tárolását is biztosítja, ezért implementálására célszerű regisztert alkalmazni. A tárolt adatok típusától függően, az interfészekben három típusú **regiszter** lehet:

- adatregiszter (adatpuffer)
- állapotregiszter

- parancsregiszter

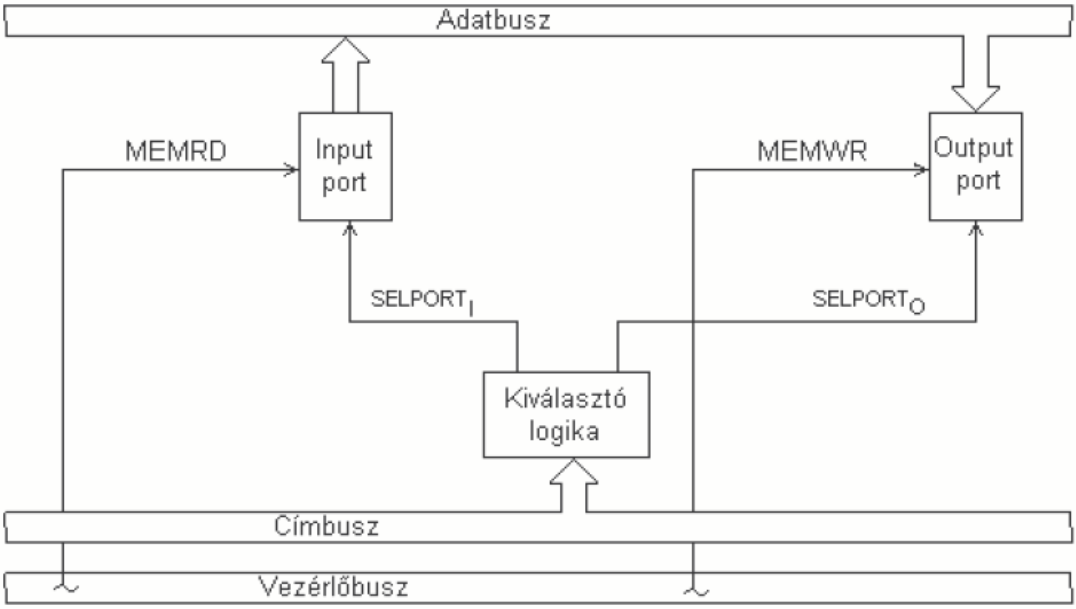
Az interfészekben lévő mindegyik regiszter külön címmel kell rendelkezzen, hogy a processzor utasításaival meg lehessen címezni egy adatátvitel érdekében. A címtartományukat kétféleképpen lehet biztosítani:

- **A memória címtartományába beágyazott I/O** (*memory mapped I/O*) – ekkor a főtár számára fenntartott címtartományból veszünk el egy részt a portok számára (minden I/O regisztert úgy kezelünk, mint egy-egy memóriarekeszt)
- **Izolált I/O** (*isolated I/O*) → egyes processzorok esetében – például az Intelnél

– egy elkülönített címtartomány áll rendelkezésünkre az I/O címek kiosztására (így a memória címtartománya teljes egészében megmarad a főtár számára). Általában az I/O címtartomány kisebb a memória címtartományánál, tehát az I/O címek rövidebbek a memóriacímeknél.

A memória címtartományába beágyazott I/O esetében a portok bekötésének elve megegyezik a memória-áramkörök bekötésével, amint az alábbi ábrán látható. Minden portnak egy-egy szelekciós jele van, de természetesen ha több port van egy áramkörben, akkor az áramkörnek lesz egy csipkiválasztó bemenete, és megfelelő számú bemenete a csipen belüli portok (regiszterek) kiválasztására. A kiválasztó logika a már ismert lineáris vagy dekódolt szelekció alapján dolgozik. Mivel ebben az esetben a portok és a memóriarekeszek elérését is azonos, MOV utasításokkal kezeljük, az átviteli buszciklusban aktiválódó MEMRD, illetve MEMWR jeleket használjuk a portok vezérlésére (beírás, illetve kiolvasás). A programban a portba írást, illetve portból olvasást a következő módon valósítjuk meg:

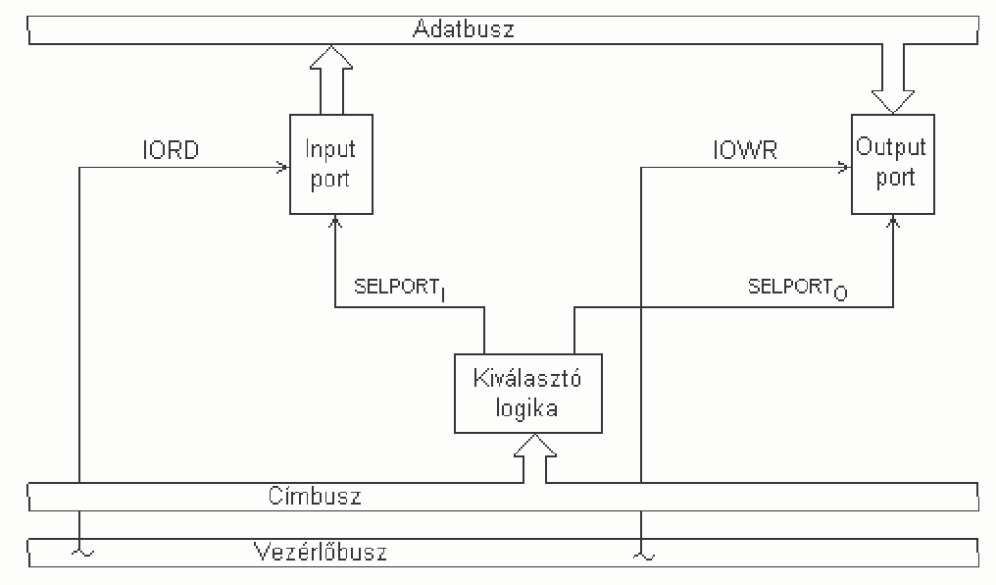
```
MOV AL, PORTIN
MOV PORTOUT, AL
```



Az izolált I/O címtartomány esetében a portok kezelésére külön utasítások állnak rendelkezésre a processzor utasításkészletében: IN a portból való kiolvasásra és OUT a portba való írásra. E két műveletet a programban a következőképpen valósítjuk meg:

IN AL, PORTIN OUT PORTOUT, AL

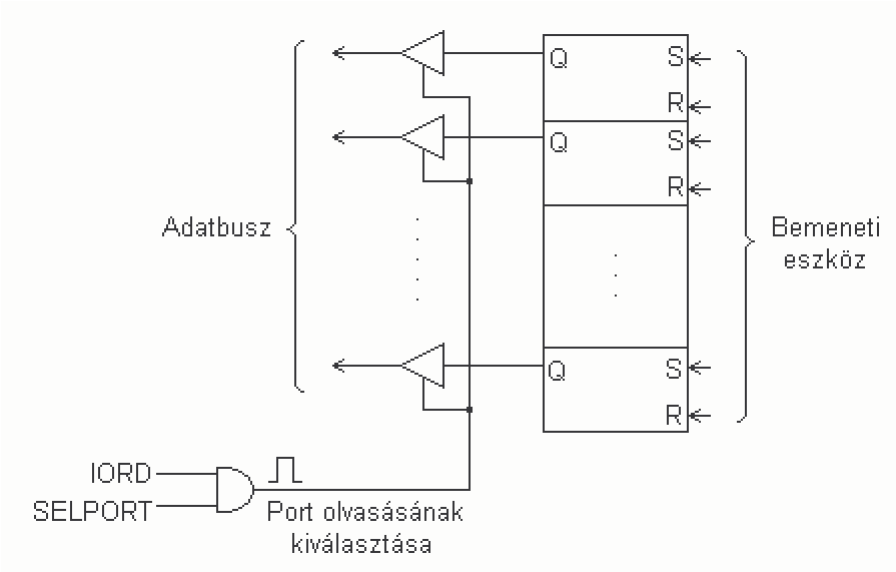
Ezen utasítások végrehajtásakor a vezérlőbuszon az IORD, illetve IOWR vezérlőjelek aktiválódnak, tehát ezeket használjuk a portok vezérlésére. A portok szelekciós jeleit a címbusz jeleiből állítjuk elő lineáris vagy dekódolt szelekcióval:



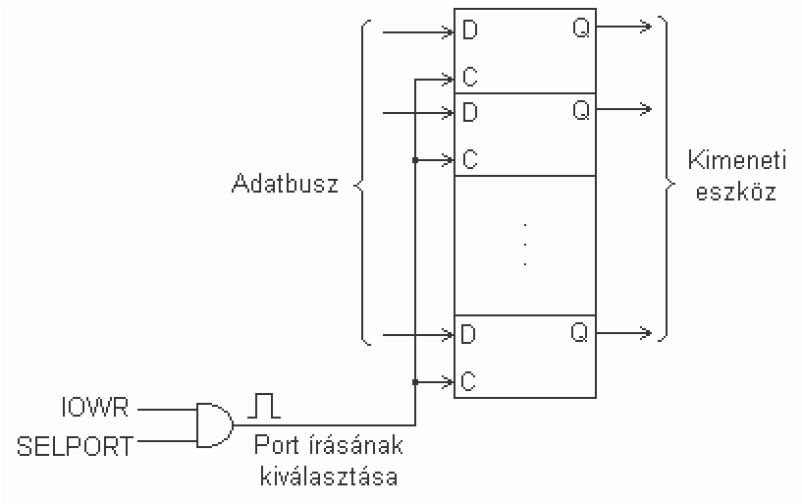
6.2 Regiszterekből megvalósított egyszerű portok

A processzor és egy külső egység közötti adatátvitel egy puffer regiszter által megvalósított porton keresztül történik úgy, hogy a küldő behelyezi az adatot a regiszterbe, a fogadó pedig onnan átveszi.

Egy regiszterből megvalósított **bemeneti portra** mutat példát az alábbi ábra. Az adatbuszra való csatlakozást háromállapotú meghajtó kapukkal kell megoldani. Ezek megnyitása az IN utasítás hatására aktiválódó vezérlőjel és a címből generált szelekciós jel hatására történik.

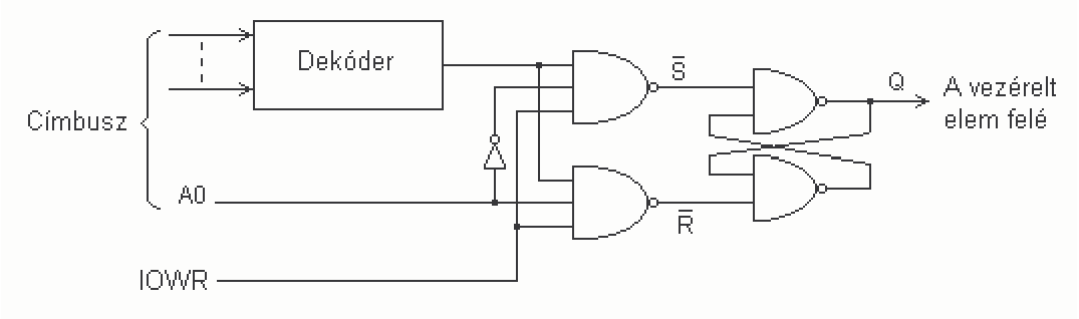


A következő ábra D típusú flip-flopokból épített regiszter használatát mutatja **kimeneti portként**. Az adat átvitele a processzorból a port regiszterébe OUT utasítással történik, ami a port kiválasztását, és így az adatnak a regiszterbe való beírását eredményezi.



Egyes esetekben nincs szükség adat átvitelére, csak egy **kétállapotú elem vezérlésére** (relé, szelep, motor stb.) Az ilyen elemet egy portként alkalmazott

flip-flop kimenetére kell kötni. A flip-flop vezérlését (0-ra vagy 1-re állítását) meg lehet oldani az adatbusz egyik bitjével is, de alternatív megoldásként az alábbi kapcsolást lehet alkalmazni:



Amint látható, nem alkalmazunk egy adatvonalat sem, de a port két címet foglal el az I/O címtartományban: egy párosat (A0=0), és a szomszédos páratlant (A0=1). A flip- flop 0-ra hozatalát a páros címre való hivatkozással, az 1-re állítását pedig a páratlannal érjük el a kimeneti utasításból:

OUT (2k), AL ; Q←1

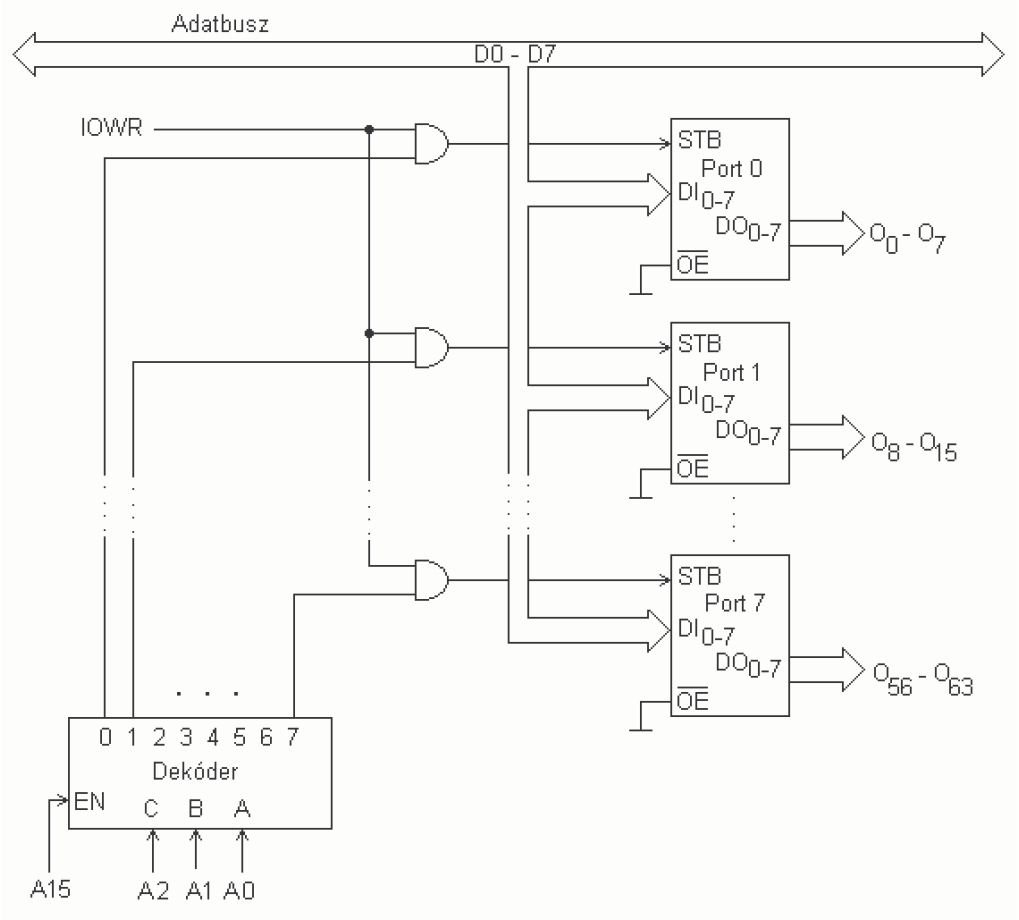
OUT (2k + 1), AL ; Q←0

A következő példa egy **64 digitális kimenetet** tartalmazó interfészt mutat be. A kimenetek 8 portba vannak szervezve, mindegyik port egy-egy 8 bites *buffer latch* áramkörrel (lásd a 2. fejezetet) lett implementálva. Az interfész kiválasztása A15=1- gyel történik, a 8000_H, 8001_H, 8002_H, 8003_H, 8004_H, 8005_H, 8006_H, 8007_H címeken.

Az A0–A2 bitek kombinációit használjuk a portok kiválasztására. Az egyszerűség végett az A3–A14 bitek nem vesznek részt a szelekcióban, ami egy nem teljes dekódoláshoz vezet. (Ezt a módszert szokták alkalmazni akkor, amikor kevés portot tartalmaz a rendszer, és nem szükséges felhasználni az egész címtartományt.) Egy kiírás például a Port 0-ba a következő utasítással valósítható meg:

OUT 8000H, AL

Ennek hatására a dekóder aktiválja a 0-ás kimenetén lévő jelet, ami kombinálva az I/O írás vezérlőjelével beírja az adatbuszon érkező bájtot a regiszterbe.



6.3 A párhuzamos interfész

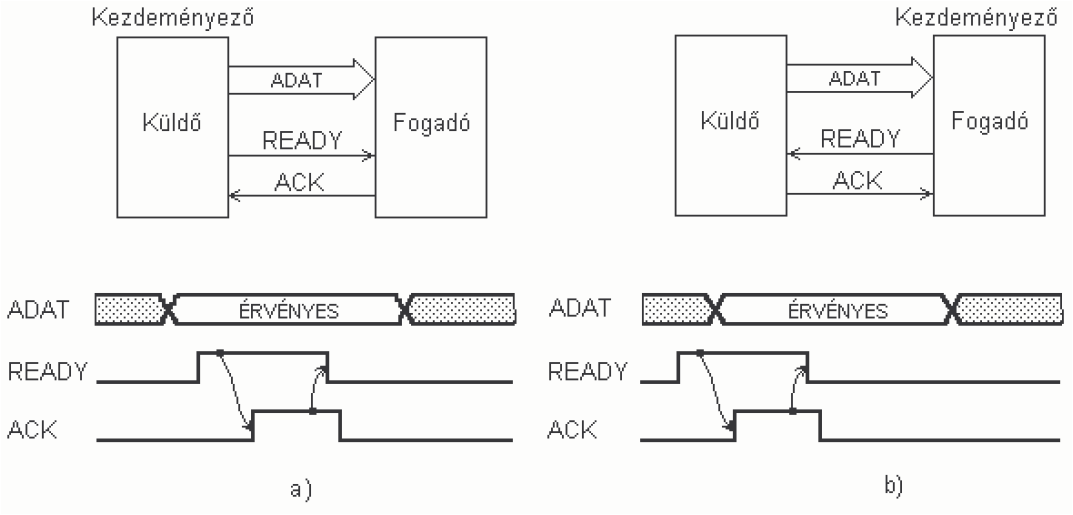
A párhuzamos adatátvitel az interfész és a periféria között úgy zajlik, hogy egy szó (általában bájt) minden bitje külön vonalon kerül átvitelre, tehát az egész szó átvitele egy ütemben megy végbe. Az átvitelt meg lehet valósítani

- csak adatjelekkel, ha nincs szükség szinkronizálásra a külső eszköz és a rendszer között (például egy LED-sor kigyújtására);
- adat- és vezérlőjelekkel, amikor szinkronba kell hozni az átvitelt a két fél között.

Ez utóbbi esetben a vezérlőjelek a **kézfogásos protokollt** (*handshaking*) valósítják meg, aminek két változata van, attól függően, hogy melyik fél a kezdeményező (lásd az ábrát).

a) Ha az átvitelt *a küldő kezdeményezi*, akkor a READY jelzi, hogy az adatvonalakon átvételre kész adatok vannak. Az ACK (*Acknowledge*) az átvételt igazolja vissza.

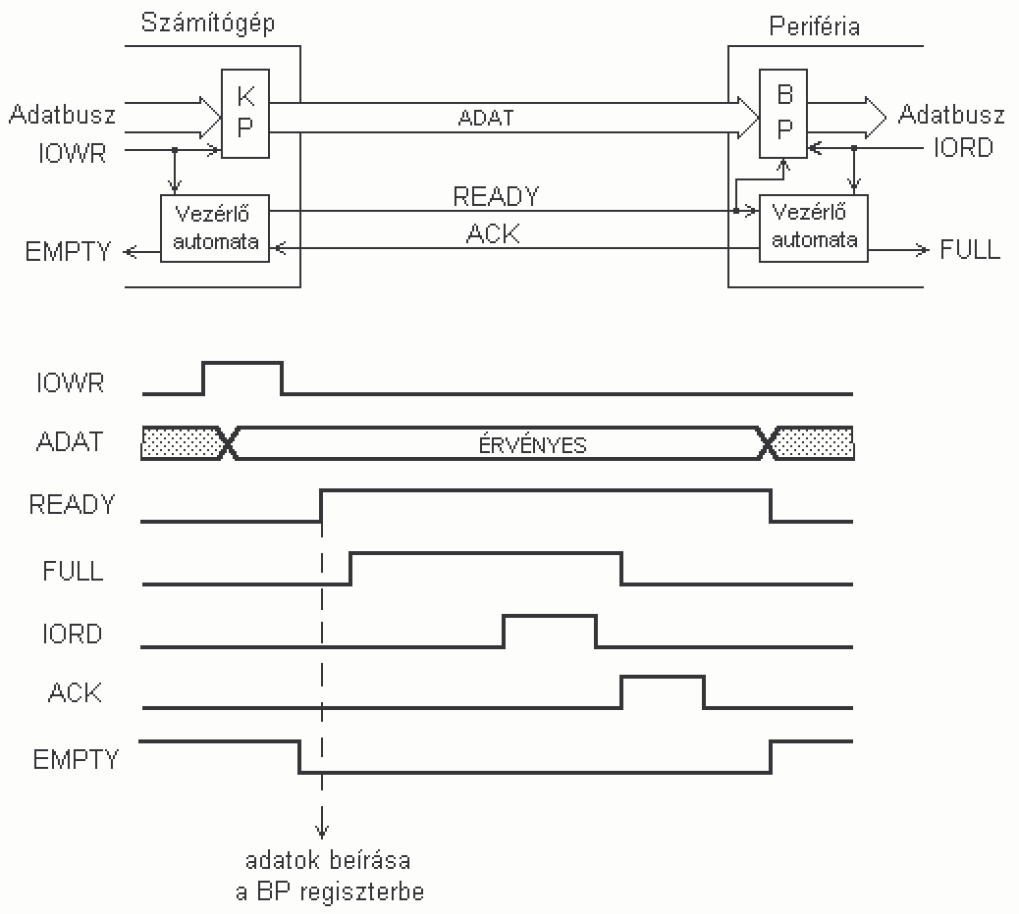
b) Ha az átvitelt *a fogadó kezdeményezi*, akkor a READY jelzi, hogy a fogadó adatokat kér, kész ezeket átvenni. Az ACK az adatok adatvonalakra való helyezését igazolja vissza.



A továbbiakban részletesebben áttekintjük a számítógépben, valamint a perifériában lévő interfészben szükséges alkotóelemeket és együttműködésüket a párhuzamos adatátvitel során. Kezdeményezőnek a számítógépet tekintjük.

a) Adatkivitel (output)

Az két interfész kapcsolatának elvi vázlata, valamint a jelek idődiagramja a következő:



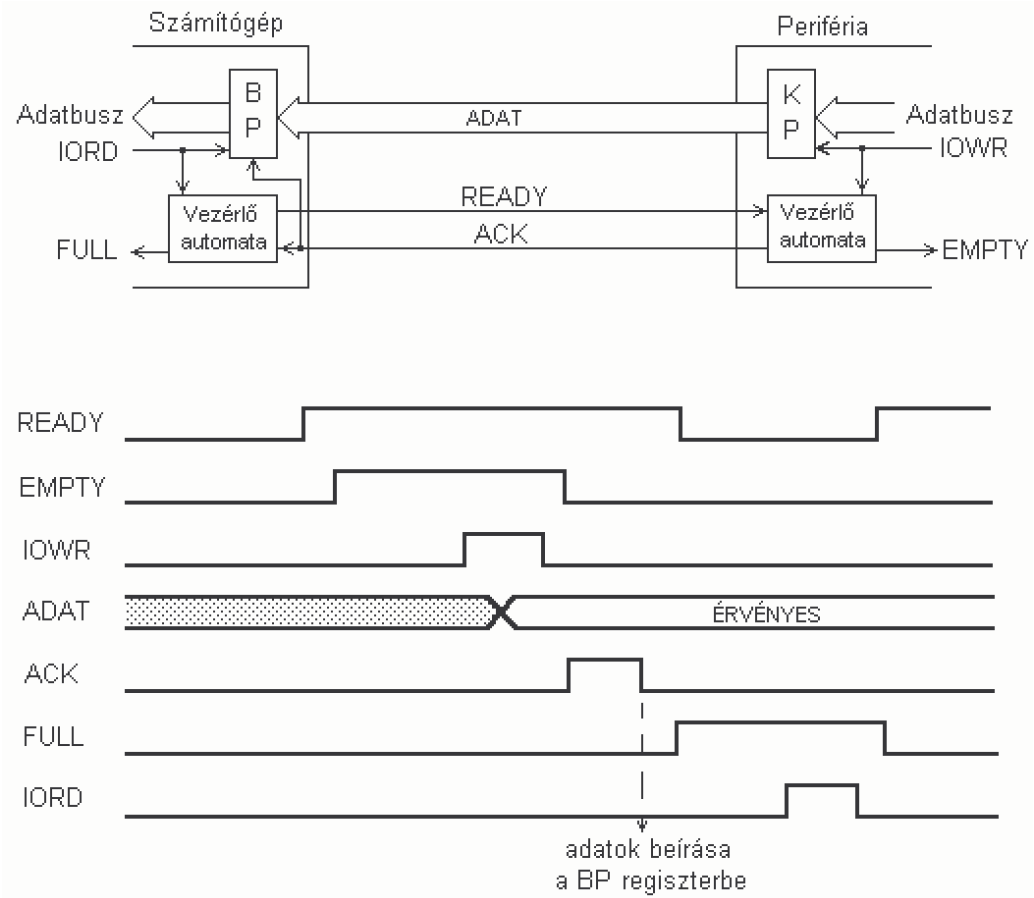
A működés megértéséhez néhány pontosítás szükséges:

- A KP (kimeneti puffer) a kimeneti port regisztere, a BP (bemeneti puffer) a bemeneti port regisztere;
- Az EMPTY a KP üres állapotát mutató jelzőbit;
- A FULL a BP teli állapotát mutató jelzőbit;
- Az interfészek vezérlését egy-egy szekvenciális automata áramkör biztosítja.

Az elküldendő adatot a gép processzora egy OUT utasítással (amely az IOWR jelet aktiválja) írja be a KP-be. A vezérlőlogika, ezt észlelve, aktiválja a READY-t, jelezve a vevő felé, hogy az adat átvételre készen áll. Ezzel a jellel megtörténik az adat beírása a BP-be, amit a periféria interfészének vezérlője érzékel. Következésképpen aktiválja a FULL jelzőbitet, ami – lekérdezés vagy megszakítás útján – tudatja a periféria processzorával, hogy adat érkezett a portba. Az adatot a processzor egy IN utasítással olvassa be (ez aktiválja az IORD jelet), ami után a vezérlő automata visszaigazolja az adat átvételét (ACK=1). A küldő számítógép oldalán a vezérlő ezután aktiválja az EMPTY-t, ezzel jelezve (lekérdezéses vagy megszakításos módon) a saját processzorának, hogy a KP ismét üres, küldheti a következő adatot.

b) **Adatbevitel (input)**

Az két interfész kapcsolatának elvi vázlata, valamint a jelek idődiagramja a következő:



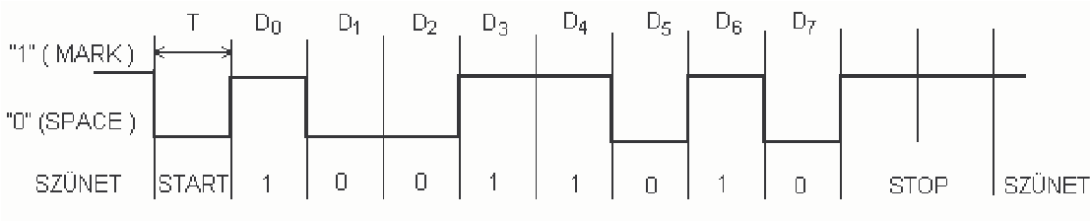
A jelölések megegyeznek az előző esettel. A számítógép a READY jel aktiválásával jelzi a fogadókészségét. Ezt érzékelve, a periféria interfészének vezérlője aktiválja az EMPTY-t, adatot kérve a saját processzorától. Lekérdezéses vagy megszakításos technika nyomán a periféria processzora OUT utasítást hajt végre, amivel beírja az adatot a KP-be (IOWR jellel). Az interfész automatája ezt érzékelve visszaigazol a számítógépnek (ACK=1), hogy az adat átvehető (az ACK jel hatására az adat be is kerül a BP-be). A fogadó interfész vezérlője jelzi a processzorának, hogy a bemeneti pufferbe adat érkezett (FULL=1). A processzor lekérdezéssel vagy megszakítás útján tudomást szerez az adat érkezéséről, és végrehajt egy IN utasítást az adat beolvasására a portból (IORD jellel). Az interfész vezérlője most már ismét aktiválhatja a READY-t egy újabb adat átvételéhez.

6. 4 A soros interfész

A soros adatátvitel gazdaságosabb a párhuzamosnál, mivel az adatok továbbítása bitenként történik egyetlen vonalon. A helyes adatátvitel érdekében a két félnek szinkronban kell működni, azaz ugyanolyan ütemben kell a biteket átvenni a fogadónak, mint amilyennel az a küldő a vonalra helyezi. A **szinkronizmus** biztosításának több technikája van:

- Egy külön, második vonal alkalmazása az órajel átvitelére – ez a legegyszerűbb megoldás, de csak elszigetelten, kis távolságra használják (például a PC-k billentyűzete esetében);
- Közös órajel nélküli, csak adatátvitel – ekkor mindkét oldalon egy-egy külön órajel-generátorra van szükség, amelyek azonos frekvencián kell dolgozzanak

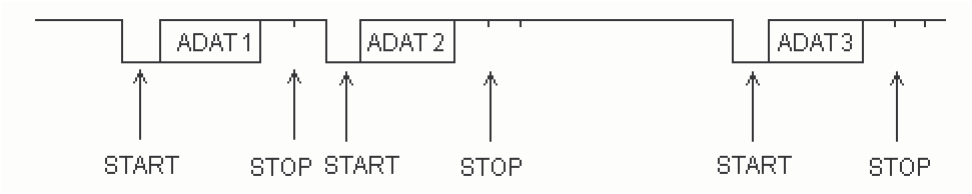
– mi a továbbiakban ezzel a megoldással foglalkozunk. Itt a gondot az jelenti, hogy az órajel-generátorok frekvenciái sem lehetnek pontosan azonosak, így ezeket időnként egymáshoz kell szinkronizálni. Erre a célra két módszer létezik:



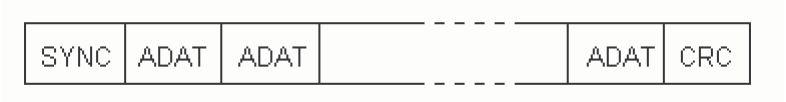
Aszinkron átvitel – amikor kisebb adategységet (általában bájtot) fognak közre START és STOP bitekkel:

A vevő órajel-generátora mindig a START bit lefutó éléhez képest T/2- vel később indul, ezért a csupán nyolc adatbitet helyesen fogja mintavételezni minden periódus közepe körül, még ha kissé el is tér a frekvenciája az adóétól (ez a módszer nem annyira igényes az órajel- generátorok pontosságát illetően). A STOP jelzi a fogadónak az adattovábbítás végét, ekkor a vonal 1-es állapotba kell álljon. Az adatbitek után szoktak egy paritásbitet is beiktatni, de

nem kötelező. A módszert azért nevezik aszinkronnak, mert két szó továbbítása között akármennyi szünetet lehet hagyni:

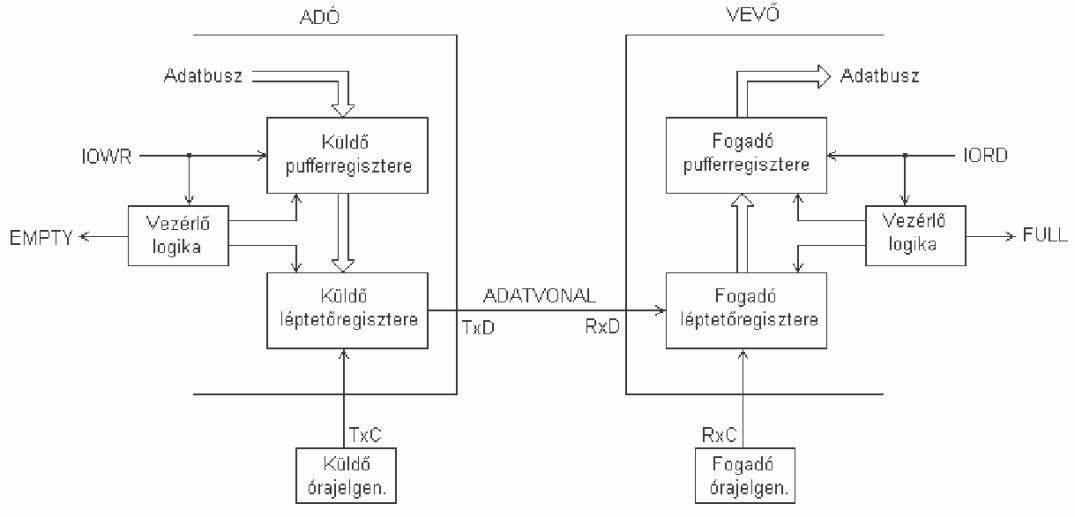


Szinkron átvitel – amikor egy folyamatosan átvitt nagyobb adatblokkhoz (több ezer bájt) csatolunk egy szinkronizációs karaktert:



A fogadó órajel-generátorának szinkronizálására ritkábban kerül sor, ezért ezt igényesebben kell megépíteni, mint az előző esetben. Ez a módszer viszont nagyobb átviteli rátát biztosít, mint az aszinkron átvitel. Az adatblokk végén hibadetektálás, esetleg hibajavítás céljából egy CRC (*Cyclic Redundancy Check* – ciklikus redundancia ellenőrzés) mezőt iktatnak be. Az itt szereplő bitek értékét az adatbiteknek megfelelő polinomnak egy generátor polinommal való osztása után létrejövő maradék adja.

Az átvitelben részt vevő két soros interfész kapcsolatának elvi vázlatát az alábbi ábra mutatja (az egyik lehet a számítógép, a másik a periféria, vagy egy másik számítógép):



A küldő processzor egy OUT utasítással (amely az IOWR jelet aktiválja) beírja az adatot a küldő pufferregiszterbe. Innen, az interfész vezérlője átviszi egy léptetőregiszterbe, ahonnan az órajel (TxC) ütemében az adat bitjei sorban kikerülnek az adatvonalra. Ezzel egyidejűleg, a vevőnél a biteket belépteti a fogadó órajele (RxC) az ottani léptetőregiszterbe. Ha minden bit megérkezett, a vezérlő átviszi az adatot a fogadó pufferregiszterbe, maj jelzi a processzornak (FULL=1), hogy a regiszter megtelt, azaz adat érkezett. A fogadó oldali processzor – lekérdezéses vagy megszakításos technikával - megtudja, hogy adat érkezett, és a beolvasására végrehajt egy IN utasítást (IORD aktiválódik). A küldő oldalon, miután a küldő pufferregisztere kiürült, az EMPTY jelzőbit jelzi a processzornak (lekérdezés vagy megszakítás útján), hogy küldhet egy újabb adatot. Mivel a soros átvitelnél – ellentétben a párhuzamossal – nincs visszaigazolás a fogadótól az adat átvételéről, **ritmushiba** (*overrun error*) keletkezhet, ha a vevő processzora nem vette át az adatot a következő megérkezéséig (ez utóbbi felülírja az előzőt a fogadó pufferben).

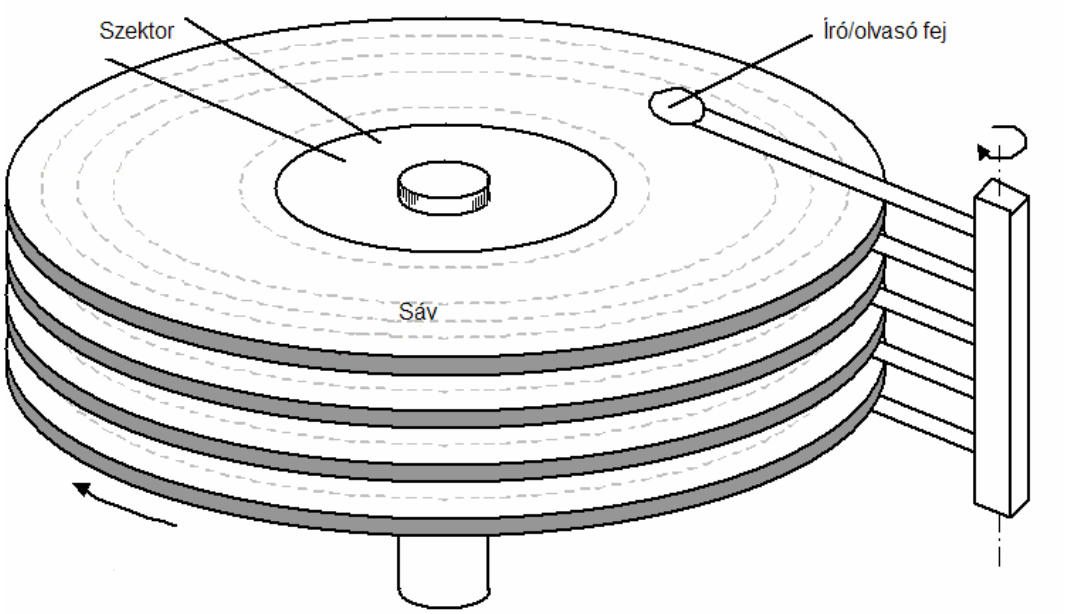
A soros adatátvitel sebességét bit/s-ban mérik, ami a másodpercenként átvitt bitek számát jelenti:

$$D=\frac{1}{T}$$

(A képletben T az órajel periódusa.)

Háttértárolók kezelése

A merevlemezes háttértároló felépítését az alábbi ábra mutatja. Az egység több lemezből áll, mindegyikhez tartozik egy **író/olvasó fej**. A lemezek koncentrikus körökre vannak osztva, ezek a sávok – az egymás fölött lévők egy-egy **cilindert** alkotnak. A sávok **szektorokra** vannak osztva.



Minden adatátvitel csak rögzített méretű blokkok formájában történik, ezek mérete a PC-knél 512 bájt, és egy szektorban lévő adatot jelent. Az írás/olvasás szempontjából tehát a merevlemezt szektorokból álló tároló tömbnek tekinthetjük. A műveletre kiválasztott szektor megcímzése kétféleképpen lehetséges:

- **Fizikai címzés** – ekkor a cilinder/fej/szektor sorszámait kell megadni (CHS – *Cylinder/Head/Sector*)

- **Logikai blokkcímzés** (LBA – *Logical Block Addressing*) – ekkor, a tárat egyetlen tömbként tekintve, a szektorokat folyamatosan számozzuk meg 0-tól kezdve

A címet CHS-ből LBA-ba a következő képlettel lehet átalakítani:

$$LBA = (C \times HpC + H) \times SpH + S - 1$$

Az egyenletben HpC (*Head Count per Cylinder Count*) az egy cilinderre eső fejek számát jelenti, az SpH (*Sector Count per Head Count*) pedig az egy fejhez tartozó (egy sávban lévő) szektorok száma.

Fordítva, LBA-ból HS-be az átalakítás a következőképpen lehetséges:

$$C = LBA / (HpC \times SpH)$$

$$H = (LBA / SpH) \bmod (HpC)$$

$$S = (LBA \bmod (SpH)) + 1$$

A merevlemezhez való hozzáférés a mai PC-kben – legelterjedtebb módon – az ATA (*Advanced Technology Attachment*) szabványos interfészen keresztül történik. A PC alaplapjához való csatlakoztatás tekintetében ennek van párhuzamos (*Parallel ATA*) és soros (*Serial ATA*) változata is, ez utóbbi a korszerűbb.

Az írási/olvasási/formázási műveletek az interfészben található **regisztereken** keresztül történik. Ezeknek külön portcímük van, amelyekre a processzoron futó programból IN és OUT utasításokkal lehet hivatkozni. A következő táblázat egy ATA vezérlő regisztereinek címkiosztását mutatja:

Port	Méret	Eltolás (az alapcímhez képest)	Címbitek				Funkció	
			A3	A2	A1	A0	Olvasható regiszter (Read)	Írható regiszter (Write)
P00	16 bit	0	0	0	0	0	Data Read	Data Write

P01	8 bit	+1	0	0	0	1	Error Register	Feature Register
P02	8 bit	+2	0	0	1	0	Sector Count Register	Sector Count Register
P03	8 bit	+3	0	0	1	1	Sector Number Register	Sector Number Register
							LBA Low Register (bits 0–7)	LBA Low Register (bits 0–7)
P04	8 bit	+4	0	1	0	0	Cylinder Low Register	Cylinder Low Register
							LBA Middle Register (bits 8–15)	LBA Middle Register (bits 8–15)
P05	8 bit	+5	0	1	0	1	Cylinder High Register	Cylinder High Register
							LBA High Register (bits 16–23)	LBA High Register (bits 16–23)
P06	8 bit	+6	0	1	1	0	Drive and Head Register	Drive and Head Register
							Drive and LBA Register (bits 24–27)	Drive and LBA Register (bits 24–27)
P07	8 bit	+7	0	1	1	1	Status Register	Command Register
P08	8 bit	+E	1	1	1	0	Alternate Status Register	Device Control Register

Az adatok olvasása/írása 16-bites egységekben történik az interfész alapcímén lévő portból. A CHS, illetve LBA módot a Drive regiszter egyik bitjével lehet beállítani. Látható, hogy egyes – a merevlemezen lévő adatok megcímzésére használt – regiszterek jelentése változik a használt címzési módnak megfelelően.

Az I/O művelet megvalósítására az interfész két lehetőséget biztosít a tervező számára:

- **Programozott** átvitel (PIO mód) – ekkor a processzor lekérdezéses alapon, IN illetve OUT utasításokkal, közvetlenül a programból vezérli az átvitelt. Például egy olvasás a következő akciókat feltételezi:
  - a CPU kiküldi a paramétereket a vezérlőnek
  - a CPU megvárja, amíg az adat rendelkezésre áll
  - a CPU átviszi az adatokat a központi memóriába

- **Közvetlen memória-hozzáféréssel** zajló átvitel (DMA mód) – ekkor a processzor nem vesz részt magában az adatátvitelben, ezt az interfészben lévő DMA egység végzi. Például olvasásnál a processzor programból a következő feladatokat kell elvégezze:
  - a CPU kiküldi a paramétereket a vezérlőnek
  - a CPU aktiválja a DMA egységet a művelet elkezdése végett
  - a CPU inaktiválja a DMA egységet a művelet befejezése után

Részletesebben, a regiszterek használatát is illusztrálva, megadjuk egy adatbevitel, illetve egy adatkivitel lépéseit PIO módban.

a) **Adatbevitel (intput)**

1. Kiválasztjuk a meghajtót az ATA eszközszelekciós protokolljának megfelelően:
   - Várakozunk, amíg BSY=0 (*Drive Busy*) és DRQ=0 (*Data Request*) az Alternate Status Registerben;
   - Beállítjuk a Device and Head Registerben a megfelelő DEV (*Device Selection*) és LBA (*Logical Block Addressing*) bitet;
   - Várakozunk 400 ns-ot;
   - Várakozunk, amíg BSY=0 és DRQ=0 az Alternate Status Registerben;
   - Beállítjuk a PIO módot a Features Registerben;
2. Beírjuk a hozzáférés paramétereit a Cylinder Low, Cylinder High, Sector Number és Sector Count regiszterekbe;
3. Beírjuk a READ SECTORS parancsot a Command Registerbe;
4. Várakozunk 400 ns-ot;
5. Várakozunk egy ATA-megszakításra, amely jelzi, hogy az adat megérkezett a meghajtóról, kész a beolvasásra;
6. Beolvassuk a paraméterekben megadott (Sector Count × 512 bájt) mennyiségű adatot 16 bites egységekben a Data Read regiszterből.

b) Adatkivitel (output)

1. Kiválasztjuk a meghajtót az ATA eszközszelekciós protokolljának megfelelően:
   - Várakozunk, amíg BSY=0 és DRQ=0 az Alternate Status Registerben;
   - Beállítjuk a Device and Head Registerben a megfelelő DEV és LBA bitet;
   - Várakozunk 400 ns-ot;
   - Várakozunk, amíg BSY=0 és DRQ=0 az Alternate Status Registerben;
   - Beállítjuk a PIO módot a Features Registerben;
2. Beírjuk a hozzáférés paramétereit a Cylinder Low, Cylinder High, Sector Number és Sector Count regiszterekbe;
3. Beírjuk a WRIRE SECTORS parancsot a Command Registerbe;
4. Várakozunk 400 ns-ot;
5. Várakozunk, amíg BSY=0 és DRQ=1 az Alternate Status Registerben;
6. Beírjuk a paraméterekben megadott (Sector Count × 512 bájt) mennyiségű adatot 16 bites egységekben a Data Write regiszterbe;
7. Várakozunk egy ATA-megszakításra, amely jelzi, hogy az adatok kiírása a meghajtóra befejeződött.

KÉRDÉSEK, TÉTELEK

6.1. Milyen módszereket alkalmazhatunk a portok (I/O regiszterek) címzésére? Ismertesse a módszerek lényegét (beleértve az utasításokat és a vezérlőjeleket is)!

6.2. Vázoljon fel egy regiszterből épített bemeneti portot!

6.3. Vázoljon fel egy regiszterből épített kimeneti portot!

6.4. Vázolja fel a számítógép és egy periféria közötti párhuzamos adatkivitelt megvalósító elemek kapcsolatát, és magyarázza el kronologikus sorrendben a működését!

6.4. Vázolja fel a számítógép és egy periféria közötti párhuzamos adatbevitelt megvalósító elemek kapcsolatát, és magyarázza el kronologikus sorrendben a működését!

6.5. Vázolja fel két eszköz közötti soros adatbevitelt megvalósító elemek kapcsolatát, és magyarázza el kronologikus sorrendben a működését!

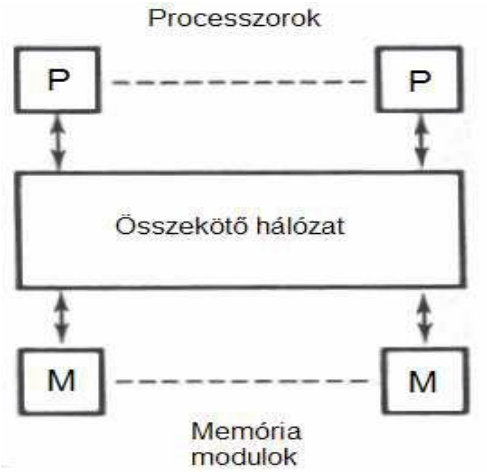
6.6. Egy merevlemezes egységnél mit értünk fizikai címzés, illetve logikai blokkcímzés (LBA) alatt?

7. MULTIPROCESSZOROS REND-SZEREK

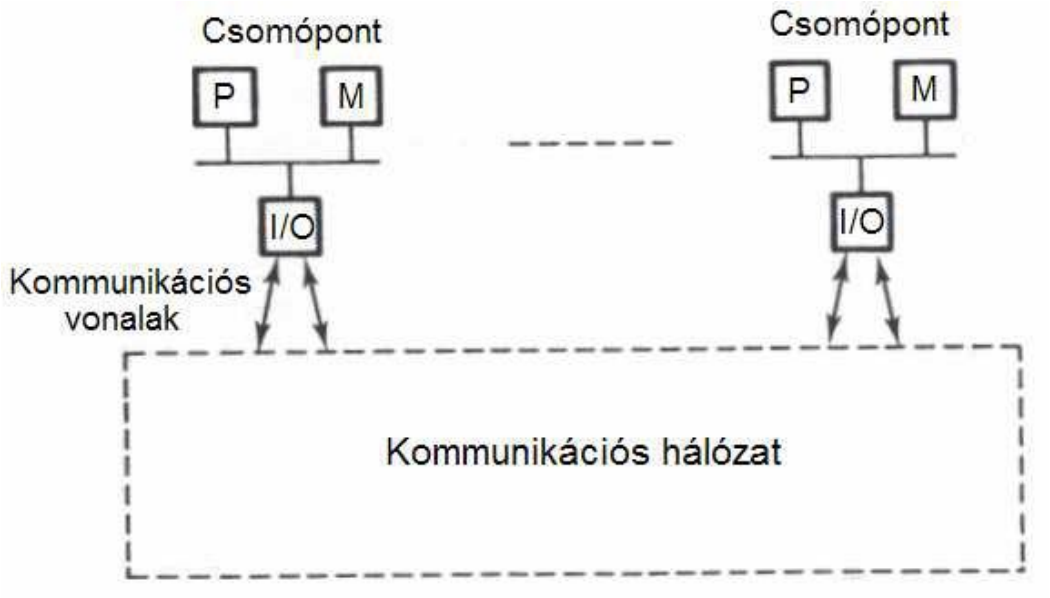
7.1 A multiprocesszoros rendszerek osztályozása

A multiprocesszoros rendszerek jellemzője, hogy egyidejűleg több független processzor dolgozza fel az alkalmazás adatait. Mindegyik processzor önálló programot hajt végre, de az alkalmazás megkívánja, hogy a folyamatok adatokat adjanak át egymásnak. Az adatátadás megvalósítása szempontjából, a multiprocesszoros rendszerek **két nagy csoportját** különböztetjük meg:

- **Megosztott memóriájú** (shared memory) rendszerek – a processzorok egy összekötő hálózaton (interconnection network) keresztül kapcsolódnak egy vagy több közösen használt memóriamodulhoz, amelyeken keresztül adatokat cserélhetnek:



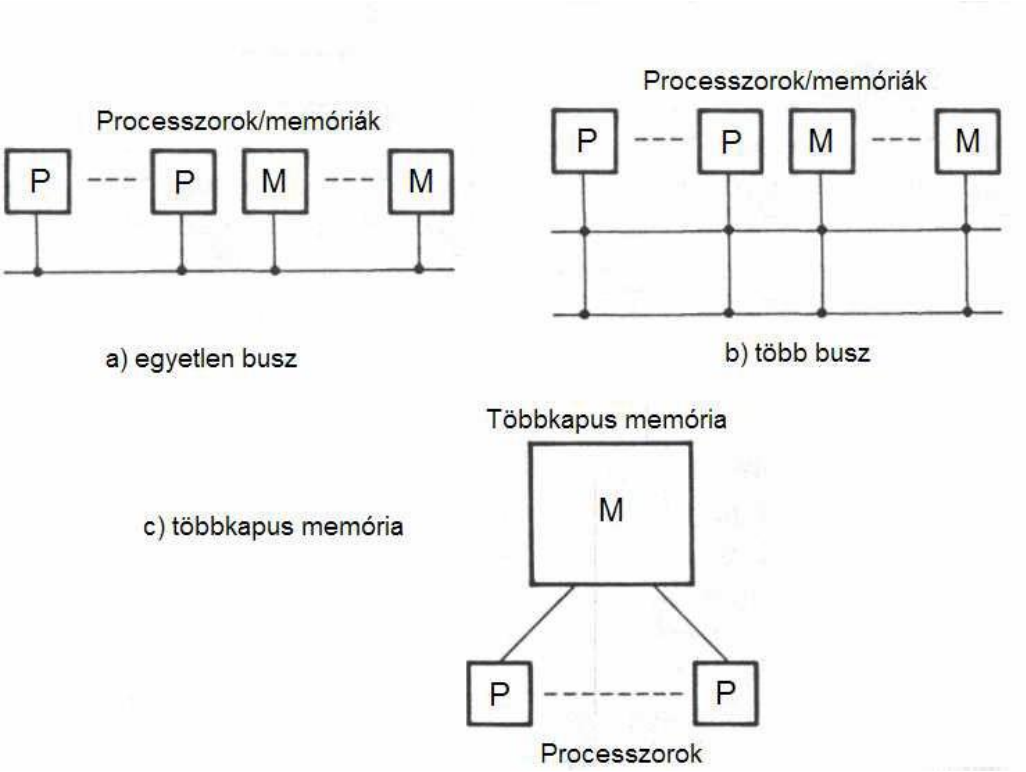
- **Elosztott (distributed)** vagy **üzenetádatásos** (message passing) rendszerek – mindegyik processzor rendelkezik csak saját memóriával, és üzenetek formájában cserél adatokat a többi processzorral, általában közvetlen kommunikációs vonalakon:



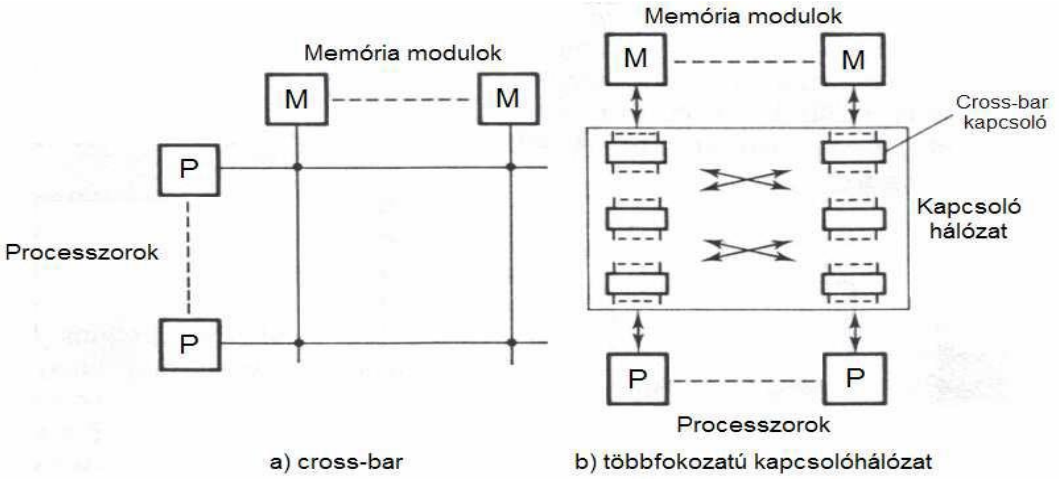
1. A **megosztott memóriájú** rendszereket két szempont szerint szokás osztályozni:

a) Az összekötő hálózat struktúrájának jellege alapján két típust különböztetünk meg:

- **Statikus** összeköttetésű rendszereket – amikor a hálózat nem tartalmaz kapcsolóelemeket. Az ilyen rendszer lehet
  - egy buszos (*single bus*);
  - több buszos (*multiple bus*);
  - többkapus memóriával (*multiport memory*) rendelkező.

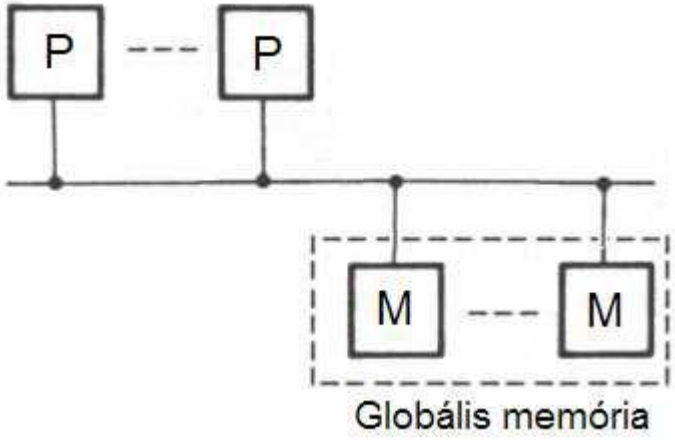


- **Dinamikus** hálózatú rendszerek – amikor a hálózat kapcsolóelemeket tartalmaz. Ez lehet
  - egyfokozatú kapcsolóhálózat, ami a *cross-bar*;
  - többfokozatú kapcsolóhálózat (*multistage nterconnection network*).



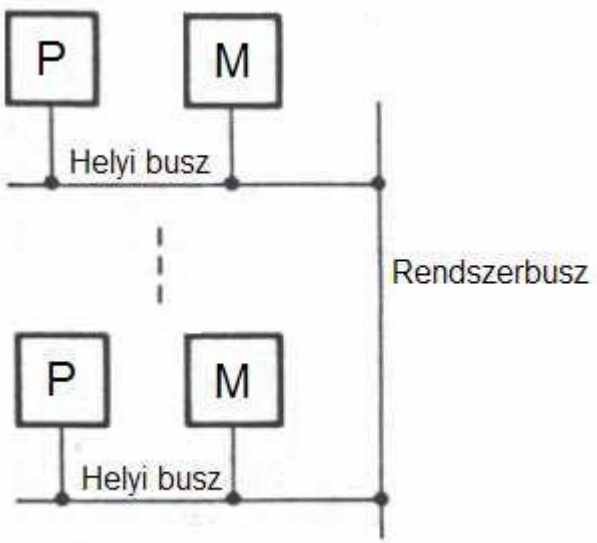
b) A megosztott memóriához való hozzáférés szerint léteznek:

- **UMA** (*Uniform Memory Access*) rendszerek – amelyeknél a közös memóriaterülethez való hozzáférés ideje azonos minden processzor számára (ezeket szokták még **szimmetrikus** multiprocesszoros rendszereknek is nevezni). Példaként megemlíthető az egyetlen globálisan elérhető memóriával rendelkező rendszer:



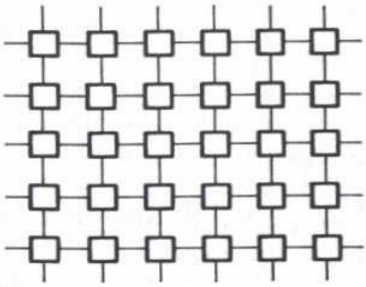
- **NUMA** (*Non-Uniform Memory Access*) rendszerek – amelyeknél a közös memóriához való hozzáférési idő eltér a processzorok között, attól függően, hogy egy bizonyos processzor a megosztott

memória mely területére hivatkozik. Például ilyen rendszert kapunk, ha mindegyik processzor rendelkezik lokális memóriával, amelyet viszont megoszt a többi processzorral. Ekkor a hozzáférési idő a saját memóriájához rövidebb lesz, mint a többi memóriához:

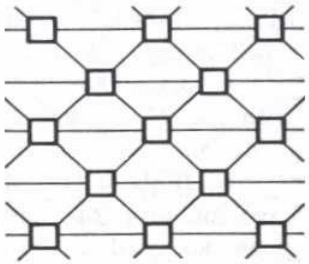


2. Az **elosztott (üzenetátadásos)** rendszereket a következőképpen osztályozhatjuk:

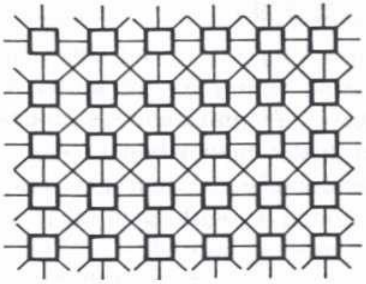
- **Erősen párhuzamos processzorok** (MPP – *Massively Parallel Processors*) – amelyek szabályosan rögzített összekötési hálózattal rendelkeznek, mint például
  - négy-, hat-, nyolckapcsolatú háló (a négykapcsolatúra példa a *transzputer*)
  - teljesen összefüggő hálózat
  - kocka, hiperkocka
  - fa struktúra



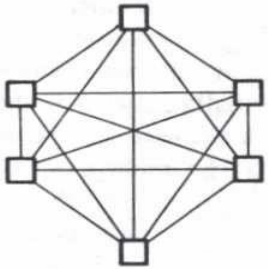
a) négykapcsolatú háló



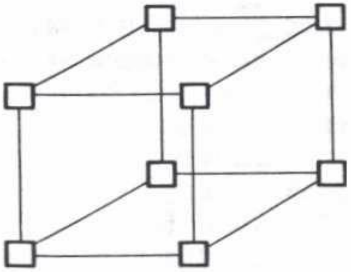
b) hatkapcsolatú háló



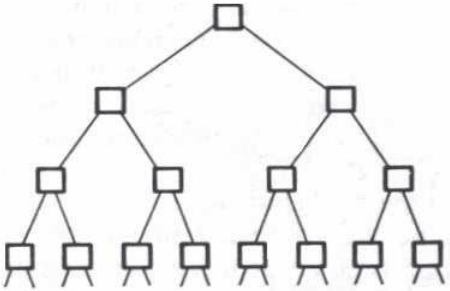
c) nyolckapcsolatú háló



d) teljesen összefüggő hálózat



e) kocka

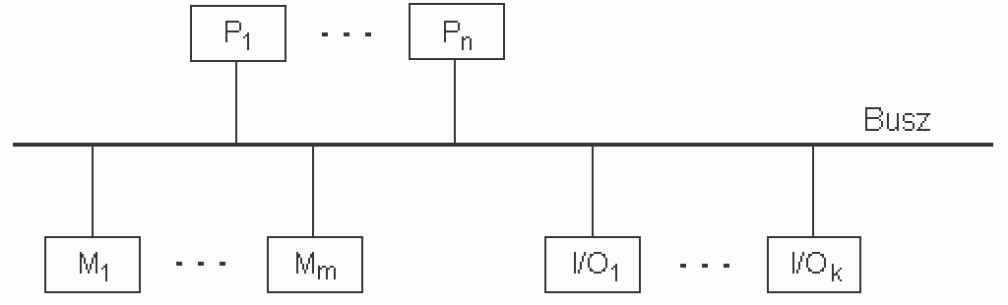


f) fa

– **Klaszterek** – amelyek egymással egy hálózaton (általában helyi hálózaton, LAN-on) keresztül összekapcsolt teljes számítógépek, és együttműködésükkel egyetlen erőforrás illúzióját keltik. Ilyen például egy munkaállomásokból álló COW (*Cluster of Workstations*). Magas rendelkezésre állásuk, skálázhatóságuk és teljesítőképességük miatt szerverek kialakítására is alkalmasak.

7.2 Egybuszos rendszerek jellemzői

Ebben az esetben a rendszer moduljai (processzor, memória, I/O) egyetlen buszhoz csatlakoznak:



A memória és I/O modulok minden processzor számára elérhetők, ezeket megosztottan használhatják. (Ez nem zárja ki, hogy a processzormoduloknak legyenek saját memória és I/O erőforrásai is.) A modulok két csoportba sorolhatók:

- ***Master*** (mester) modulok – amelyek képesek egy adatátvitelt vezérelni a buszon, miután átvették ennek használatát. Ilyenek a processzor- és a DMA- modulok.
- ***Slave*** (szolga) modulok – amelyek passzív elemekként vannak a buszra kötve, és várják, hogy egy master kezdeményezzen velük adatátvitelt, amelyre válaszolnak. A slave modulok tipikus példái a memória- és az I/O modulok.

Mivel a buszt egy adott pillanatban csak egy master használhatja, a buszidőt meg kell osztani a masterek között (*time sharing*). Annak eldöntésére, hogy az igénylők közül ki kapja meg a buszhasználat jogát, valamilyen arbitrációs mechanizmust használnak.

Az egybuszos multiprocesszoros rendszer tervezése és implementálása egyszerű és nem költséges feladat. Könnyen konfigurálható az igényeknek megfelelő rendszer, amely a későbbiekben továbbfejleszthető. A legnagyobb gon-

dot az egyetlen adatútvonal – a busz – magas forgalma jelenti, ami a rendszerbe kapcsolható processzor modulok számát korlátozza. Ha mindegyik processzor állandóan igényelné a buszt, akkor semmilyen hatékonysága sem volna a több processzor alkalmazásának, mivel csak az egyik használná a buszt, a többi pedig várakozna. Ha statisztikailag feltételezzük, hogy a processzorok egy hosszabb T időszakból egyformán t_b ideig igénylik a buszt, akkor elvileg mindegyiknek biztosított a hozzáférése mindaddig, amíg $n\,t_b : 5\,T$, ahol n a processzorok száma. A

$$k_b = \frac{t_b}{T}$$

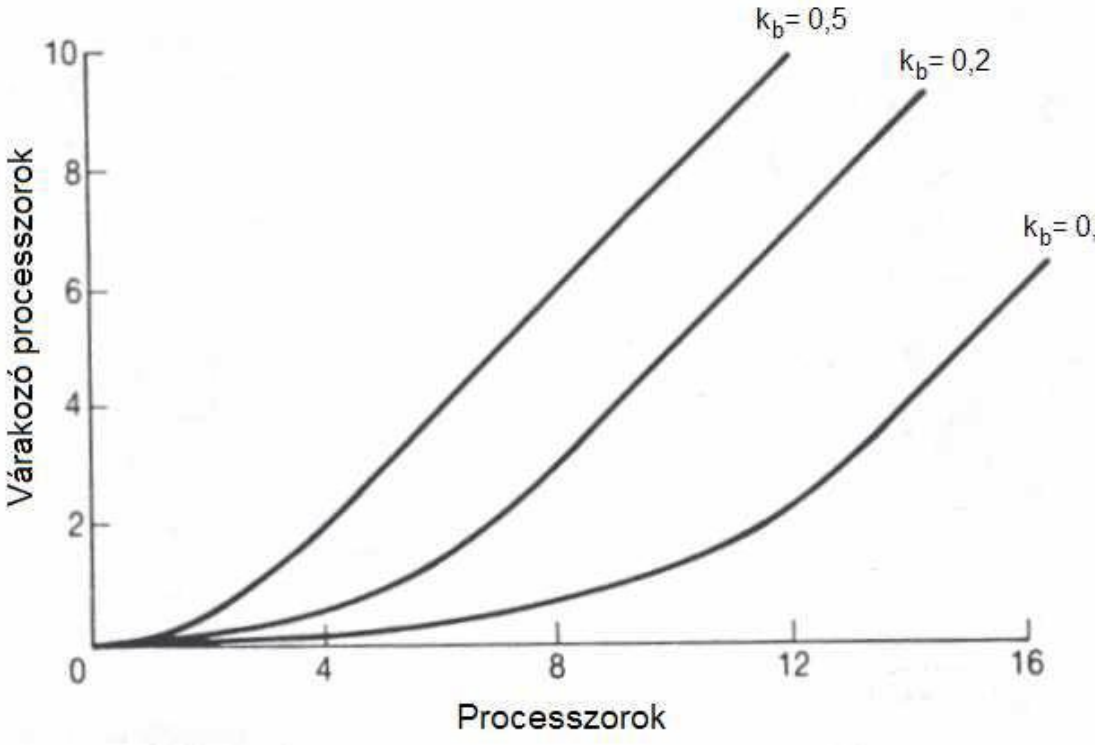
arányt buszhozzáférési együtthatónak nevezzük – értéke egy-egy processzornál annál kisebb, minél nagyobb a program által diktált autonómiája. Akkor, amikor egy újabb processzort hozzáadva a rendszerhez, az egyik processzornak már nem jut hozzáférési idő a buszhoz (várakoznia kell), elértük **a busz telítettségét** (*bus saturation*). A telítettségi küszöbnek megfelelő processzorszám:

$$n_s = \frac{1}{k_b}$$

A várakozásra kényszerülő processzorok számát (n_w) a következőképpen számíthatjuk ki:

$$n_w = \begin{cases} 0 & \text{ha} \quad n \le \dfrac{1}{k_b} \\ n - \dfrac{1}{k_b} & \text{ha} \quad n > \dfrac{1}{k_b} \end{cases}$$

Ennek a függvénynek a grafikonját – különböző k_b értékek számára – az alábbi ábra mutatja, azzal a megjegyzéssel, hogy a valós görbék nem törtvonalak, mert a gyakorlatban a telítettség elérése előtt is vannak várakozó processzorok, ha többen igénylik egyszerre a buszt:

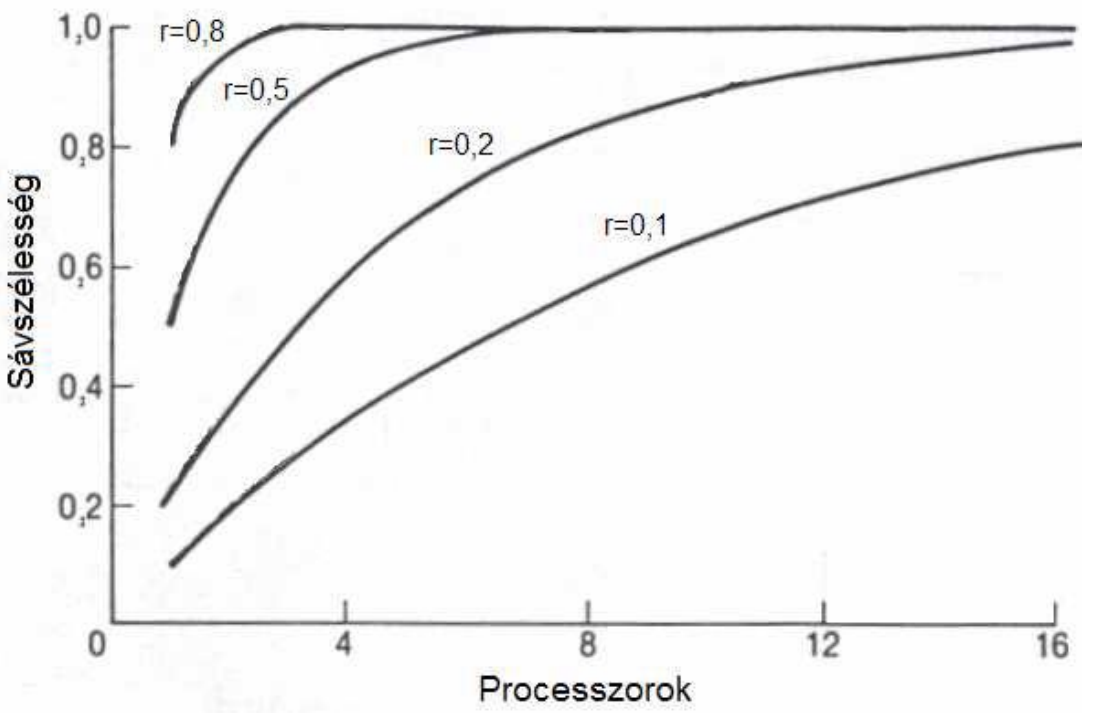


Az egybuszos rendszer tervezésénél figyelembe kell venni, hogy a buszhoz csatlakoztatott processzorok száma ne vezessen telítettséghez. Telítettség után ugyanis újabb processzorok csatlakoztatása már nem növeli a rendszer hatékonyságát, nem vezet további párhuzamos feldolgozáshoz, csak növeli a várakozó processzorok számát.

Könnyen kiszámíthatjuk **a busz sávszélességét** (*bus bandwidth*) is, azaz a buszt átlagosan használó processzorszámot. Feltételezzük, hogy a processzszorok véletlenszerűen generálnak hozzáférési kérést a megosztott memóriához, és egy processzor által kezdeményezett kérés valószínűsége r (hozzáférési ráta) Annak a valószínűsége, hogy a processzor ne kérjen hozzáférést $1-r$, annak pedig, hogy egy processzor se kérjen $(1-r)^n$. Továbbá, annak a valószínűsége, hogy legyen legalább egy kérés $1-(1-r)^n$. Ez az érték éppen a busz átlagos foglaltságát, azaz ennek a sávszélességét jelenti:

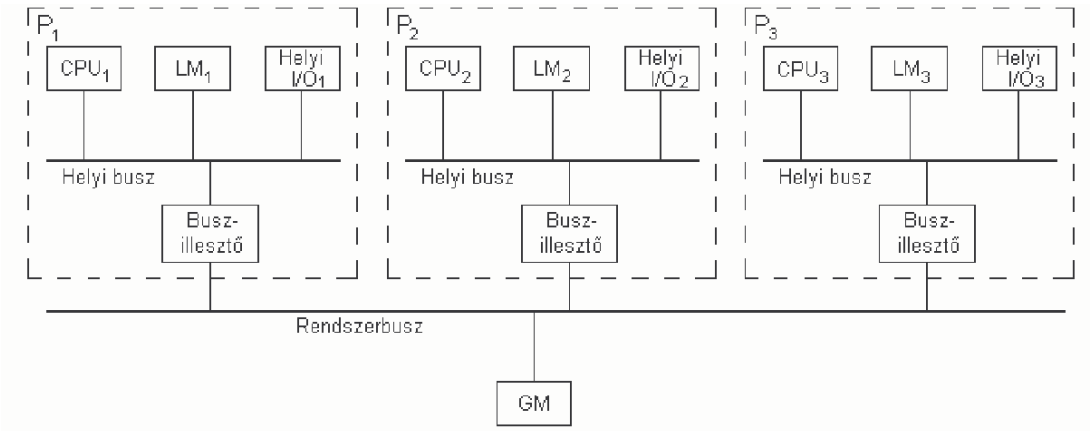
$$BW = 1-(1-r)^n$$

A sávszélesség grafikonját az alábbi ábra tartalmazza. Látható, hogy minél nagyobb a hozzáférési ráta, annál hamarabb telítődik a busz.

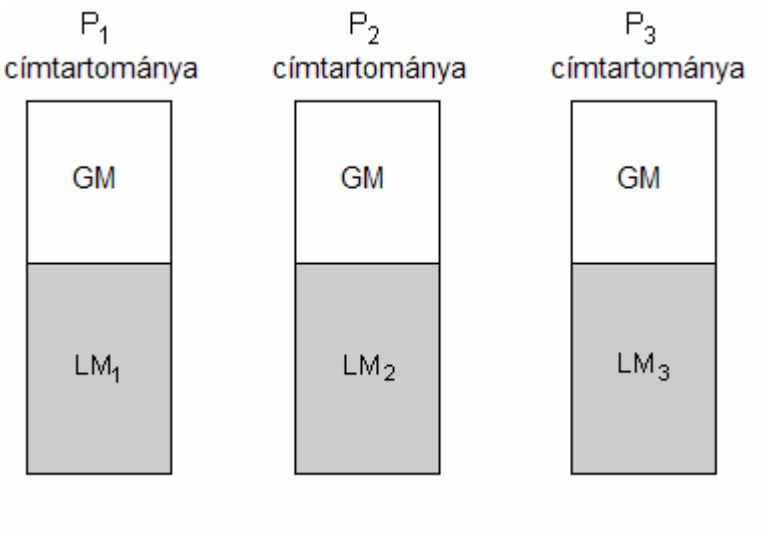


7.3 Egybuszos rendszerek címkiosztása

A buszhoz csatlakoztatott összes memóriamodul egyetlen globális, megosztott memóriát (GM) alkot, amelyhez mindegyik processzor hozzáférhet. A globális memóriát célszerű a közös adatok tárolására használni. Ezen kívül, minden egyes processzor rendelkezik egy saját, lokális memóriával (LM), amelyet a saját, helyi sínén érhet el. A két sín közötti kapcsolatot egy buszillesztő egység biztosítja:



Két processzor közötti adatátvitel két lépésben zajlik, két hozzáféréssel a rendszerbuszhoz: a küldő processzor beírja az adatot a globális memóriába, majd a célprocesszor beolvassa onnan egy újabb buszhozzáféréssel („postafiók" elv). Ahhoz, hogy mindkét processzor hozzáférhessen a közös memóriához, ennek ugyanazt a területet kell elfoglalnia mindkét processzor címtartományában. A címtartomány fennmaradó területe a processzorok lokális memóriája számára marad meg. Például, egy három processzormodult tartalmazó rendszer esetében, a processzorok címtartományát az alábbi ábra szerint lehet kialakítani:



7.4 A buszhozzáférés

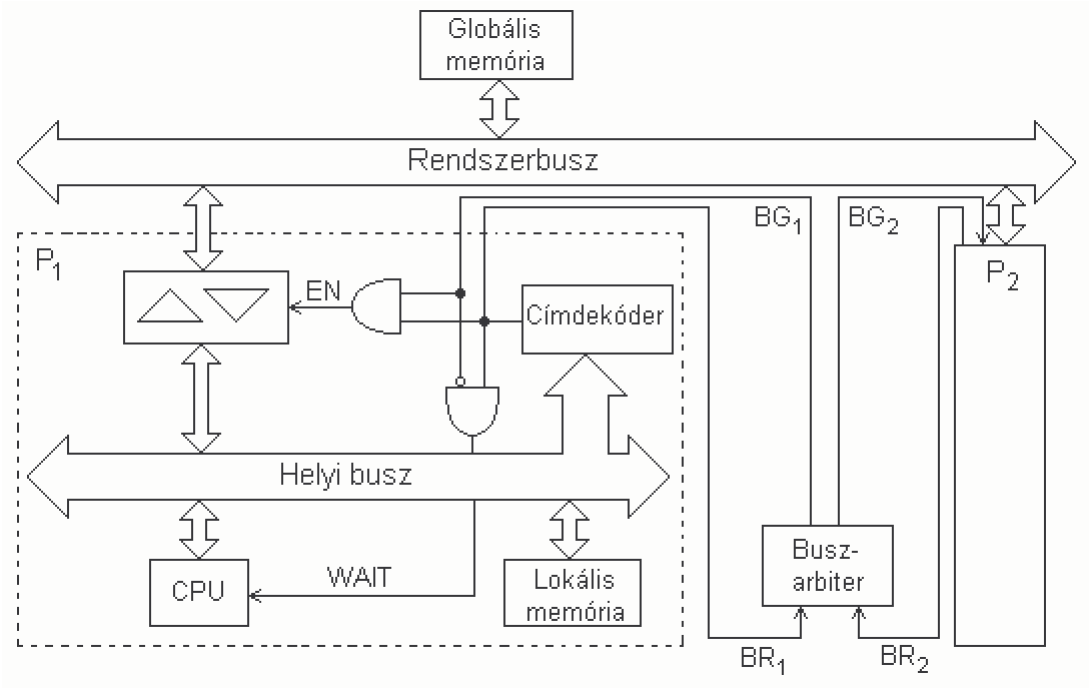
A processzor helyi sínje és a rendszerbusz között a kapcsolatot a **buszillesztő egység** biztosítja, ami a buszorientált multiprocesszoros rendszer tervezésének sajátos eleme. A buszillesztő egység fő feladata lehetővé tenni az illető processzor hozzáférését a rendszerbuszhoz egy adatátvitel (írás, olvasás) végrehajtása céljából a globális memóriával. A processzor akkor fog buszhozzáférést igényelni az illesztőtől, ha a programja végrehajtása során egy olyan adatátviteli utasításhoz ér, amelynek forrása vagy célja a globális memóriában van.

A buszillesztő egység két fő elemet tartalmaz:

- **Vezérlő-logikát** – amely figyeli és dekódolja a helyi buszon lévő címet, hogy eldöntse, mikro vonatkozik a külső erőforrásra
- **Puffereket** – amelyek a két busz azonos vonalai közötti jelátvitelt biztosítják. Hozzáférés hiányában a pufferek kimenetei magas impedanciájú állapotban vannak.

Buszhozzáférési igény esetén, ha minden feltétel teljesült (a busz szabad és nincs más, magasabb prioritású igénylő) a vezérlő logika kiadja a hozzáférést visszaigazoló jelet, ami megnyitja a puffereket és így az átvitel megvalósulhat.

A buszhozzáférés egyszerűsített vázlatát mutatja a következő ábra:

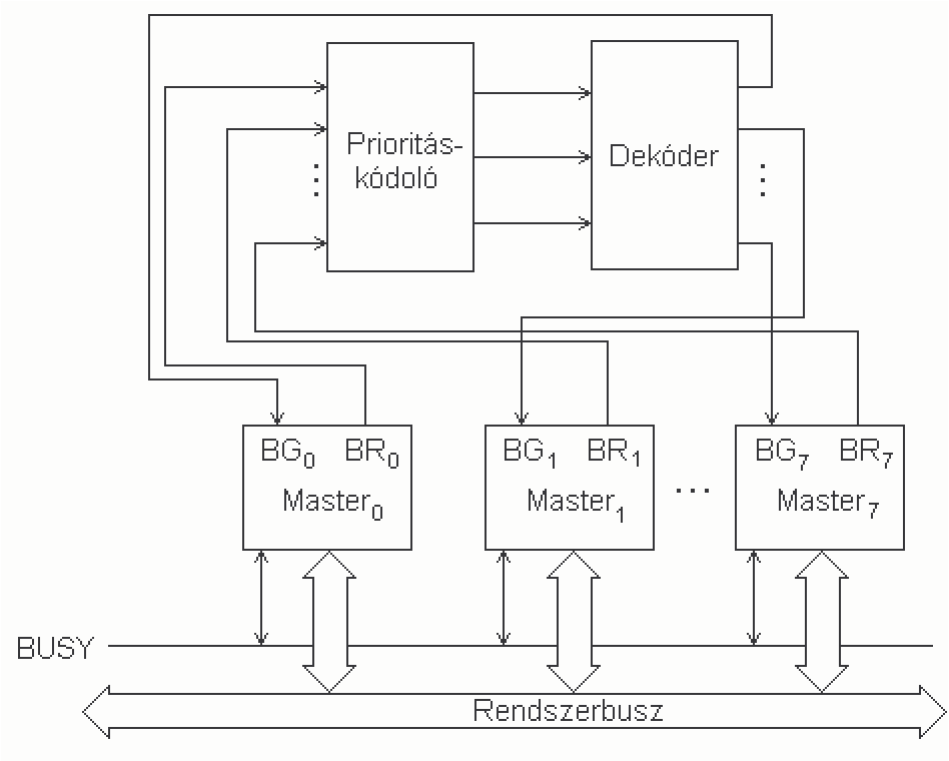


Amikor a dekóder egy, a megosztott memória-tartományba tartozó címet ismer fel, akkor egy buszkérelmet generál (BR – *Bus Request*) a buszarbiter felé. Ameddig a hozzáférési jogosultság nem valósul meg és a busz nem szabadul fel, a processzort várakozó állapotba kell kényszeríteni a WAIT jellel. Egyidejű igények esetén a buszarbiter a legnagyobb prioritással rendelkező processzormodul felé fog visszaigazoló jelet (BG – *Bus Grant*) küldeni. Ennek a jelnek az aktiválása után megtörténik a pufferek megnyitása (EN=1) és a processzor kikerül a várakozó állapotból, befejezi a buszciklust.

A buszarbitráció megvalósítására két módszert alkalmazhatunk:

- **Párhuzamos arbitráció** – amikor minden master egyszerre történik az igények elbírálása valamilyen prioritási algoritmus szerint (fix, forgó, LRU stb.);
- **Soros arbitráció** – amikor a masterek láncba vannak kapcsolva a prioritásuk csökkenő sorrendjében, és ebben a sorrendben történik az igények feldolgozása.

A párhuzamos arbitrációra mutat példát a következő ábra:



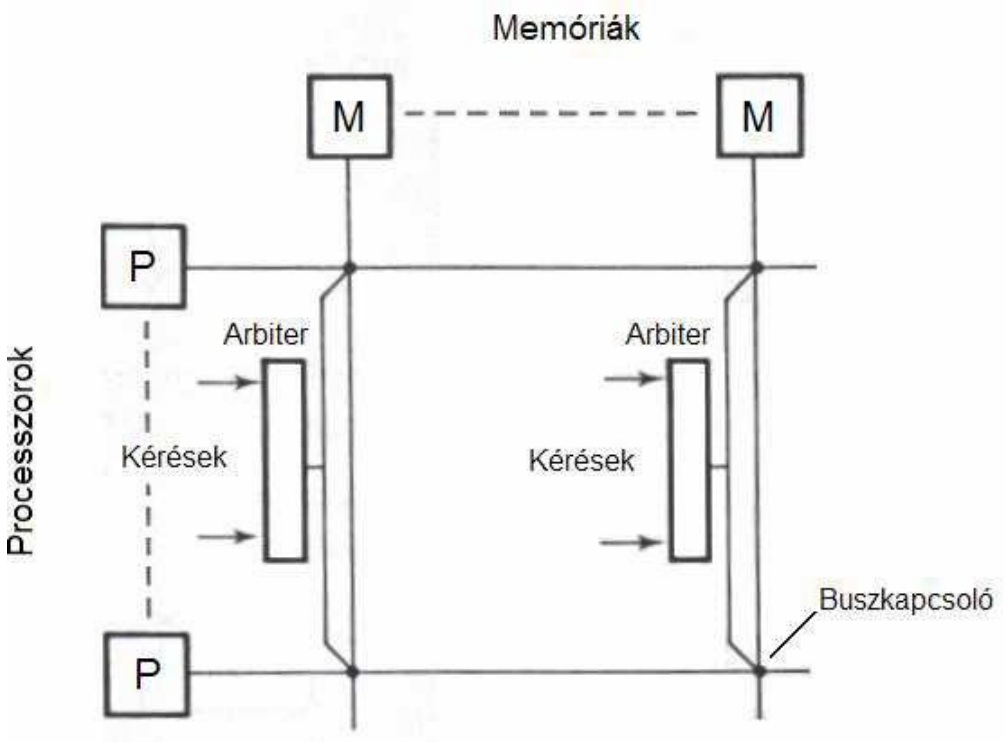
A buszarbitert egy prioritáskódoló és egy dekódoló áramkör valósítja meg. A prioritáskódoló bemenetei rögzített, csökkenő sorrendben lévő prioritással rendelkeznek, ide érkeznek a buszigények (BR). A kimeneteken az aktív bemenetek közül a legmagasabb prioritással rendelkezőnek a sorszáma jelenik meg. Ez a sorszám kiválasztja a dekóder egyik kimenetén azt a buszhozzáférést engedélyező jelet (BG), amelyik a legmagasabb prioritással rendelkező masterhez tartozik az igénylők közül. A hozzáférés másik feltétele, hogy az előző átvitelt végrehajtó master szabadítsa fel a buszt (BUSY=0).

7.5 A cross-bar-rendszer

Ebben a rendszerben a processzorok és a memóriák egy mátrix alakú kapcsolóhálózattal vannak összekötve úgy, hogy mindegyik processzor-memória összeköttetésre jut egy kapcsoló. Így egyszerre minden processzor dolgozhat,

természetesen külön-külön memóriával – ezért az ilyen fajta rendszert *blokkolásmentesnek* nevezik.

Az elemek közötti összekötő vonalak valóban buszokat jelentenek, így egy-egy kapcsolónak 50-100 jel továbbítását kell kezelnie. Minden memóriamodulhoz tartozó oszlopnak van egy saját arbitrációs logikája, amely eldönti, hogy melyik processzor jogosult az illető memóriához való hozzáférésre. A döntésnek megfelelően fog „bezárni" az igénylők közül legnagyobb prioritással rendelkező processzor sorában lévő kapcsoló. A processzorok memóriahozzáférési igényei az illető memóriamodulhoz tartozó arbiterhez futnak be:

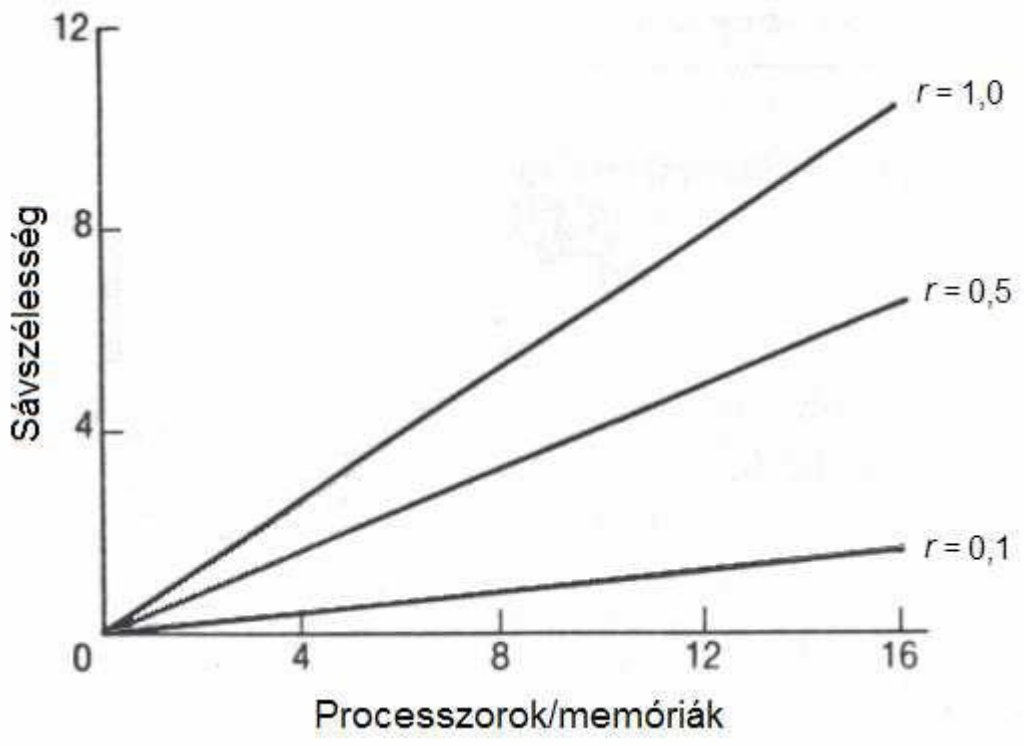


A **cross-bar-rendszer sávszélességének** kiszámításához legyen r annak a valószínűsége, hogy egy processzor kérést intéz a megosztott memóriához. Ha m a memóriamodulok száma, akkor annak a valószínűsége, hogy ez a kérés egy bizonyos modulra vonatkozzon r/m. Annak a valószínűsége, hogy a kérés nem egy bizonyos memóriamodulra vonatkozik $1–r/m$. Ha p a pro-

cesszorok száma, akkor annak a valószínűsége, hogy egy processzor sem hivatkozik erre a bizonyos memóriára $(1-r/m)^p$. Ebből következik, hogy $1-(1-r/m)^p$. lesz annak a valószínűsége, hogy legalább egy processzor hivatkozik erre az egy memóriamodulra. Most már kiszámítható a cross-bar rendszer sávszélessége, azaz az aktív (hivatkozott, tehát adatátvitelt lebonyolító) memóriamodulok száma:

$$BW = m\left(1-\left(1-\frac{r}{m}\right)^p\right)$$

A sávszélesség grafikonja az alábbi ábrán látható (abban az egyszerűbb esetben, amikor $p=m$). Megfigyelhető, hogy ha minden ciklusban történik memóriahivatkozás ($r=1$), akkor is maradnak várakozó processzorok (az egyenes nem 45 fokos), mert több processzor is hivatkozhat ugyanarra a memóriamodulra.



7.6 Többfokozatú kapcsolóhálózatok

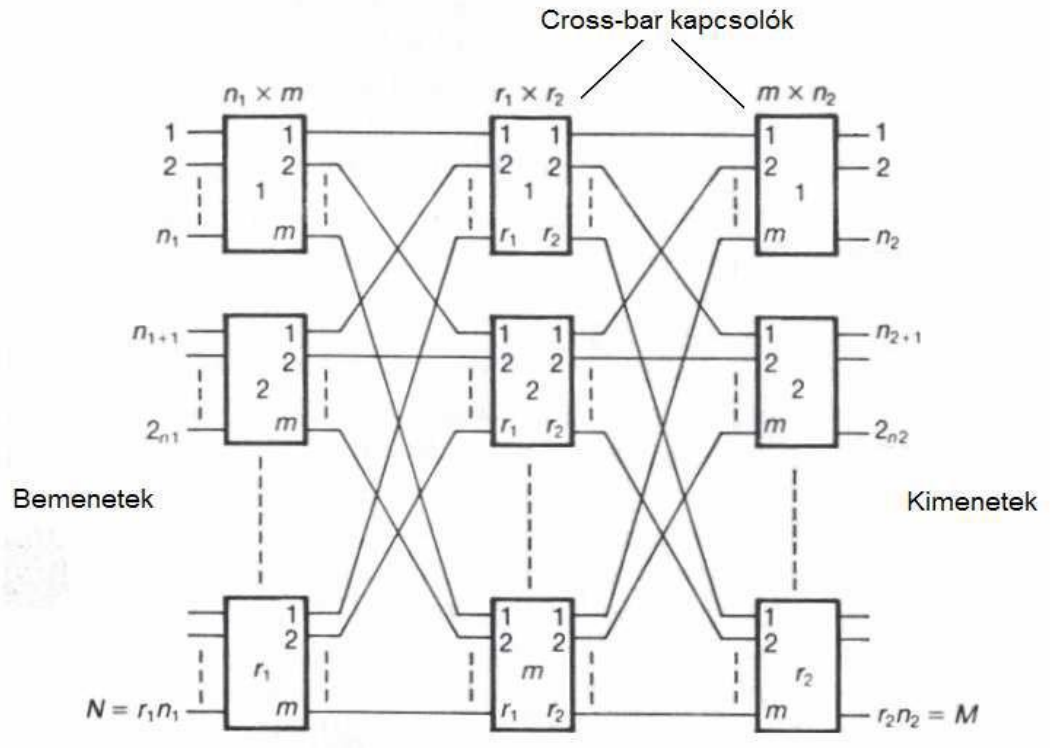
Ezekben a dinamikus hálózatokban (angolul *Multistage Interconnection Networks*) két csomópont (processzor, processzor/memória) közötti összeköttetést több szinten elhelyezett kapcsolók biztosítják, amelyeket különböző kapcsolatok létrehozására lehet beállítani. A bementtől a kimenetig haladó adatok több kapcsolófokozaton keresztül haladnak át. (Természetesen a kapcsolat lehet kétirányú is.) A hálózat lehet:

- **blokkolásmentes** – ha egyidejűleg minden egyes bemenet összekapcsolható egy-egy kimenettel, és ezt az összes bemenet/kimenet kombinációra meg lehet valósítani;
- **blokkoló** – ha nem teszik lehetővé az összes kombinációban az egyidejű bemenet/kimenet kapcsolatot.

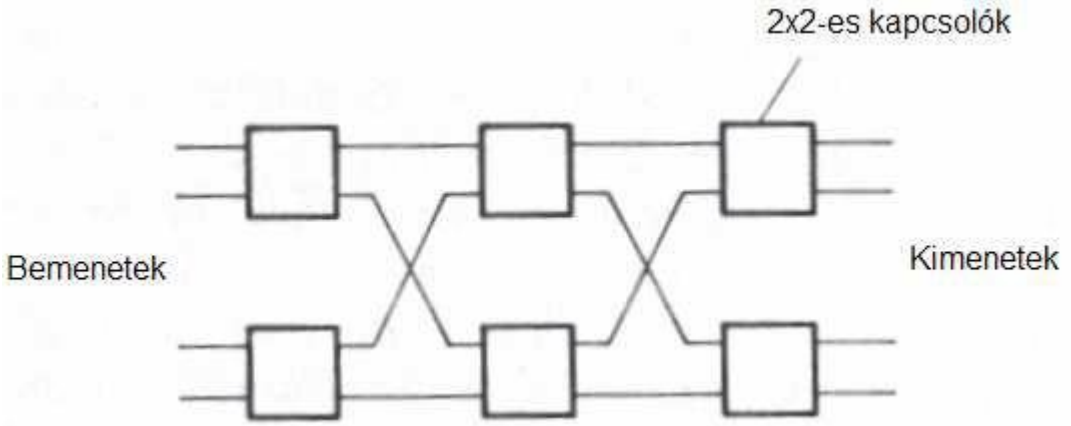
A többfokozatú kapcsolóhálózatnak két típusa van:

- Cross-bar kapcsolókkal ellátott hálózat
- Cella-alapú hálózat (ez az előbbinek a sajátos esete, 2×2-es cross-bar kapcsolókkal)

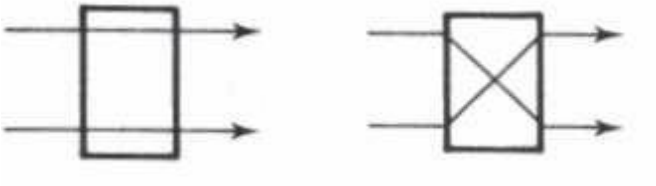
Az alábbi ábra egy háromfokozatú, cross-bar-kapcsolókkal kialakított **Clos hálózatot** mutat. A hálózat blokkolásmentes, ha $m \geq n_2 + n_1 - 1$.



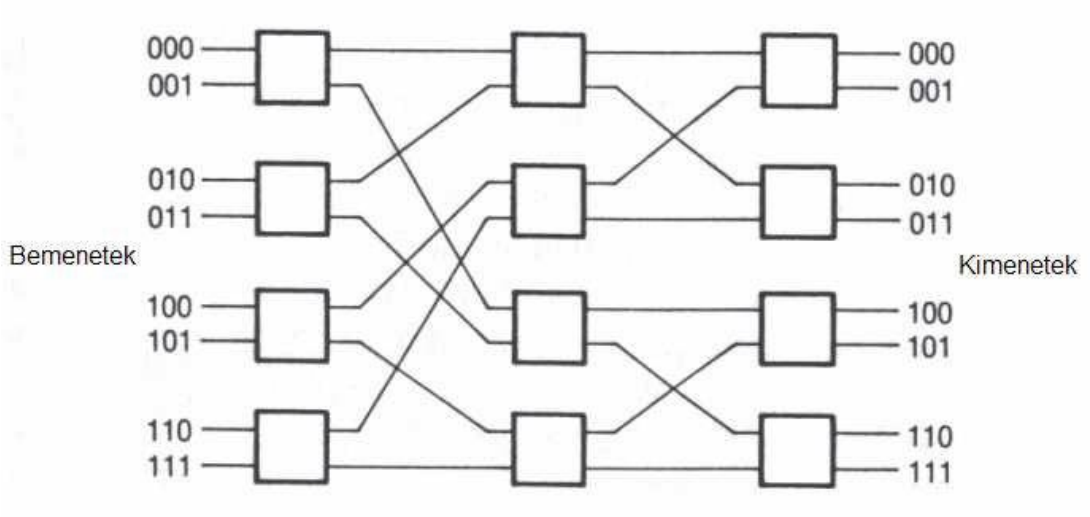
A Clos hálózat sajátos esete a két bemenetű és két kimenetű cella alapú **Benes hálózat**:



A cellák két állapotban lehetnek, párhuzamosan vagy keresztben kapcsolhatják össze a bemenetet a kimenettel:



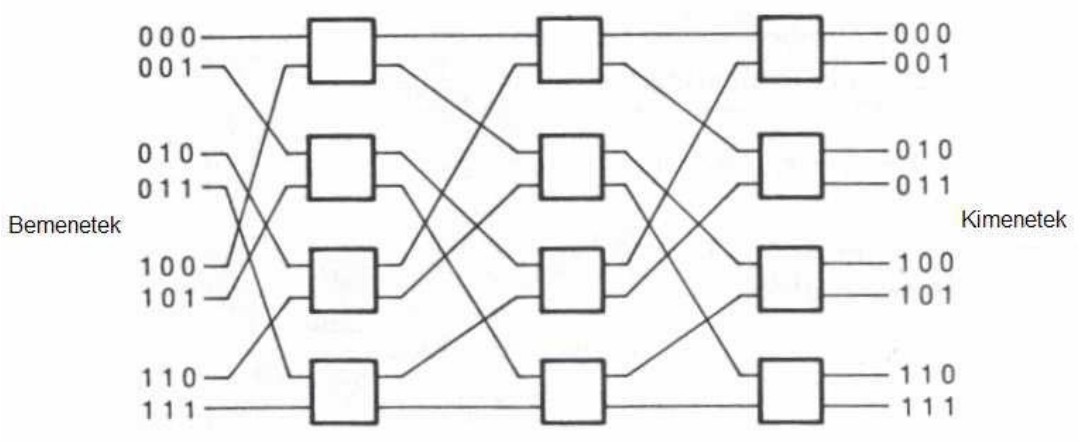
A cella alapú hálózatok általában blokkoló típusúak. Az alábbi ábrán bemutatott **baseline hálózat** is ilyen, de rendelkezik az automatikus útválasztás (*self-routing*) tulajdonságával: a célkimenet címének bitjei sorban a következő fokozat kapcsolóit irányítják (ha a bit=0, akkor párhuzamos kapcsolás, ha a bit=1, akkor keresztirányú kapcsolás). Ezekben a hálózatokban mindegyik bemenet és kimenet között csak egy útvonal lehetséges, ezért nem tolerálják a kapcsolók meghibásodását.



A baseliene hálózat minden egyes fokozata megfelezi a további lehetséges útvonalakat: az első fokozat után az adat a hálózat felső vagy az alsó felébe kerül, a második után valamelyik negyedbe, és végül a célba. Ha a bemenetek és kimenetek száma $N=2^n$, akkor a hálózat fokozatainak száma $\log_2 N$.

A többfokozatú kapcsolóhálózatok kialakításának egy másik változata a tökéletes keverésű hálózat (*perfect shuffle network*). Ebben az esetben egy fokozat bemeneteinek első fele közé beékelik a bemenetek második felét. Ilyen

az **Omega hálózat**, amely blokkoló, de rendelkezik az önálló útvonalválasztás tulajdonságával:



A **többfokozatú kapcsolóhálózat sávszélességét** a kapcsolócellákkal megvalósított típusokra számítjuk ki. Egy kapcsolócella lényegében egy 2□2-es cross-bar, tehát az előző alfejezetben meghatározott egyenleteket itt alkalmazhatjuk, azzal a pontosítással, hogy $p=m=2$. Ha a bemeneti processzorok a kimenetre kötött memóriamodulokra r_0 valószínűséggel hivatkoznak, akkor:

- az első fokozati kapcsoló egyik kimenetén a memóriahivatkozás valószínűsége

$$r_1 = 1 - \left(1 - \frac{r_0}{2}\right)^2$$

- a második fokozati kapcsoló egyik kimenetén a memóriahivatkozás valószínűsége

$$r_2 = 1 - \left(1 - \frac{r_1}{2}\right)^2$$

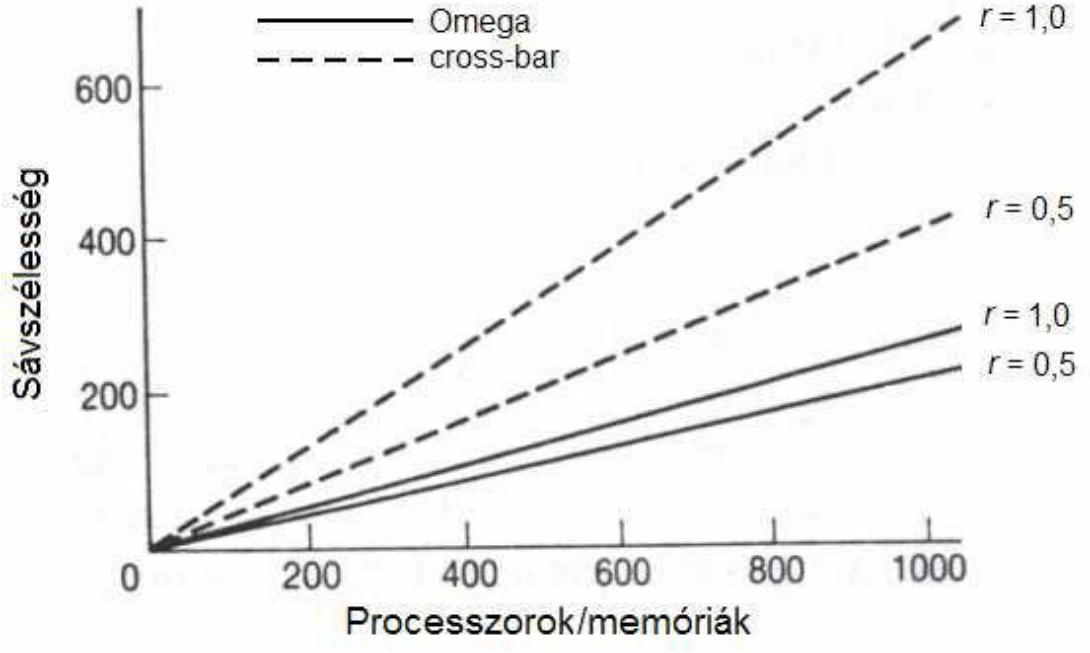
– az utolsó (n-edik) fokozati kapcsoló egyik kimenetén (azaz a hálózat egyik kimenetén) a memóriahivatkozás valószínűsége

$$r_n = 1 - \left(1 - \frac{r_{n-1}}{2}\right)^2$$

Most már kiszámítható a hálózat sávszélessége, vagyis hogy hány kimenetre (a 2^n- ből) történik hivatkozás:

$$BW = 2^n r_n$$

Az r_n kiszámítása rekurzívan történik, a fenti egyenletek alapján. Egy Omega hálózat sávszélességét a bemenetek számának függvényében mutatja az alábbi ábra. Összehasonlításként fel van tüntetve a hasonló számú bemenettel és kimenettel rendelkező csoss-bar hálózat sávszélessége is. Látható, hogy a többfokozatú hálózat sávszélessége kisebb, mert itt egyes memóriamodulokra való hivatkozás nem teljesíthető, a blokkolt útvonal miatt.



KÉRDÉSEK, TÉTELEK

7.1. Mit értünk megosztott memóriájú multiprocesszoros rendszeren? Vázolja fel az elvi felépítését, és sorolja fel a különböző típusait!

7.2. Mit értünk üzenetátadásos (elosztott) multiprocesszoros rendszeren? Vázolja fel az elvi felépítését, és sorolja fel a különböző típusait!

7.3. Vázolja fel az egybuszos multiprocesszoros rendszer felépítését! Mi a szerepük a master és slave moduloknak?

7.4. Mi a szerepe a buszillesztő egységnek? Milyen részegységeket tartalmaz, mi ezeknek a szerepe?

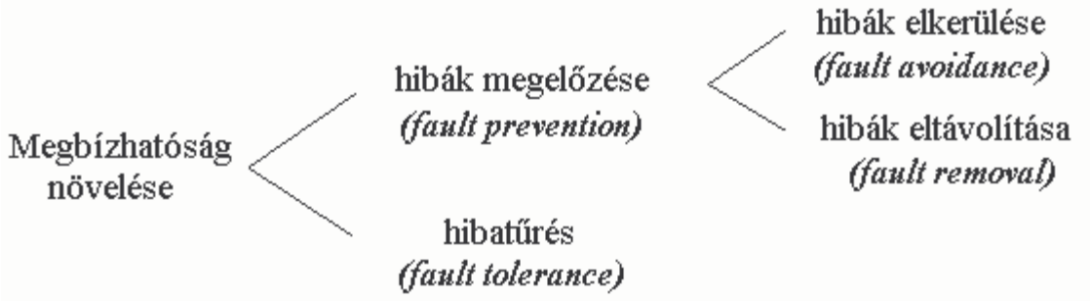
7.5. Magyarázza el a buszhozzáférés menetét, egy kisegítő ábrát is felhasználva!

7.6. Vázoljon fel egy cella alapú többfokozatú kapcsolóhálózatot! Mit jelent a blokkolásmentes hálózat? Hogy történik az automatikus útválasztás?

8. HIBATŰRŐ ARCHITEKTÚRÁK

8.1 A hibatűrés fogalma és alkalmazásai

A rendszerek meghibásodás nélküli működése olyan cél, amit sosem lehet elérni. Egyrészt a rendszer alkotóelemei hibásodhatnak meg, másrészt még egy tökéletesnek hitt terv is tartalmazhat felderítetlen hibákat. A magas megbízhatóság elérésének elvileg két lehetősége van: a hibák megelőzése (elkerülése a tervezés és implementálás közben, illetve eltávolítása a tesztelés során) és tolerálása (a működés közben).



A **hibatűrő rendszer** (*fault tolerant system*) képes ellátni a specifikált feladatait korlátozott számú hibás alegység (hardver és szoftver) jelenlétében is. A tolerálandó hibák lehetnek

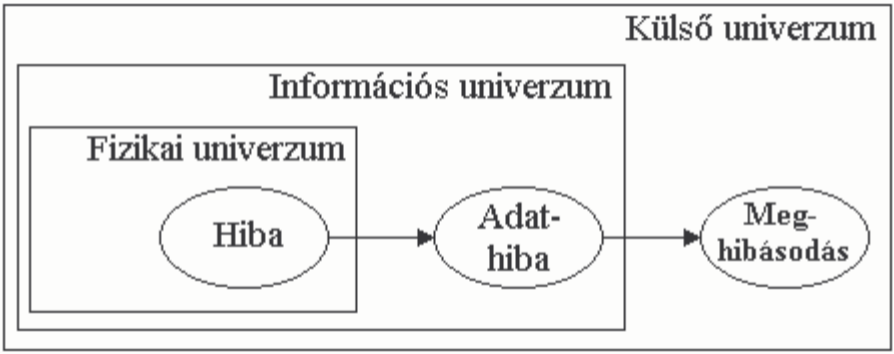
- Előre látott hibák (*anticipated faults*) – a meghibásodott hardver alkatrészekre vonatkoznak, ezekre lehet számítani
- Előre nem látott hibák (*unanticipated faults*) – a hardverben és szoftverben található tervezési hibák

Ha a rendszer működése nem a várakozásoknak megfelelően zajlik, akkor azt mondjuk, hogy a rendszer meghibásodott. A **meghibásodás** (*failure*) jelenti a rendszer viselkedésének első eltérését a specifikációtól. Ha már megtörtént, a meghibásodást nem lehet visszafordítani.

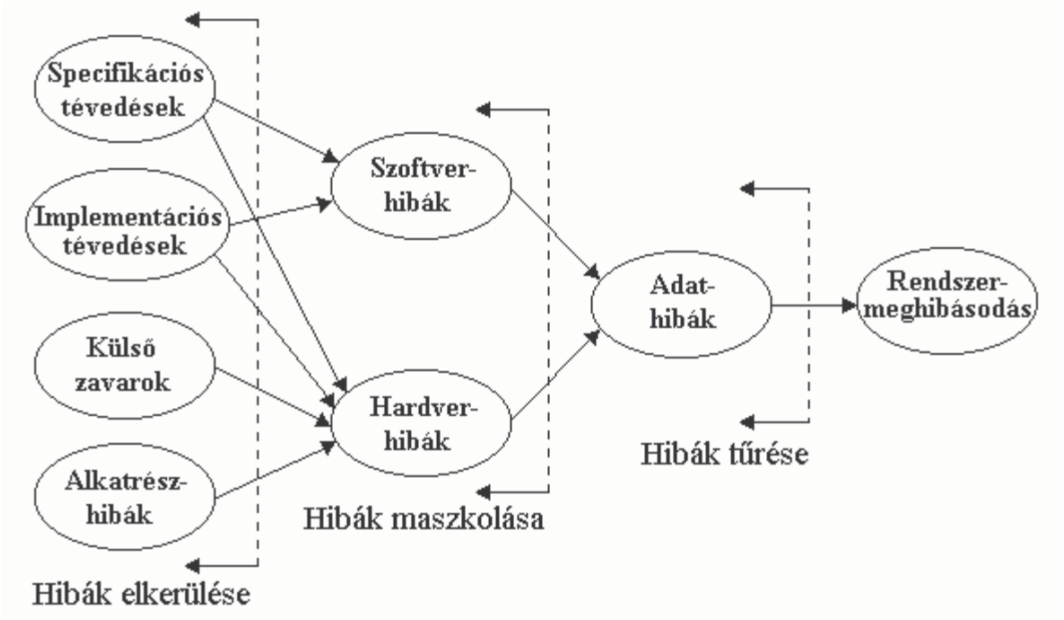
A működése közben a rendszer belső állapotok sorozatán megy át. Ha eközben fellép egy hibás átmenet, akkor a rendszer helytelen állapotba megy át, ami a további állapotváltások során a rendszer meghibásodásához vezethet. Egy rossz értéket a rendszer belső állapotában – például egy változó egy vagy több bitje fordított értéken áll – **adathibának** (*error*) nevezünk.

A helytelen állapotba való jutás oka lehet egy alkatrész meghibásodása (előre látható esemény), egy tervezési hiba a hardverben vagy a szoftverben, illetve egy kivitelezési hiba a megépítésben (előre nem látható események). A **hiba** (*fault*) tehát egy fizikai romlás a hardverben, vagy egy hiányosság a rendszer hardver vagy szoftver részében.

A rendszer valamely pontján fellépő hiba okozza az adathibát, ami a részegységek közötti adatcsere folytán elterjedhet, és végül a külső megfigyelő számára a rendszer meghibásodását eredményezheti:



A különböző hibák megnyilvánulása figyelembevételével határozhatjuk meg magas megbízhatóság elérésére alkalmazott eljárások korlátait (lásd az alábbi ábrát). A **hibatűrő mechanizmusok feladata** a hibákat még azelőtt felismerni, hogy a rendszer meghibásodásához vezetnének, és ennek megfelelően az adatokat helyes állapotba hozni, illetve – ha szükséges – kiiktatni a rendszerből a meghibásodott elemeket (újrakonfigurálás – *reconfiguration*).



A hibatűrő rendszerek néhány jellegzetes alkalmazási területe:

- **Kritikus rendszerek** az emberi életre, a környezetre, vagy értékes berendezésekre nézve, amelyek magas biztonságot (*safety*) igényelnek – űrhajók, repülők vezérlőrendszerei, vegyi vagy nukleáris folyamatok vezérlőrendszerei, intenzív terápiás egészségügyi berendezések
- **Hosszú élettartammal rendelkező rendszerek** – személyzet nélküli űrhajók, műholdak
- **Késleltetett karbantartású rendszerek** – repülőgépek, távfeldolgozó rendszerek
- **Magas rendelkezésre állású rendszerek** – szerverek, elosztott tranzakciós (például banki) rendszerek
- **Nagy teljesítőképességű rendszerek** – szuperszámítógépek, párhuzamos számítógépek

8.2 A redundancia formái

Redundancia alatt a rendszerhez hozzáadott össze pótlólagos elemet értjük, amelyekre a funkciók ellátásához nem lenne szükség egy hibamentes világban. Még Neumann János kimutatta, hogy a hibatűrő képesség eléréséhez redundanciára van szükségünk. Ennek több formája van:

- **Fizikai redundancia** (*physical redundancy*) – a hardver vagy szoftver erőforrások sokszorozását (*replication*) feltételezi a hibák felismerése és tolerálása céljából. Az alapgondolat alternatívák párhuzamos feldolgozásából, majd az eredmények összehasonlításából, esetleg többségi szavazás alá vételéből áll. Nevezik még **strukturális redundanciának** is.
- **Információs redundancia** (*informational redundancy*) – a hasznos adatokhoz pótlólagos adatok hozzáadását feltételezi úgy, hogy kódolás által lehetővé váljon a hibák detektálása és esetleg kijavítása.
- **Időbeli redundancia** (*temporal redundancy*) – a feldolgozáshoz pótlólagos idő hozzáadását feltételezi, aminek a végrehajtás megismétlése céljából van szükség. Első sorban átmeneti hibák kezelésére hasznos.

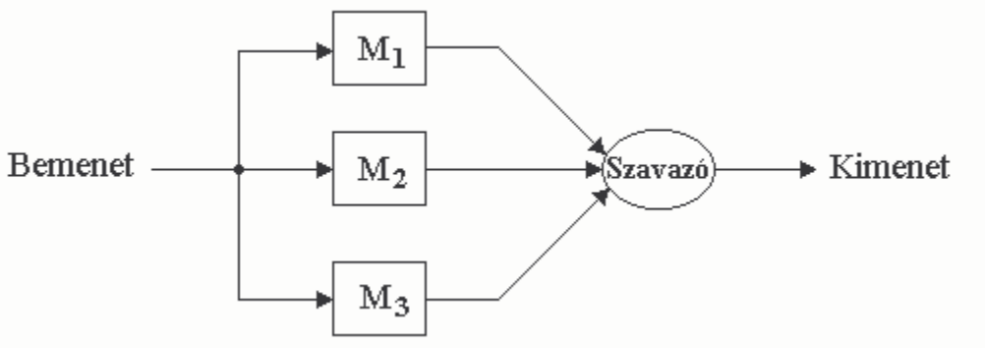
8.3 Hardver-redundancián alapuló technikák

8.3.1 Statikus redundancia

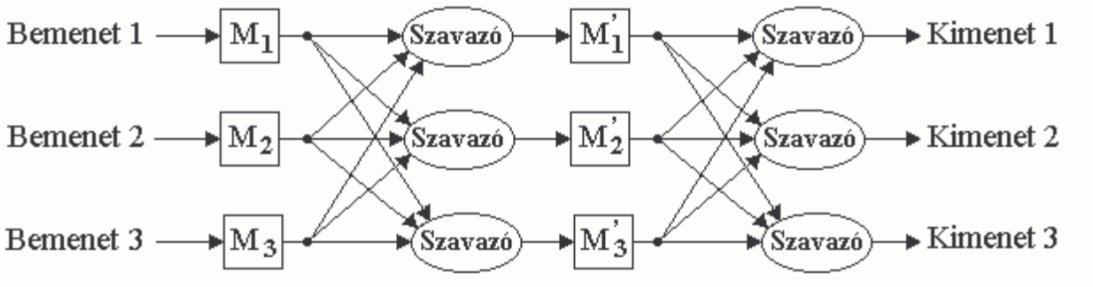
A statikus redundancia alapelve a **hibák maszkolása** (*fault masking*), ami így megakadályozza ezek átváltozását a rendszerben elterjedhető adathibákká. A technika lényege az erőforrás sokszorozásából áll, ami egy többségi szavazás követ.

A legegyszerűbb konfiguráció a **hármas moduláris redundancia** (TMR – *Triple Modular Redundancy*), amely három azonos modulból áll (lásd az

alábbi ábrát). A helyes működéshez elég legalább két működőképes modul, mert a kimeneti értékeket a szavazó összehasonlítja, és a többségnek megfelelő értéket választja ki. A rendszer egy modul meghibásodását tolerálja.



Hogy a szavazó meghibásodása ne vezessen a rendszer összeomlásához, kidolgozták a szavazó áramkör megháromszorozásával megépített rendszert (ez az adatútvonalak megháromszorozását is feltételezi):

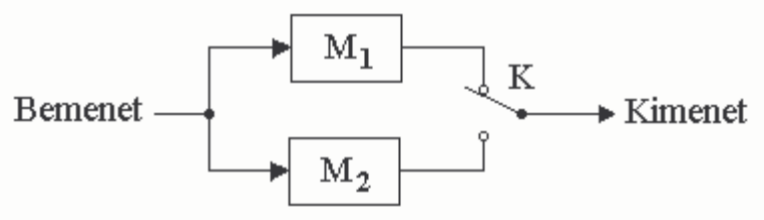


A TMR általánosítása az **N-moduláris redundancia** (NMR – *N-Modular Redundancy*), amely az alapmodul N darab másolatát használja (N páratlan szám). Ha $N=2p+1$, akkor a rendszer p modul vagy szavazó meghibásodását képes tolerálni.

A statikus redundanciát első sorban a kritikus rendszerekben alkalmazzák, ahol egy helytelen eredmény szolgáltatása – akár csak kis ideig – elfogadhatatlan. A hiba maszkolásához viszont jelentős plusz hardverre van szükség, ami ezt a technikát költségessé teszi.

8.3.2 Dinamikus redundancia

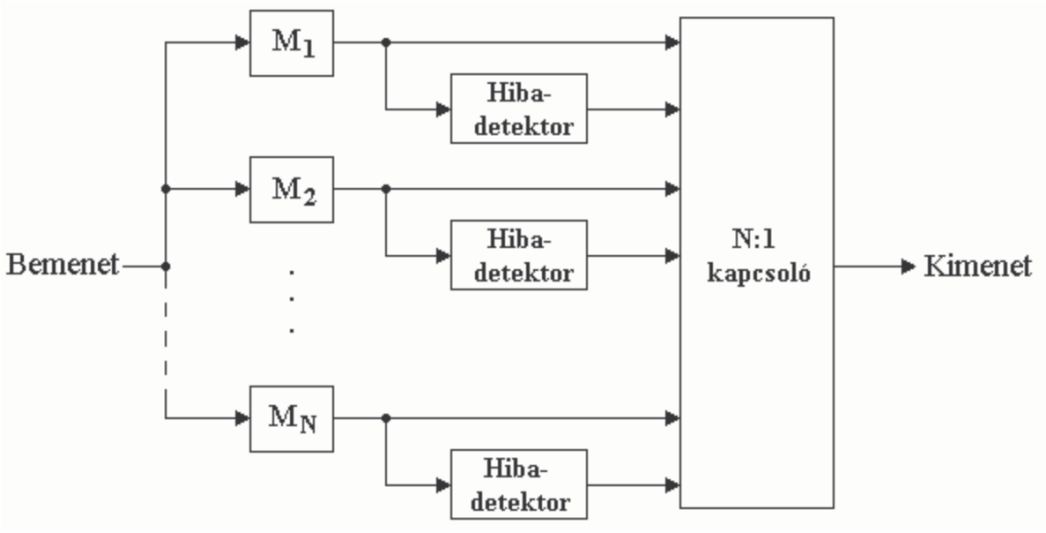
Ez a típusú redundancia, egy hiba felismerése után, a rendszer kimenetének egy tartalék (*standby*) modulra való automatikus átkapcsolására épül (a kapcsolót egy digitális áramkör valósítja meg):



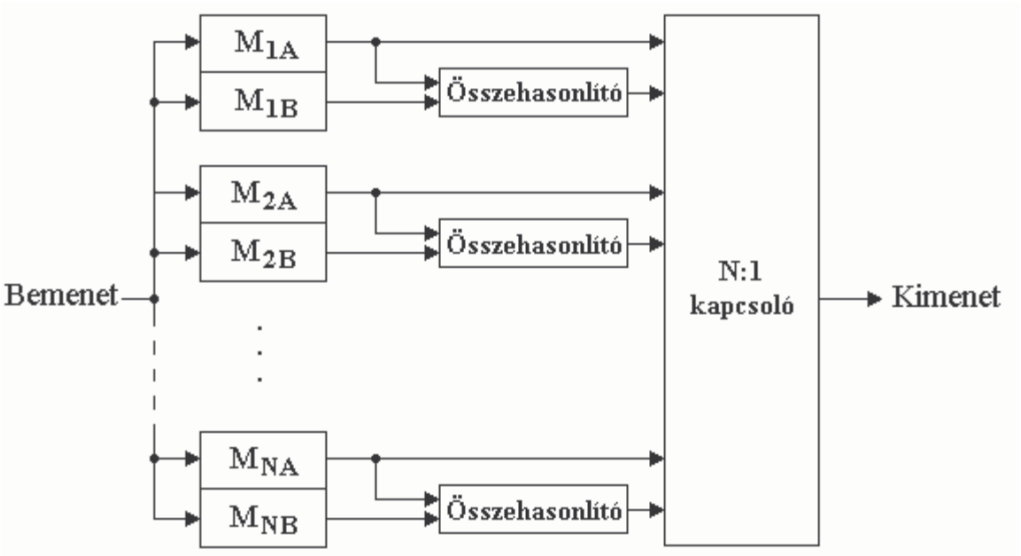
A megfelelő működés végett, szükséges egy hatékony hibadetektálási módszer implementálása. Erre több lehetőség van, mint a kimeneti érték abszolút tesztelése (például az eredmény határértékek közzé való illeszkedése), hibadetektáló kód alkalmazása, vagy *watchdog* (periodikusan újraindított időzítő) használata.

Íme két példa dinamikus redundanciát használó rendszerre:

- A ***standby sparing*** rendszerben az *N* egyforma modul egyike szolgáltatja a kimenetet, a többi *N*–1 készenléti tartalék. Ez a rendszer *N*–1 modul meghibásodását képes tolerálni. A megfelelő működést nagymértékben meghatározza a hibadetektor hatékonysága.



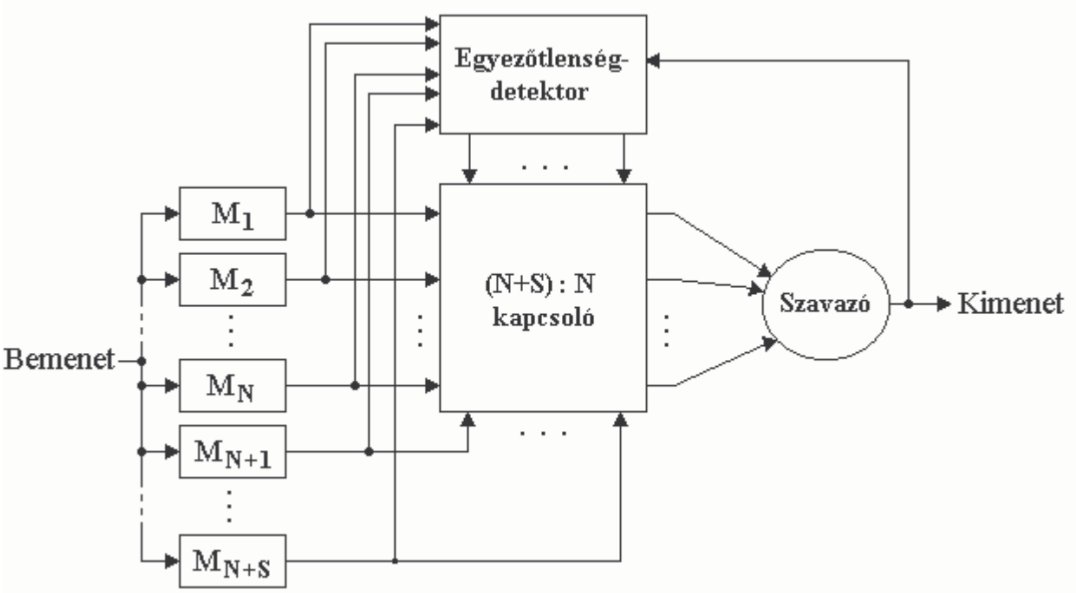
- A ***pair-and-spare*** rendszerben egyidejűleg két aktív modul van, amelyeknek az eredményeit összehasonlítják. Ha eltérés van, akkor az egész modulpárt kicseréli a kimeneti kapcsolóáramkör egy másikkal, a tartalékok közül. Ez a megoldás költségesebb, mint az előző, de az összehasonlítás alapú hibadetektálásnak igen magas a hatékonysága tetszőleges hibák esetén is.



Általában a dinamikus redundanciát a hosszú élettartammal rendelkező, valamint a magas rendelkezésre állást igénylő rendszereknél alkalmazzák, ahol ideiglenesen el lehet fogadni a helytelen kimenetet, de a működőképes állapotot hamar vissza lehet állítani a újrakonfigurálás által. Ez a technika kevesebb pótlólagos hardvert használ, mint a statikus, ezért annál olcsóbb.

8.3.3 Hibrid redundancia

A hibrid redundancia alapelve a **tartalékos N-moduláris redundanciában** (*NMR with spares*) testesül meg (lásd az alábbi ábrát). Egy N egyforma modulból álló mag mindig részt vesz a szavazásban, és meghatározza a rendszer kimenetét. Egy modul meghibásodása esetén megtörténik a rendszer újrakonfigurálása: a kapcsoló hálózat kicseréli a meghibásodott modult egy tartalékkal. Ha a rendszer $N+S$ modulból áll ($N=2p+1$), akkor maximálisan $p+S$ modul hibáját tudja tolerálni.



KÉRDÉSEK, TÉTELEK

8.1. Mik a számítógépek megbízhatósága növelésének módszerei, és ezeket mikor kell alkalmazni?

8.2. Melyek a hibatűrő rendszerekben alkalmazott redundancia formái, és ezek legfontosabb jellemzői?

8.3. Vázolja fel a TMR (hármas moduláris redundancia) rendszert, és magyarázza el a működését!

8.4. Vázolja fel a *standby sparing* architektúrát, és magyarázza el a működését!

8.5. Vázolja fel a *pair-and-spare* architektúrát, és magyarázza el a működését!

MODELLEZÉS ÉS SZIMULÁCIÓ

[Dr. Szász Gábor, Dr. Kun István és Dr. Zsigmond Gyula jegyzetei alapján]

9.1 A MARKOV-folyamatok elmélete

A hibatűrő rendszerek alkalmazásának elsődleges célja a megbízhatóság növelése azáltal, hogy akár meghibásodott alkotóelem esetében is a rendszer működőképes maradhat. Az elektronikai (informatikai) rendszerek megbízhatósági vizsgálatainál széleskörű alkalmazási lehetőséget nyújt a véges állapotú, folytonos idejű MARKOV- folyamatok elméletére támaszkodó modellezés; megkötést tulajdonképpen csak az exponenciális eloszlás feltételezése jelent. Ez a feltételezés azonban a komplex villamos rendszerekben használt eszközök jelentős részénél nem jelent különösebb megszorítást.

Egy *X(t)* sztochasztikus folyamat folytonos idejű MARKOV-lánc, ha minden egész *n* szám esetén tetszőleges olyan $t_1 \ldots t_{n+1}$ sorozatra, amelyre $t_1 < \ldots < t_{n+1}$, teljesül, és eleget tesz az alábbi összefüggésnek:

$$P\left[X(t_{n+1}) = j \middle| X(t_1) = i_1, X(t_2) = i_2, \ldots X(t_n) = i_n\right] = P\left[X(t_{n+1}) = j \middle| X(t_n) = i_n\right] \quad (9.1)$$

A (9.1) kifejezése tükrözi a MARKOV-folyamatoknak azt az alapvető tulajdonságát, hogy a múlt befolyását a folyamat jövőjére a pillanatnyi állapot tartalmazza. A folyamat „emlékezet nélküli". A folytonos idejű MARKOV-lánc esetében bármely időpillanatban bekövetkezhet állapotváltozás. A rendszer vizsgálatához szükségünk van annak megítéléséhez, hogy a folyamat mennyi ideig van egyik állapotában, mielőtt átlépne egy másik állapotba. Esetünkben ennek az időtartamnak exponenciális eloszlásúnak kell lennie (*markov-tulajdonság).*

A MARKOV-láncok állapotait a

$$\frac{dP_i(t)}{dt} = a_{ii}(t) \cdot P_i(t) + \sum a_{ij}(t) \cdot P_j(t) \qquad (9.2)$$

közönséges differenciál-egyenlet írja le, ahol $P_i(t)$ annak a valószínűsége, hogy a folyamat a t időpontban az i állapotban van, hasonlóan értelmezendő a P_j is.

Az $a_{ij}(t)$ intenzitást a következő összefüggés definiálja:

$$a_{ij} = \frac{p_{ij}(t, t+dt)}{dt} \qquad (9.3)$$

A $p_{ij}(t_n, t_{n+1})$ átlépési valószínűséget a

$$p_{ij}(t_n, t_{n+1}) = P\left[X(t_{n+1}) = j \middle| X(t_n) = i\right] \qquad (9.4)$$

képlet adja meg. Az $a_{ii}(t)$ kifejezhető $a_{ij}(t)$ segítségével:

$$a_{ii}(t) = \sum_j a_{ij}(t) \qquad (9.5)$$

Ha a $P_{ij}(t_n, t_{n+1})$ nem függ a t_n időponttól, a MARKOV-lánc ***homogén***. Amennyiben egy MARKOV-lánc homogén, véges számú állapota van és az átlépési valószínűségek pozitívak, a MARKOV-lánc ***ergodikus*** (azaz az ilyen rendszer egy adott időben bejárja az állapottér összes lehetséges állapotát.).

9.2 MARKOV-féle megbízhatósági modell

Az informatikai rendszerek megbízhatósági vizsgálatainál széleskörű alkalmazási lehetőséget nyújt a véges állapotú, folytonos idejű MARKOV-folyamatok elméletére támaszkodó modellezés. Megkötést tulajdonképpen csak az exponenciális eloszlás feltétele jelent, ez azonban az informatikai rendszerek jelentős részénél teljesül. A modell jellemzői:

- A rendszer egy sztochasztikus folyamat szerint különböző állapotokba kerülhet.
- A modellezés során az egyes állapotok közötti átmenetek lehetőségéből indulunk ki.
- A rendszer állapotait a t=0 időponttól vizsgáljuk.

A modell nagy előnye, hogy lehetővé teszi javítható rendszerek számítását is. Természetesen a MARKOV-féle megbízhatósági modell alkalmazásánál is figyelembe kell venni a vizsgált technikai rendszer speciális tulajdonságait is. Mindenképpen célszerű megvizsgálni, hogy a technikai rendszernek nincs-e valamilyen tanulási algoritmusa, önszerveződő képessége.

A MARKOV-folyamatokkal modellezett technikai rendszerek megbízhatósági vizsgálatainál elterjedt a gráfelméleti módszerek alkalmazása. A jelfolyam-gráfokat különösen eredményesen lehet használni az MTTF kiszámításához. Ezért a következő pontban összefoglaljuk a gyakorlati alkalmazásokhoz szükséges jelfolyam-gráfokkal kapcsolatos ismereteket.

9.2.1 Jelfolyam-gráfok

A jelfolyam-gráf olyan eszköz, amellyel ábrázolhatók egy lineáris algebrai egyenletrendszer változói között fennálló ok és okozati kapcsolatok. A kapcsolatok feltárása nagymértékben elősegítheti egy adott rendszer tulajdonságainak megismerését és esetenként a szükséges matematikai műveletek elvégzését.

A jelfolyam-gráfok élekből és csúcsokból állnak. A csúcs itt bármilyen változót jelenthet, de összegző és átvivő tulajdonsága is van. Az élhez átviteli tényezőket, vagy függvényeket rendelünk. A jelfolyam-gráfokkal kapcsolatos további megnevezések és definíciók:

- *Bemenő csúcsoknak* (forrásnak) csak kifutó ágai vannak (pl. a 9.1 ábra x_1 csúcsa).
- *Kimenő csúcs* (nyelő) olyan csúcs, amelynek csak befutó ágai vannak (pl. az x_5 csúcs). Ha olyan csúcsot kell kimenő csúcsnak tekinteni, amelyre nem teljesül a fenti feltétel egy egységnyi átviteli tényezőjű éllel

csatlakozunk a kérdéses csúcshoz, és pótlólagos változót (kimenő csúcsot) vezetünk be.

- Az *út* egymáshoz folyamatosan csatlakozó azonos irányban bejárt élek bármely sorozata.
- Az *előrevezető út*, olyan út, amely egy bemenő csúcstól egy kimenő csúcsig vezet úgy, hogy minden érintett csúcson csak egyszer halad át.
- A *hurok* olyan út, amely ugyanabból a csúcsból indul, ahová befut, és az érintett csúcsokon csak egyszer halad át.
- Az *út átviteli tényezője* az út befutása során érintett élek átviteli tényezőinek szorzata.
- Az *előrevezető út átviteli tényezője* az előrevezető út befutása során érintett élek átviteli tényezőinek szorzata.

A *huroktényező* a hurkot alkotó út átviteli tényezője.

Egy adott egyenletrendszer, amely $j=1,2\ldots N$ egyenletből áll ok és okozati formában az

$$X_j = \sum_{k=1}^{N} a_{kj} x_k \qquad (9.6)$$

alakban írható fel. Abban az esetben, ha egy lineáris rendszert integro-differenciálegyenletek írják le, a jelfolyam-gráf alkalmazásához először LAPLACE-transzformációt kell végrehajtani. A gyakorlati vizsgálatoknál a (9.6) összefüggést sokszor célszerű az alábbiak szerint értelmezni.

kimenőjel = (átviteli tényező) · (bemenőjel)

Egy adott egyenletrendszer jelfolyam-gráfjának megszerkesztése lényegében az ok és okozati összefüggések végigkövetését jelenti: mindegyik változót önmagával és más változókkal kifejezve kapcsolatban hozunk.

Példaképpen rajzoljuk fel az alábbi egyenletrendszer jelfolyam-gráfját, független változók: x_1 és x_2.

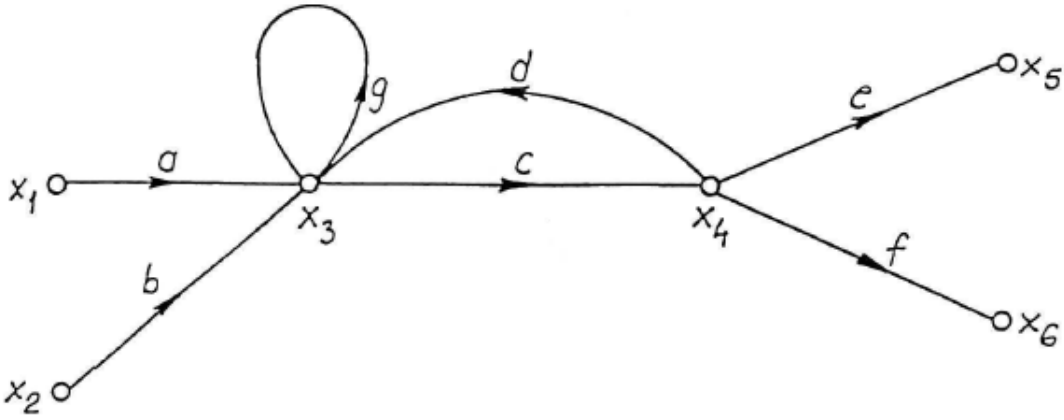
$$x_3 = ax_1+bx_2+dx_4+gx_3$$

$$x_4 = cx_3$$

$$x_5 = ex_4$$

$$x_6 = fx_4$$

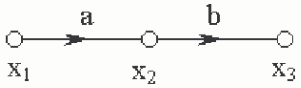
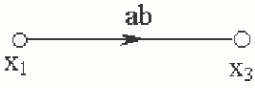
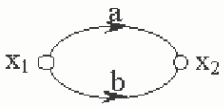
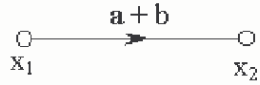
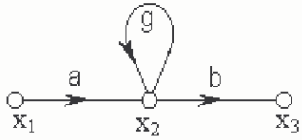
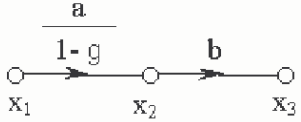
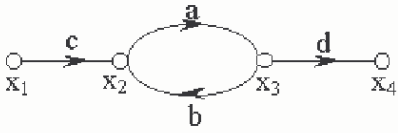
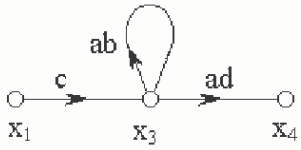
A jelfolyam-gráf a 9.1 ábrán látható.



9.1. ábra. Egy jelfolyam-gráf

Az ok és okozati kapcsolatok tisztázása után rendszerint felmerül a kérdés, milyen összefüggés áll fenn két kiválasztott lényeges változó (jel) között. Ezt a feladatot a gyakorlatban legtöbbször a jelfolyam-gráf egyszerűsítésével oldhatjuk meg. Bonyolult esetekben számítógépes módszereket kell alkalmazni.

A jelfolyam-gráfok egyszerűsítésének legfontosabb szabályai a 9.2 ábrában találhatók meg.

Megnevezés	Eredeti alakzat	Átalakított alakzat
Közbenső csúcs kiküszöbölése		
Párhuzamos élek összevonása		
Saját hurok kiküszöbölése		
Csúcs kiküszöbölése		

Egyszerűsítési szabályok

9.2.2 Jelfolyam-gráfok alkalmazása

A jelfolyam-gráfokat leggyakrabban az MTTF kiszámítására alkalmazzuk. Az alábbiakban e a fontos mutató számításának menetét mutatjuk be. Először felrajzoljuk a jelfolyam-gráfot. A jelfolyam-gráf felrajzolása az alábbiak szerint történik:

a. Az N különböző állapotnak N különböző csúcs felel meg.

b. A csúcsokat összekötő éleket az állapotokból történő átlépéseknek megfelelően a szükséges a_{ij} intenzitás $(i \neq j)$ LAPLACE-transzformáltjával súlyozzuk. Ha $i > j$, az intenzitás λ. Ha $j > i$, az intenzitás μ.

c. A csúcsoknál hurkot képezünk, melyet az elmenő élek intenzitás összegének LAPLACE-transzformáltjával súlyozunk és azt (-1)-el szorozzuk.

d. A kezdeti feltételeket a kiindulási állapothoz tartozó $1/s$ súlyozású éllel kell figyelembe venni.

A felrajzolt jelfolyam-gráfot ezután, átalakítjuk úgy, hogy a hurkokat és a csúcsokat eltüntetjük, a hibás állapothoz tartozó k csúcs kivételével, az így kapott $Z_k(s)$ függvény a hibás állapotban való tartózkodás valószínűségének LAPLACE- transzformáltja. A $Z_k(s)$ kiszámításához a reális üzemi feltételek számára tervezett rendszerek esetében általában elégséges az egyszerűsítő módszerek alkalmazása.

A $Z_k(s)$ ismeretében kiszámíthatjuk a megbízhatósági függvény $R(s)$ LAPLACE- transzformáltját:

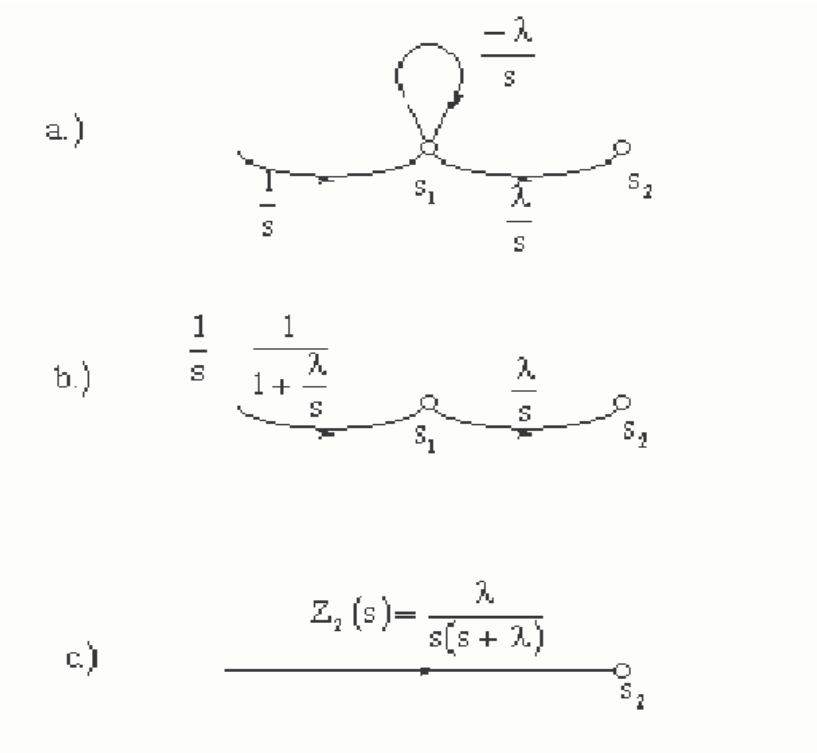
$$R(s) = \frac{1}{s} \cdot Z_k(s) \tag{9.7}$$

Kihasználva az improprius integrálok és a LAPLACE-transzformáció aszimptotikus tulajdonságait, felírható, hogy

$$MTTF = \lim_{s \to 0} \left(\frac{1}{s} - Z_k(s) \right) \tag{9.8}$$

Illusztratív példaként jelfolyam-gráf segítségével számítsuk ki egy nem javítható rendszer *MTTF* értékét! A meghibásodási ráta: λ.

Először a 9.3 ábrán látható módon meghatározzuk a $Z_2(s)$ függvényt.



9.3. ábra - Nem javítható elem jelfolyam-gráfja

(9.7) szerint a megbízhatósági függvény LAPLACE-transzformáltja:

$$R(s) = \frac{1}{s} - Z_2(s) = \frac{1}{s} - \frac{\lambda}{s(s+\lambda)} = \frac{1}{s+\lambda}$$

Így megkapjuk az előzőekből jól ismert eredményt:

$$MTTF = \lim_{s \to 0} \frac{1}{s+\lambda} = \frac{1}{\lambda}$$

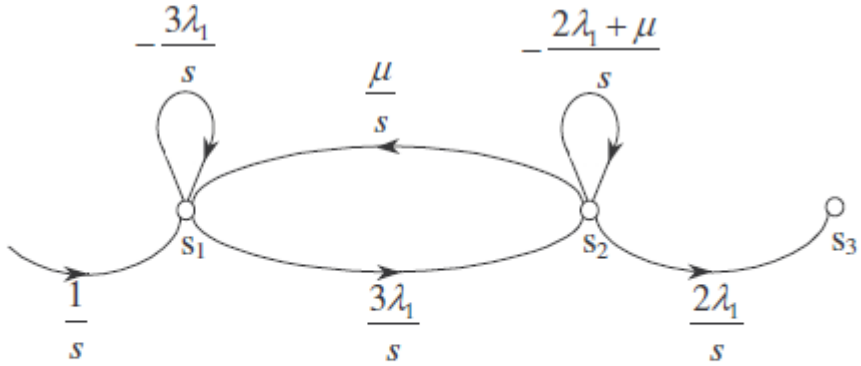
9.2.3 MARKOV-féle modell alkalmazása

A MARKOV-féle modell alkalmazásának illusztrálására számítsuk ki egy olyan rendszerre az *MTTF* értékét, amely három alrendszerből áll. A rendszert

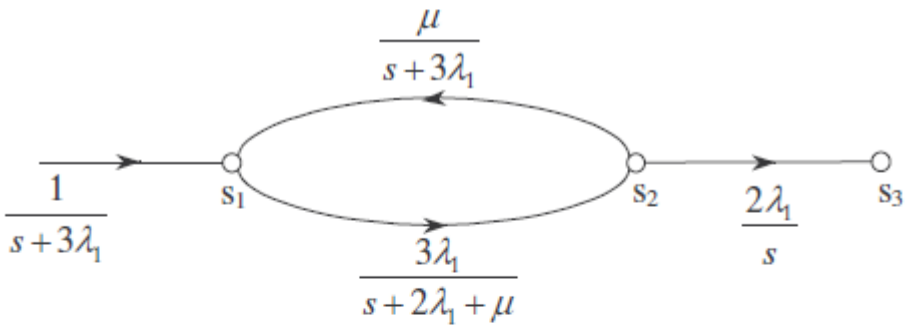
akkor tekintjük jónak, ha legalább két alrendszer üzemel. Amennyiben valamelyik alrendszer meghibásodik, akkor kezdődik el a javítás. A javítás befejezésével a rendszer visszakerül előző állapotába, hacsak a javítás ideje alatt meg nem hibásodik a tartalék. A részrendszerek meghibásodási rátája □ 1, javítási rátája.

Először felrajzoljuk a jelfolyam-gráfot (9.4. a. ábra), majd elvégezzük a szükséges átalakításokat (9.4. b. és c. ábrák).

a)



b)



c)

$$Z_3(s)=\frac{6\lambda_1^2}{s\left[(s+2\lambda_1+\mu)(s+3\lambda_1)-3\lambda_1\mu\right]}$$

9.4. ábra - Az MTTF számításához szükséges jelfolyam-gráfok

A $Z_3(s)$ ismeretében:

$$R(s) = \frac{1}{s} - Z_3(s) = \frac{s + 5\lambda_1 + \mu}{s^2 + s(5\lambda_1 + \mu) + 6\lambda_1^2}$$

Így (9.8) alapján:

$$MTTFF = \lim_{s \to 0} R(s) = \frac{5\lambda_1 + \mu}{6\lambda_1^2}$$

9.3 Az állapotfüggő karbantartás kibernetikai alapjai

Tegyük fel, hogy van valamilyen műszaki berendezésünk, amelynek viselkedését az alábbi *rendszeregyenlet* írja le:

$$\Phi(\mathbf{x}, \mathbf{u}, t) = \mathbf{0} \qquad (9.9)$$

A kimeneti folyamatot pedig az alábbi kimeneti egyenlet:

$$\mathbf{v} = \Psi(\mathbf{x}, \mathbf{u}, t) \qquad (9.10)$$

($\mathbf{x} \in \Re n$, $\mathbf{u} \in \Re m$ és $\mathbf{v} \in \Re k$. $t \in \Re$ a rendszeridő. Φ és Ψ valós, vektorértékű operátorok az $n+m+1$ dimenziós időfüggvények terén.)

- **u** a rendszer bemenőjele, amely az irányító és a zavaró jeleket is tartalmazza.
- **v** a kimenőjel, vagyis a rendszer válasza.

- **w** **v** kiértékelése után új beavatkozó jel képződik, amely megpróbálja a rendszert „jobb” kimenőjel adására késztetni. A bemenőjel zajos is lehet, a rendszer sztochasztikusan is működhet.

Legyen a rendszer állapotvektora felbontható az alábbi formában:

$\mathbf{x} = (\mathbf{y}^T, \mathbf{w}^T)^T$, ahol $\mathbf{y} \in \Re$ jellemzi a műszaki állapotot vagy *kondíciót*,

$\mathbf{w} \in \Re_{n-q}$, pedig a működési állapotot vagy *státuszt*.

Legyen adva $\Re_q$ diszjunkt részhalmazainak egy osztálya, amelyeket $Y_1, Y_2, \ldots, Y_r$ jelöl. Ezen részhalmazok mindegyike megfelel a rendszer valamilyen *minőségi osztályának* (előírásos állapot, degradált állapot stb.). Mindegyik minőségi osztálynak lehet saját rendszeregyenlete:

$$\mathbf{\Phi}_i(\mathbf{x}, \mathbf{u}) = \mathbf{0} \qquad i = 1,2,\ldots,r \qquad (9.11)$$

és kimeneti egyenlete:

$$\mathbf{v} = \mathbf{\Psi}_i(\mathbf{x}, \mathbf{u}) \qquad i = 1,2,\ldots,r \qquad (9.12)$$

ahol $\mathbf{\Phi}_i$ és $\mathbf{\Psi}_i$ valós, vektorértékű operátorok a valós $n+m$ dimenziós időfüggvények terén. (9.11) és (9.12) abban különbözik (9.10) és (9.11)-től, hogy eltűnt az explicit időfüggés.

A minőségi osztályok közti **átmenet**nek két formája lehetséges:

kényszerátmenet	**véletlen átmenet**
(pl. javítás)	(pl. meghibásodás)

Legyen $\mathbf{x}(t) = [\mathbf{y}(t)^T, \mathbf{w}(t)^T]^T$ a rendszerállapotot leíró időfüggvény, és definiáljuk az alábbi eseményt:

$$A_i(t) = \{\mathbf{y}_i(t) \in Y_i\} \qquad (9.13)$$

Bevezetjük a λ_i i=1, 2,…, r és y_{ij}, i,j=1, 2,…, r nem negatív számokat a következő átmenetmodellhez:

$$P\left[A_i(t+\Delta t)\middle|A_i(t)\right]=1-\lambda_i\Delta t+o(\Delta t) \qquad (9.14)$$

$$P\left[A_j(t+\Delta t)\middle|A_i(t)\right]=\lambda_i y_{ij}\Delta t+o(\Delta t) \qquad (9.15)$$

Ha egy minőségi osztályt a rendszer képtelen elhagyni, akkor a λ_i állapot-elhagyási ráta értéke nullával egyenlő.

Megjegyzés: a nagy, közepes és kicsiny megbízhatóság megfelel a kicsiny, közepes és nagy λ_i>0 értékeknek.

A rendszert *osztályozható*nak nevezzük, ha **v** trajektóriájából (pályavonalából) meghatározható az aktuális minőségi osztály valamilyen osztályozási algoritmus segítségével egy olyan valószínűségi szinten, amely valamilyen értelemben kielégítőnek tekinthető. Legyen az osztályozás eredménye a *j* osztályindex. Ha elégedettek vagyunk, továbblépünk, de ha nem, akkor kényszerítjük a rendszert, hogy osztályt változtasson (pl. javítás útján). Az ***állapotfüggő karbantartás*** éppen ezt a funkciót látja el.

Ezen a ponton elvégezhetjük a rendszer „finomhangolását" **u** értékének megfelelő módosításával. Ezután ismét megfigyeljük **v**-t.

Minden osztályváltási, és finomhangolási tevékenységnek, a matematikai alakfelismeréssel együtt, van valamekkora ***költség***e, míg a kimeneti vektor kívánatos értékei ***fedezet***et termelnek, a nemkívánatosak pedig ismét csak költséget (vagy elmaradó hasznot).

(Költségfüggvény és célfüggvény-konstrukciót illetően lásd a szakirodalmat.)

Kibernetikai értelemben egy szabályozott ***MARKOV-folyamat***ot kell optimalizálnunk, vagyis ***markovi döntésfolyamat***ot kapunk.

Általában nem közvetlenül a kimeneti vektor, hanem több megfigyelésből származó adat alapján osztályozunk. Ezért kell ***dimenziócsökkentés***t alkalmazni. Ez a ***lényegkiemelés*** statisztikai vagy műszaki jellegű.

9.4 Kidolgozott példák

1. Folyamatos gyártóvonalon készülő terméket automatikusan ellenőriznek. Az ellenőrző műszer meghibásodása esetén azonnal átkapcsolnak egy tartalék műszerre, amely állandó készenlétben van (hideg tartalék). A rendszer megbízhatóságának jobb modellezése érdekében feltételezünk még az átkapcsolásnál bizonyos mértékű hibalehetőséget, és azt, hogy az első műszert meghibásodásakor – a hiba helyének felderítése után – azonnal javítani kezdik, így az egy idő múlva ismét üzembe helyezhető. A rendszer eredő megbízhatóságának megállapításához az alábbi állapotok definiálhatók:

S_1: Az egyik műszer **működik**, a másik tartalékban van.

S_2: Az egyik műszer meghibásodott, átkapcsoltunk a tartalék egységre. Megkezdődik a hiba keresése. A rendszer ***működik***.

S_3: A hiba lokalizálása után megkezdődött a javítás. A tartalék egység végzi a mérést. A rendszer ***működik***.

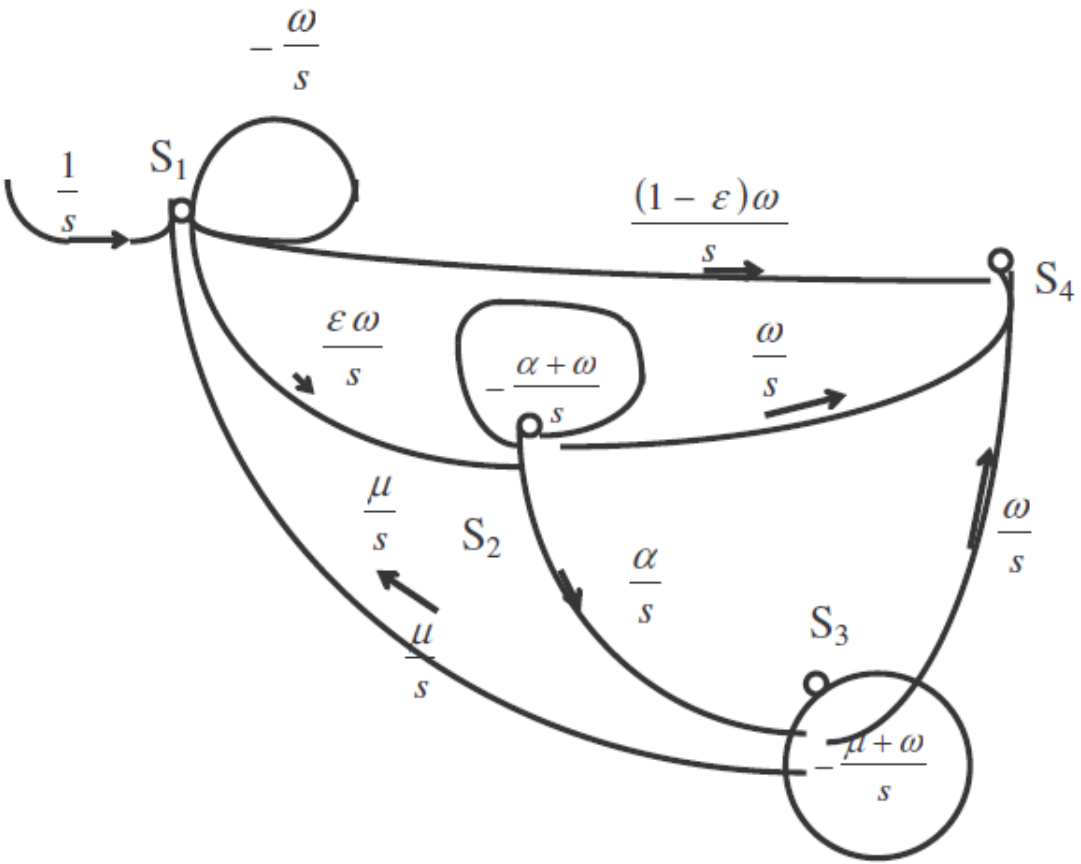
S_4: Hibás átkapcsolás következtében nem indult meg, illetve a tartalék egység meghibásodása miatt megszűnt a mérés, még mielőtt az eredetileg üzemelő műszert helyreállították volna. A rendszer ***nem működik***.

Legyen minden műszerre ω a két meghibásodás közt eltelt átlagos idő reciproka (azonos típusú műszerek), ε az átkapcsolási művelet sikerességének valószínűsége,

α a hibakeresés várható időtartamának reciproka és μ a javításhoz átlagosan szükséges idő reciproka. Legyen pl.: ω=10^{-3}[h^{-1}], ε=0,995, α=1[h^{-1}] és μ=0,5[h^{-1}]!

a. Készítsük el a megbízhatósági modell jelfolyam-gráfját!
b. Határozzuk meg a rendszer első leállásának várható idejét!
c. Viszonyítsuk ezt egyetlen műszer *MTTF*-jéhez! A válasz részletes kidolgozása

A megbízhatósági modell jelfolyam-gráfja:



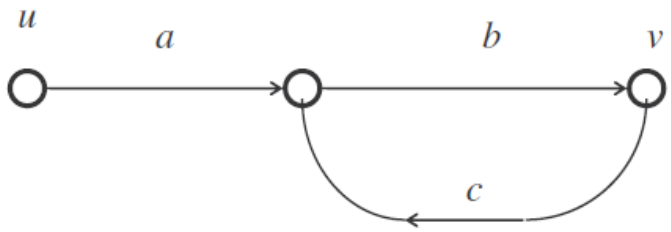
Az 1/s bemenő él azt jelzi, hogy a folyamat az S_1 állapotból indul.

b. Mielőtt a jelfolyam-gráf alapján meghatároznánk a működő állapotokban tartózkodás valószínűségének a LAPLACE-transzformáltját, érdemes egy kis elméleti kitérőt tenni. (9.8) alapján:

$$M(\xi) = \lim_{s\to 0} s\,\mathcal{L}\left\{\int_0^t R(x)\,dx\right\} = \lim_{s\to 0} R(s)$$

Tehát a várható élettartam a megbízhatósági függvény LAPLACE-transzformáltjának $s\to 0$ határértékével egyenlő. Ezt a fontos eredményt fogjuk fölhasználni a továbbiakban.

A működő állapotokat összekötő éleket μ/s-nél fölbontjuk. Ekkor kiszámítjuk a hurokátviteli függvényt, melynél figyelembe kell venni a működő állapotoknál lévő hurkokat is. A jelfolyam-gráfok algebrája szerint az alábbi összefüggést alkalmazhatjuk:



$$T(s) \triangleq \frac{V(s)}{U(s)} = \frac{ab}{1-bc}$$

Tehát a hurokátviteli függvény:

$$Y(s) = \frac{1}{1+\frac{\omega}{s}} \cdot \frac{\varepsilon\omega}{s} \cdot \frac{\frac{\alpha}{s}}{1+\frac{\alpha+\omega}{s}} \cdot \frac{\frac{\mu}{s}}{1+\frac{\mu+\omega}{s}} = \frac{\alpha \quad \varepsilon \quad \mu \quad \omega}{(s+\omega)(s+\alpha+\omega)(s+\mu+\omega)}$$

Jelöljük az egyes állapotokbantartózkodás valószínűségének LAPLACE- transzformáltját rendre $X_i(s)$-sel. Tudjuk, hogy

$$R(s) = \sum_{i=1}^{3} X_i(s)$$

Most pedig meghatározzuk a jelfolyam-gráfok algebrájának ismételt alkalmazásával a működő állapotokban tartózkodás valószínűségeinek LAPLACE-transzformáltját:

$$X_1(s) = \frac{\frac{1}{1+\frac{\omega}{s}} \cdot \frac{1}{s}}{1-Y(s)} = \frac{\frac{1}{s+\omega}}{1-Y(s)};$$

$$X_2(s) = \frac{\varepsilon\omega}{s} \cdot \frac{1}{1+\frac{\alpha+\omega}{s}} \cdot X_1(s) \text{ és}$$

$$X_3(s) = \frac{\frac{\alpha}{s}}{1+\frac{\mu+\omega}{s}} \cdot X_2(s) = \frac{\alpha}{s+\mu+\omega} \cdot X_2(s).$$

Most már csak az összegzés és a határértékképzés van hátra:

$$R(s) = \frac{\frac{1}{s+\omega}\left[1+\frac{\varepsilon\omega}{s+\alpha+\omega}\left(1+\frac{\alpha}{s+\mu+\omega}\right)\right]}{1-Y(s)} =$$

$$= \frac{(s+\alpha+\omega)(s+\mu+\omega)\left[1+\frac{\varepsilon\omega}{s+\alpha+\omega}\left(1+\frac{\alpha}{s+\mu+\omega}\right)\right]}{(s+\omega)(s+\alpha+\omega)(s+\mu+\omega)-\alpha\varepsilon\mu\omega} =$$

$$= \frac{[s+\alpha+\omega(1+\varepsilon)](s+\mu+\omega)+\alpha\varepsilon\omega}{(s+\omega)(s+\alpha+\omega)(s+\mu+\omega)-\alpha\varepsilon\mu\omega}$$

$$M(\xi) = \lim_{s\to 0} R(s) = \frac{[\alpha+\omega(1+\varepsilon)](\mu+\omega)+\alpha\varepsilon\omega}{\omega[(\alpha+\omega)(\mu+\omega)-\alpha\varepsilon\omega]} = \frac{1}{\omega} \cdot \frac{1{,}001995 \cdot 0{,}501 + 0{,}000995}{1{,}001 \cdot 0{,}501 - 0{,}5 \cdot 0{,}995} =$$

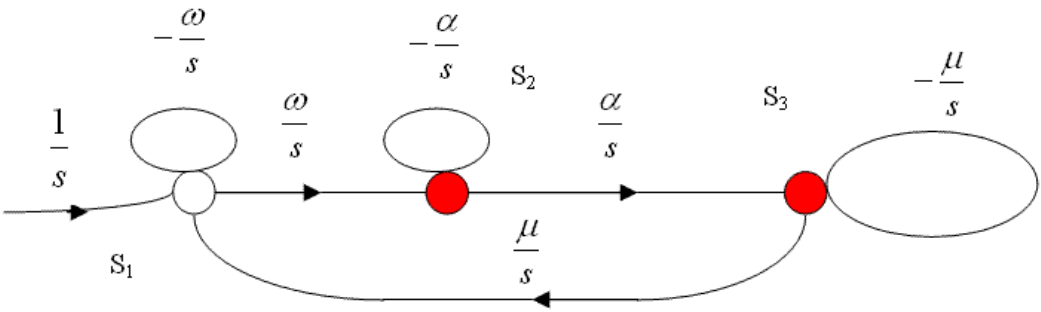
$$= \frac{1}{\omega} \cdot \frac{0{,}5029945}{0{,}004001} \cong \frac{125{,}717}{\omega} = 125717[\text{h}]$$

c. Tehát a hideg tartalékolt rendszer várható élettartama egy műszeréhez képest kb. 126-szorosára növekedett.

Adott egy felújítási Markov-folyamat jelfolyam-gráfja. A nem működő rendszer állapotait piros csomópontokkal jelöltük. A paramétereket a születési dátumából kell generálnia.

$$\omega = \frac{1}{1945}\left[\frac{1}{\text{h}}\right],\quad \alpha = \frac{1}{5}\left[\frac{1}{\text{h}}\right] \quad \text{és} \quad \mu = \frac{1}{9}\left[\frac{1}{\text{h}}\right]$$

Mekkora a készenléti tényező aszimptotikus értéke?



Ad1. állapottér-módszerrel: Az S_i állapotokban tartózkodás valószínűsége függ az időtől. Feltételezzük, hogy ezeknek az $x_i(t)$ függvényeknek létezik az $X_i(s)$ Laplace-transzformáltja. A jelfolyam-gráf alapján felírható egyenletek:

$$X_1(s) = \frac{1}{s} - \frac{\omega}{s}X_1(s) + \frac{\mu}{s}X_3(s)$$

$$X_2(s) = \frac{\omega}{s}X_1(s) - \frac{\alpha}{s}X_2(s)$$

$$X_3(s) = \frac{\alpha}{s}X_2(s) - \frac{\mu}{s}X_3(s)$$

Ha mindhárom egyenletet megszorozzuk az s operátorral és elvégezzük az inverz Laplace-transzformációt, akkor egy differenciálegyenlet-rendszert kapunk, amelynek két tetszőleges egyenletét kell megtartanunk, mert az állapotjelzők nem függetlenek, hiszen teljes eseményrendszert alkotva érvényes rájuk, hogy

$$\sum_{i=1}^{3} x_i(t) = 1$$

$$\dot{x}_1 = -\omega x_1 + \mu x_3 = -(\mu + \omega)x_1 - \mu x_2 + \mu$$

$$\dot{x}_2 = \omega x_1 - \alpha x_2$$

Állandósult állapotban a deriváltak értéke zérus, tehát egy lineáris algebrai egyenletrendszert kapunk, amelynek x_1 megoldása maga a keresett aszimptotikus készenléti tényező.

$$\begin{cases} (\mu + \omega)x_1 + \mu x_2 = \mu \\ \omega x_1 - \alpha x_2 = 0 \end{cases}$$

$$\begin{cases} \alpha(\mu + \omega)x_1 + \alpha\mu x_2 = \alpha\mu \\ \mu\omega x_1 - \alpha\mu x_2 = 0 \end{cases}$$

$$[\alpha(\mu + \omega) + \mu\omega]x_1 = \alpha\mu$$

$$K = x_1 = \frac{\alpha\mu}{\alpha(\mu + \omega) + \mu\omega} = \frac{\frac{1}{5} \cdot \frac{1}{9}}{\frac{1}{5}\left(\frac{1}{9} + \frac{1}{1945}\right) + \frac{1}{9} \cdot \frac{1}{1945}} = \frac{1945}{1945 + 9 + 5} =$$

$$= \frac{1945}{1959} = 0{,}992853496682 \approx 0{,}993$$

Ad2. Jelfolyam-gráf kapcsolási módszerrel:

A visszacsatolásnál felnyitott kör átviteli függvénye:

$$Y(s) = \frac{\omega}{s + \omega} \cdot \frac{\alpha}{s + \alpha} \cdot \frac{\mu}{s + \mu}$$

Az első állapotban tartózkodás valószínűségének Laplace-transzformáltja:

$$X_1(s)=\frac{\frac{s}{s+\omega}\cdot\frac{1}{s}}{1-Y(s)}=\frac{(s+\alpha)(s+\mu)}{(s+\alpha)(s+\mu)(s+\omega)-\alpha\mu\omega}=$$

$$=\frac{(s+\alpha)(s+\mu)}{s\left[s^2+(\alpha+\mu+\omega)s+\alpha(\mu+\omega)+\mu\omega\right]}$$

E valószínűségi időfüggvény végtelenben vett határértéke a Laplace- transzformáció végérték-tételével:

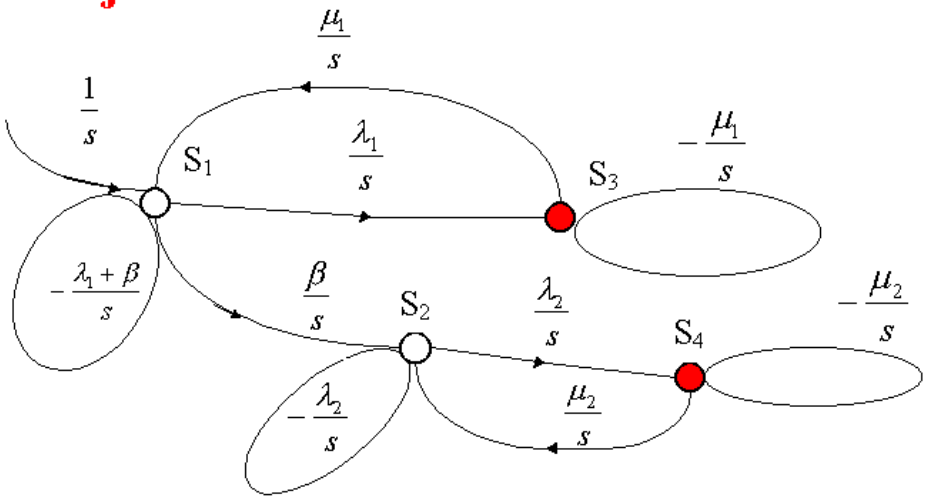
$$K=\lim_{t\to\infty}x_1(t)=\lim_{s\to0}sX_1(s)=\lim_{s\to0}\frac{(s+\alpha)(s+\mu)}{s^2+(\alpha+\mu+\omega)s+\alpha(\mu+\omega)+\mu\omega}=$$

$$=\frac{\alpha\mu}{\alpha(\mu+\omega)+\mu\omega}$$

3. Adott egy Markov-folyamat jelfolyam-gráfja. A nem működő rendszer állapotait piros csomópontokkal jelöltük. Mekkora a készenléti tényező aszimptotikus értéke, ha a paraméterek:

$$\beta=\frac{1}{20\text{h}},\quad \lambda_1=\frac{1}{50\text{h}},\quad \lambda_2=\frac{1}{1000\text{h}},\quad \mu_1=\frac{2}{\text{h}}\ \text{és}\ \mu_2=\frac{1}{4\text{h}}?$$

Bejáratás javítással



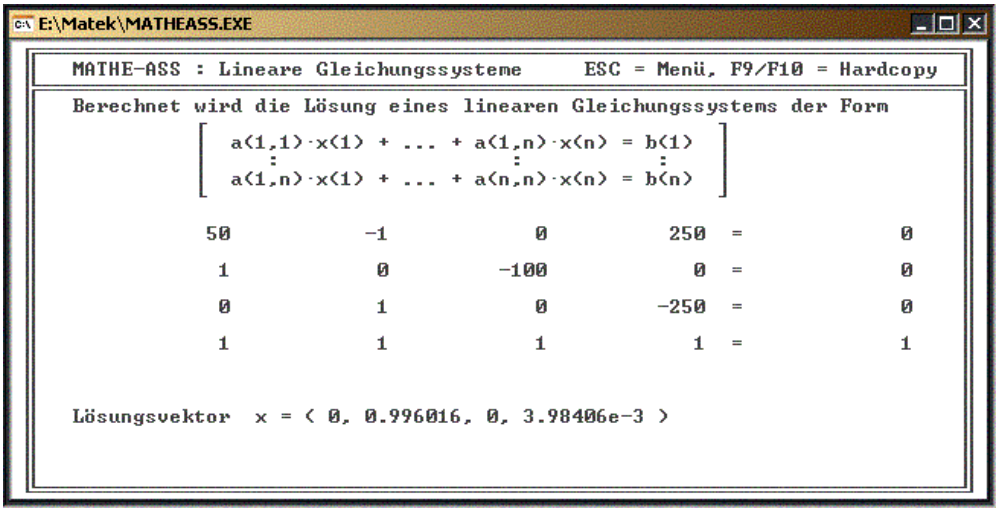
Kidolgozás:

$$\begin{cases} \dot{x}_1 = -(\beta+\lambda_1)x_1 + \mu_1 x_3 \\ \dot{x}_2 = \beta x_1 - \lambda_2 x_2 + \mu_2 x_4 \\ \dot{x}_3 = \lambda_1 x_1 - \mu_1 x_3 \\ \dot{x}_4 = \lambda_2 x_2 - \mu_2 x_4 \end{cases}$$

Állandósult állapotban a deriváltak értéke zérus:

$$\begin{cases} \beta x_1 - \lambda_2 x_2 + \mu_2 x_4 = 0 \\ \lambda_1 x_1 - \mu_1 x_3 = 0 \\ \lambda_2 x_2 - \mu_2 x_4 = 0 \\ x_1 + x_2 + x_3 + x_4 = 1 \end{cases}$$

Az utolsó egyenlet a teljes eseményrendszerre érvényes. Ezzel lecserélve, pl. az első homogén egyenletet, egyértelmű megoldást szolgáltató inhomogén, lineáris, algebrai egyenletrendszert kaptunk, amely a megadott paraméterekkel, de 1000-rel, illetve 100-zal beszorozva a megoldó program képernyőjén látható mátrixos formában:



A keresett aszimptotikus készenléti tényező a működő állapotokban, végül a bejáratott állapotban tartózkodás valószínűsége:

$$K = \lim_{t \to \infty} K(t) = \lim_{t \to \infty} x_2(t) \cong 0{,}996$$

A differenciálegyenletet akkor célszerű megoldani, ha valaki a $K(t)=x_1(t)+x_2(t)$ függvényre is kíváncsi. A második differenciálegyenlet helyére téve az algebrai egyenletet, a feladatot számítógéppel, konkrétan blokkorientált szimulációval megoldó TUTSIM modell:

```
MODEL:
   1.0000          1 INT      7      -8          ;x1
                   2 SUM      9      -1     -3   ;x2
                             -4
   0.0000          3 INT      6      -7          ;x3
   0.0000          4 INT      5     -10          ;x4
   0.0010000       5 GAI      2                  ;x2/1000
   0.0200000       6 GAI      1                  ;x1/50
   2.0000          7 GAI      3                  ;2x3
   0.0700000       8 GAI      1                  ;0.07x1
   1.0000          9 CON                         ;1
   0.2500000      10 GAI      4                  ;x4/4
                  11 SUM      1       2          ;K
```

A grafikonon felülről lefelé az $x_3(t)$, $x_4(t)$ és $K(t)$ függvény látható: A számszerű futtatás értéktáblázatának vége:

Az alsó (3-mal jelölt) grafikonon látszik, hogy $K(t)$ kezdetben az 1-ről lecsökken, majd felkúszik állandósult értékére, amint a bejáratás teljesen befejeződött.

A felső grafikon éppen azt mutatja, hogy a bejáratáskor fellépő, de gyorsan elhárítható üzemzavar miatti leállás valószínűsége gyorsan eléri maximumát, majd onnan lassan, de biztosan lecsökken egészen nullára.

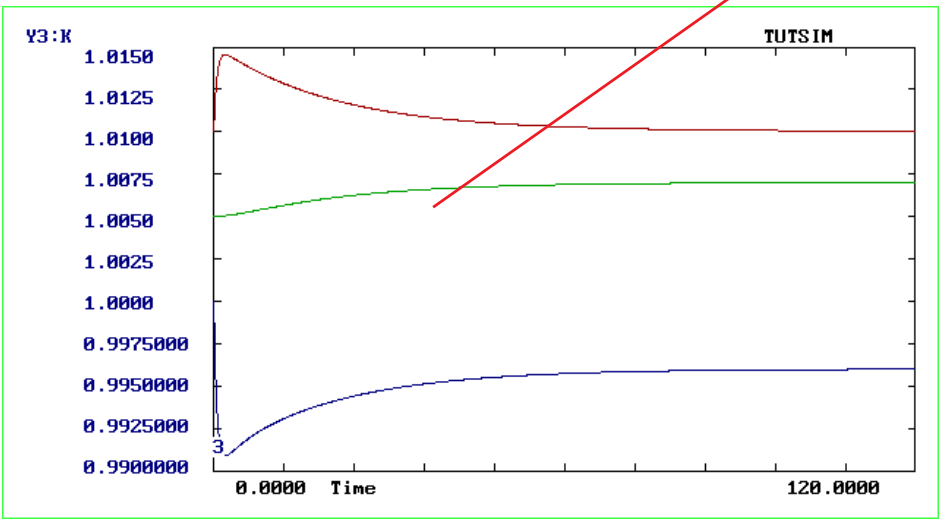
A bejáratott állapotban is vannak leállások, de ezek elfogadhatóan kis (kb. 0,004) valószínűséggel fordulnak elő. (Ez a középső grafikon.)

TUTSIM

Auto

294.7500	4.690E-09	0.0039840	0.9960160
295.0000	4.633E-09	0.0039840	0.9960160
295.2500	4.576E-09	0.0039840	0.9960160
295.5000	4.520E-09	0.0039840	0.9960160
295.7500	4.464E-09	0.0039840	0.9960160
296.0000	4.409E-09	0.0039840	0.9960160
296.2500	4.355E-09	0.0039840	0.9960160
296.5000	4.301E-09	0.0039840	0.9960160
296.7500	4.248E-09	0.0039840	0.9960160
297.0000	4.196E-09	0.0039840	0.9960160
297.2500	4.145E-09	0.0039840	0.9960160
297.5000	4.094E-09	0.0039840	0.9960160
297.7500	4.043E-09	0.0039840	0.9960160
298.0000	3.994E-09	0.0039840	0.9960160
298.2500	3.944E-09	0.0039840	0.9960160
298.5000	3.896E-09	0.0039840	0.9960160
298.7500	3.848E-09	0.0039840	0.9960160
299.0000	3.801E-09	0.0039840	0.9960160
299.2500	3.754E-09	0.0039840	0.9960160
299.5000	3.708E-09	0.0039840	0.9960160
299.7500	3.662E-09	0.0039840	0.9960160
300.0000	3.617E-09	0.0039840	0.9960160

COMMAND : K az utolsó

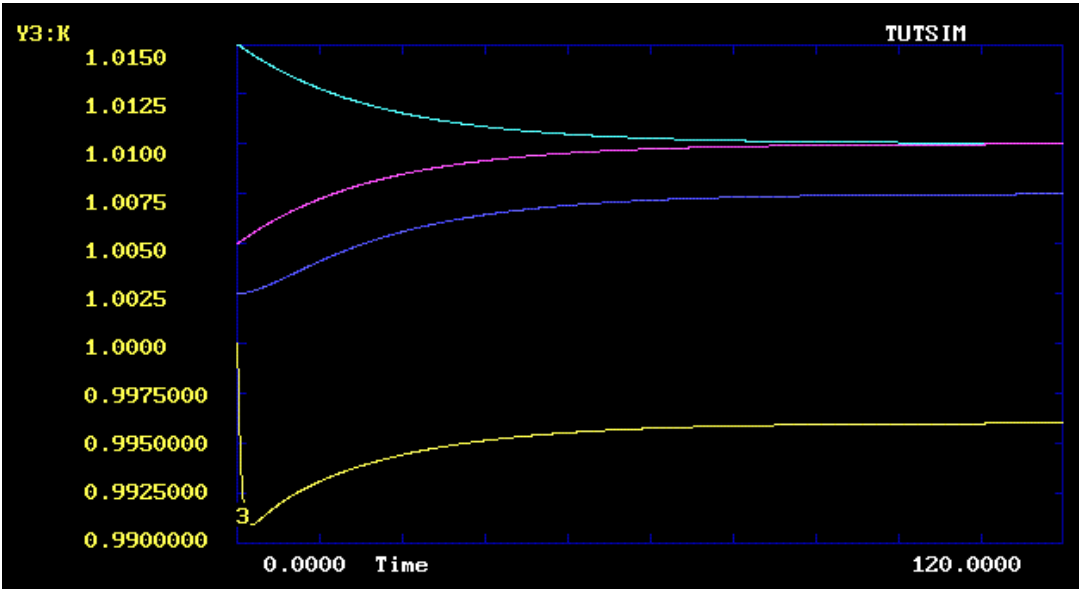


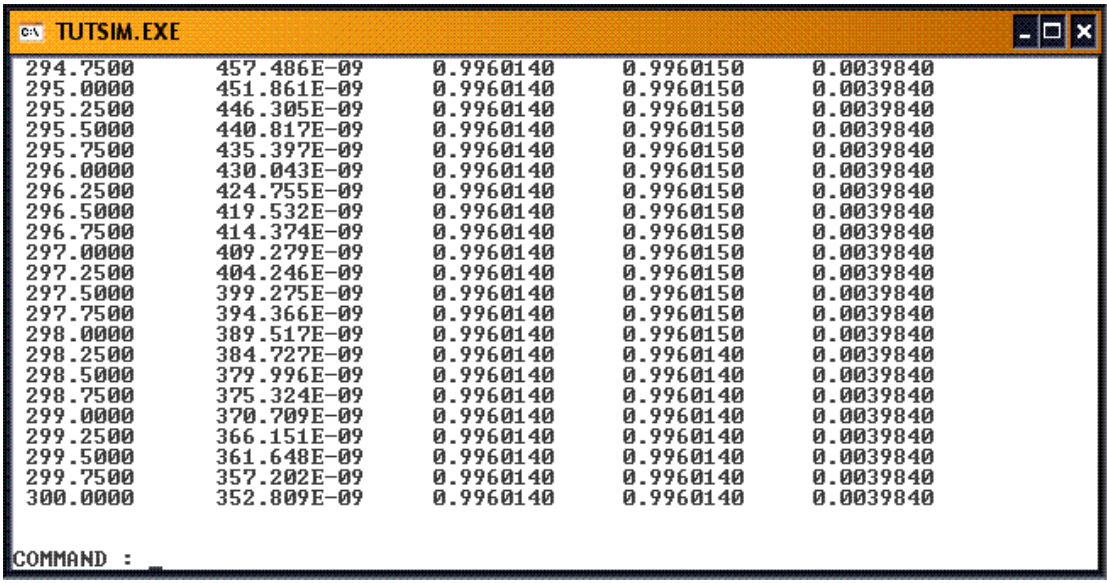
```
TUTSIM.EXE
Horz:      0 ,    0.0000      , 120.0000      ; Time
  Y1:      1 ,   -4.0000      ,   1.0000      ; x1
  Y2:      2 ,   -3.0000      ,   2.0000      ; x2
  Y3:     10 ,    0.9900000  ,   1.0150      ; K
  Y4:      4 ,   -0.0100000  ,   0.0100000   ; x4

This PERSONAL TUTSIM is not licensed for commercial use by corporations.
Use "EE" command for other license restrictions, including classroom use.


MODEL:
   1.0000          1 INT      -5       6      -7  ;x1
   0.0000          2 INT       7      -8       9  ;x2
   0.0000          3 INT       5      -6          ;x3
   0.0000          4 INT       8      -9          ;x4
   0.0200000       5 GAI       1                  ;lam1*x1
   2.0000          6 GAI       3                  ;mü1*x3
   0.0500000       7 GAI       1                  ;β*x1
   0.0010000       8 GAI       2                  ;lam2*x2
   0.2500000       9 GAI       4                  ;mü2*x4
                  10 SUM       1       2          ;K


COMMAND :
```





Ellenőrző kérdések

1. Mit nevezünk Markov-láncnak és mi a Markov-tulajdonság?
2. Melyek a Markov-féle megbízhatósági modell jellemzői?
3. Kibernetikai értelemben az állapotfüggő megbízhatóság-szabályozás modelljének kialakításakor milyen *döntésfolyamat*ot kapunk?

10. REKONFIGURÁLHATÓ SZÁMÍTÓGÉPEK (*RECONFIGURABLE COMPUTING*)

Angol nyelvű szakirodalom feldolgozását igénylő tananyagrész:

1. André DeHon. Comparing Computing Machines. In *Configurable Computing: Technology and Applications*, Proceedings of SPIE 3526, p. 124, November 1998.

http://brass.cs.berkeley.edu/documents/ccmpare_spie98.pdf

2. K. Compton, S. Hauck, "An Introduction to Reconfigurable Computing", *Northwestern University, Dept. of ECE Technical Report*, 1999.

http://www.ece.wisc.edu/~kati/Publications/Compton_ReconfigIntro.pdf

3. K. Compton, S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software", *ACM Computing Surveys*, Vol. 34, No. 2. pp. 171-210. June 2002.

http://www.ece.wisc.edu/~kati/Publications/Compton_ReconfigSurvey.pdf

4. R. Tessier and W. Burleson, Reconfigurable Computing and Digital Signal Processing: A Survey, *in Journal of VLSI Signal Processing*, May/June 2001, pp. 7- 27.

http://www.ecs.umass.edu/ece/tessier/jvsp00.pdf

5. R. Hartenstein (embedded tutorial): A Decade of Research on Reconfigurable Architectures – a Visionary Retrospective; Proc. International Conference on Design Automation and Testing in Europe 2001 (DATE 2001), Exhibit and Congress Center, Munich, Germany, March 2001.

http://xputers.informatik.uni-kl.de/papers/paper111.pdf

6. André DeHon, John Wawrzynek. Reconfigurable Computing: What, Why, and Design Automation Requirements? In *Proceedings of the 1999 Design Automation Conference*, pages 610--615, June 1999.

http://brass.cs.berkeley.edu/documents/rc_dac99.pdf

KÉRDÉSEK

10.1 Mi a rekonfigurálható számítógépek elvi gondolata, és mi nyújtja a megvalósításhoz a technológiai hátteret?

10.2 Milyen az FPGA felépítése és mi a LUT táblázatok szerepe?

10.3 Mutassa be, hogyan történik az utasításblokkok áthelyezése a hardverbe!

10.4 Mutasson példákat a rekonfigurálható processzorok alkalmazására!