Pannon Egyetem
Veszprém

Képfeldolgozás és Neuroszámítógépek Tanszék

Digitális Rendszerek és Számítógép Architektúrák

Segédlet szigorlati tételekhez

[Javított változat: 2006. november 28.]
<Bővítése folyamatosan>

Összeállította:

Vörösházi Zsolt
Dr. Szolgay Péter

1. Információ reprezentáció: (részletes leírás)

I.) NUMERIKUS INFORMÁCIÓ ÁBRÁZOLÁSA:

a.) Egész (integer) típusú számrendszer

A számítógépek számábrázolása és a legtöbb kommunikációs eljárás a "bit" fogalmán alapul. Egy bit olyan változó mennyiség, amely képes két különböző érték egyikét felvenni (vagy "0"-át vagy "1"-et). A bit más értelmezésben lehet: igaz v. hamis, megerősített v. nem megerősített stb. Bitek csoportja alkotja a számokat: minden egyes új bitpozíció *megduplázza* a számrendszerbeli lehetséges ábrázolhatóságot. Ily módon az ábrázolásnál rendelkezésre álló bitek száma határozza meg a reprezentálható értékek számát.(*N db bit esetén 2^N lehetséges érték van)*, de nem közvetlenül befolyásolja azok értékhatárát.

Az ábrázolásra adott feltételezések és követelmények határozzák meg a rendszer határait és fontosságát. A legegyszerűbb feltételezés, hogy bináris számokkal ábrázolják az **előjel nélküli egészeket (unsigned integer)**. Ekkor a reprezentálható értékek határa 0-tól 2^N-1 -ig terjedhet, és minden egyes szám között 1 egésznyi a különbség. Ez helyiértékes rendszer. Minden egyes k bitpozíció 2^k lehetséges értéknek felel meg. A bitcsoporttal ábrázolható érték a következő:

$$V_{\text{UNSIGNED INTEGER}} = \sum_{i=0}^{N-1} b_i \times 2^i$$

ahol b_i az i-edik pozícióban lévő 0 vagy 1. Ha például 101101 a bitmintázat, akkor ez $1x2^5 + 0x2^4 + 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 = 45_{10}$ jelent. Ez az előjel nélküli egész számábrázolás egyszerű és könnyen kezelhető, de a *negatív számok reprezentálása nem lehetséges*.

A legszélesebb körben használt rendszer a **kettes komplemens (two's complement) rendszer**. A 2^N reprezentálható érték határa $-(2^{N-1})$ től $2^{N-1}-1$ ig terjed. Ahhoz, hogy egy érték negáltját megkapjuk, ki kell vonnunk azt 2^N-ből. Ez az ábrázolási mód is helyiértékes, és egy adott megjelenítés értéke a következő módon adható meg:

$$V_{\text{2'S COMPLEMENT}} = -b_{N-1} \times 2^{N-1} + \sum_{i=0}^{N-2} b_i \times 2^i$$

Az 101101 bitmintázat 2-es komplemens esetén $-1x2^5 + 0x2^4 + 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 = -19_{10}$ jelent. Fontos megemlíteni: ha a legnagyobb helyiértékű bit (most significant bit=MSB) nincs is előjelbitként definiálva, alaphelyzetben mindig annak kell tekinteni. Tehát ha az előjelbit "1"-re van állítva, akkor a kapott érték negatív lesz, mivel ez a bit nagyobb súllyal számít, mint az összes többi. Ebből következik, hogy pozitív számoknál az első bit mindig „0". Milyen a -76_{10} (10-es alapú) bitmintázata egy 8 bites, 2-es komplemens rendszerben?

Először megkeressük a 76_{10} 8 bites bináris bitmintázatát, amely a 01001100. Hogy a negáltját megkapjuk, kivonjuk 2^8-ból:

$$\begin{array}{r} 100000000 \\ -\ 01001100 \\ \hline 10110100 \end{array} \qquad \begin{array}{r} 256 \\ -\ 76 \\ \hline 180 \end{array}$$

Tehát 10110100 decimális értéke megegyezik 180-al. De -76_{10} bináris bitmintázata a következő módszerrel is megadható: először felírjuk a 76 bináris bitmintázatát, majd bitenként negáljuk és hozzáadunk 1-et, végül a kapott értékeket összeadjuk.

$$\begin{array}{r} 01001100 \\ 10110011 \\ +\qquad 1 \\ \hline 10110100 \end{array}$$

Az egyik megvalósítása a 2-es komplemens rendszernek a **körkörös állapot számláló (circular nature).** Egy 4 bites 2-es komplemens rendszer, amelynél a számok a kör mentén sorrendben helyezkednek el 1000-tól 0111-ig (–8 tól 7 ig). Egyik számról a következőre léphetünk úgy, hogy egyszerűen csak növeljük, vagy csökkentjük eggyel az értéket. Azonban a 0111 és 1000 határán, a szám pozitívból negatívba vált át, aminek következtében szakadás lesz a kívánt számsorozatban. Ezt nevezzük túlcsordulásnak (overflow): a művelet meghaladja a számrendszer ábrázoló képességét. Ugyanez történik, ha először 1001-ről 1000-ra dekrementáljuk az értéket, majd 1000-ról 0111-re lépünk: ismét csak szakadás (alulcsordulás – underflow) következik be. Ha ezek a hiba egy aritmetikai műveletvégzés (ALU) során lépnek fel, akkor a számítógép egy túlcsordulást / alulcsordulást jelző bittel válaszol.

b.) Fixpontos számrendszer:

Általánosan elfogadott az a tény, hogy a tizedespont közvetlenül a legkisebb helyiértékű számjegytől jobbra helyezkedik el. Ezzel a megkötéssel valójában lehetséges egész számokat is ábrázolni. Azonban, hogyha a tizedesvessző a legkisebb helyiértékű bittől néhány pozícióval távolabb helyezkedik el, akkor a reprezentálható értékhatárok megváltoznak.

A leglényegesebb szempont a tizedespont elhelyezkedése: nincs olyan hardver, amely meg tudná határozni egy tizedesvessző pontos helyét. Ha a reprezentálni kívánt információ tartalmaz tört értékeket, akkor a tizedespont helyét úgy kell előre definiálni egy *fixpontos rendszerben*, hogy az lefedje az ábrázolni kívánt tartományt. Összeadás és kivonás esetén ez a rendszer pontosan úgy viselkedik, mint egész típus esetén. Azonban szorzás és osztás elvégzése után meg kell bizonyosodni arról, hogy a tizedespont a helyén maradt. Például két 16 bites szám szorzata egy 32 bites szám lesz, a tizedespont helyzetétől eltekintve. Probléma azonban, ha a két 16 bites szám eredményét is 16 biten szeretnénk tárolni, akkor ez a határ szabja meg a szám méretét. Egy fixpontos rendszer esetében ki kell választani a javító biteket azért, hogy a szorzandó és szorzó tizedespontjának helyét illetően tett megkötések segítségével helyes eredményt kapjunk.

A fixpontos rendszer is helyiértékes rendszer. Az egyetlen különbség csupán, hogy a tizedespont helyének változása miatt egy új tényezőt kell bevezetni: ennek jele "*p", amely a tizedespont feltételezett helyét* mutatja, pontosabban, hogy a legkisebb helyiértékű bittől balra hány bitpozícióval helyezkedik el (p értéke egész típusú számrendszer esetén 0). A következő egyenlettel egy 2-es komplemens fixpontos szám értékét lehet megadni:

$$V_{\text{FIXED POINT}} = -b_{N-1} \times 2^{N-p-1} + \sum_{i=0}^{N-2} b_i \times 2^{i-p}$$

A legjellemzőbb érték, amely segít meghatározni egy számrendszer használhatóságát, a *differencia*, amelyet *Δr-el* jelölünk. (a differencia jelentése: két szomszédos szám közötti különbség, abszolút értékben.) Egész típusú rendszer esetén ennek értéke 1, fixpontos rendszer esetén 2^{-p} (finomság).

Egy másik széles körben használt módszer az **1-es komplemens rendszer**. Egy V értékkel rendelkező N bites rendszer matematikailag a következő képlettel határozható meg: 2^N-1-V. A képlet "2^N-1" része N db egyesnek felel meg, és ha a V-t ebből a csupa egyesből álló bitmintából kivonjuk „0" lesz ott ahol „1"-es volt, „1"-es lesz ott ahol „0" volt (mivel egy szám negatív alakját, bitjeinek kiegészítésével adjuk meg). Tehát ahhoz, hogy egy szám negáltját megkapjuk, elég csupán minden bitjét negálni, amely így nagyon gyors műveletet eredményez. A lefedett értékhatár ($2^{N-1}-1$) től -($2^{N-1}-1$) ig terjed, amely éppen csak 1-el tér el a 2-es komplemens rendszertől. Azonban az eddig megismert rendszerekhez képest, ez nem helyiértékes rendszer, mivel a bitek helyiértékbeli eltérése a szám előjelétől függ. Ebben a rendszerben kétféleképpen is lehet ábrázolni a zérust, de ezeket egy megfelelő tesztelő rutinnal ellenőrizni kell. Végezetül, az átvitelkezelés is eltér az eddig tapasztaltaktól, mivel ez egy "körbeforgó átvitelű rendszer" (end-around carry), amelyben a részeredményhez kell hozzáadni a végrehajtás eredményét, hogy megkapjuk a helyes eredményt.

További fontos, számábrázoláshoz használt kódolási technika az **Excess kódrendszer**. Az egyik leggyakoribb felhasználási módja az excess kódnak, a lebegőpontos számok kitevőinek tárolásánál mutatkozik. Legyen S, a reprezentálni kívánt érték (eredmény), amit tárolni és használni fogunk; V a szám valódi értéke, és E az excess. Ekkor a következő kapcsolat áll fenn közöttük:

$$S = V + E.$$

Ennél a műveletnél figyelni kell arra, hogy az eredmény a kívánt határon belül legyen, mivel ha két számot összeadunk, akkor a következő történik:

$$S1 + S2 = (V1 + E) + (V2 + E) = (V1 + V2) + 2 \times E$$

Ahhoz, hogy megkapjuk a pontos eredményt - amely a [(V1+V2) + E] - a számított eredményből ki kell vonnunk E-t.

c.) Lebegőpontos számrendszer (Floating Point Number System: FPNS)

Sok probléma megoldásához olyan rendszerre van szükség, amely nagyságrendekkel nagyobb vagy kisebb értékeket is meg tud jeleníteni, mint mondjuk az egy fixpontos rendszerben lehetséges. Tudományos jelöléseket használhatunk nagyon nagy (pl. az Avogadro-szám: 6.022×10^{23}), ill. nagyon kicsi számok (pl. a proton tömege 1.673×10^{-24} g) felírásához. Egy lebegőpontos szám meghatározásához 7 különböző tényező szükséges: *a számrendszer alapja, előjele és nagysága, a mantissza alapja, előjele és hosszúsága, ill. a kitevő alapja*. Az ilyen típusú számok matematikai jelölése a következő módon lehetséges:

$$(\text{előjel})\ \text{Mantissza} \times \text{Alap}^{\text{Kitevő}}$$

A fenti összefüggésben az "alap" a rendszer alapszámát jelöli. Ez decimális rendszer esetén 10 (0-9 ig). A számítógépek többsége Neumann óta 2-es számrendszerrel dolgozik. Ez az alapszám egy konstans érték, amelyet a rendszer definiálásakor határozunk meg, és amely közvetlenül befolyásolja a reprezentálható értékhatárokat. A későbbiekben jelöljük "r_b"-vel a rendszer alapszámát, amely a mantisszára vonatkozik. A mantisszát egy szám értékjegyeinek meghatározásához használjuk. Míg a gyakorlatban használhatunk több vagy kevesebb jegyű mantisszát is, addig a gépi kódolásnál a mantissza ábrázoláshoz használt számjegyek összege azonos típusú szám esetén egyforma (úgymint egyszeres vagy dupla pontosság). Egyik jellegzetessége a lebegőpontos rendszernek, hogy a mantisszát a jegyek száma reprezentálja, ezt "m"-el jelöljük. Ezért egy speciális lebegőpontos rendszer esetén mindenegyes mantissza *m* darab r_b alapú számjegyből áll. A mantissza értékét jelöljük V_M-el. Azonban, ha jobban meggondoljuk, ismernünk kell még a mantissza legnagyobb és legkisebb megengedett értékét is, amit jelöljünk $V_{M\,(MAX)}$ és $V_{M\,(MIN)}$ -el.

Egy lebegőpontos számmal kifejezett valós számsor értékhelyeit a kitevő (exponent) határozza meg. Ha az exponens nagy pozitív szám, akkor a lebegőpontos szám értéke is nagyon nagy lesz, ha pedig az exponens nagy negatív szám, akkor a lebegőpontos szám értéke nagyon kicsi lesz. Ha az exponens 0, akkor a lebegőpontos szám értéke éppen a mantissza értékével lesz egyenlő. A kitevő értékének meghatározásához három adat szükséges: előjele, alapja, és nagysága. A későbbiekben az exponens alapját "r_e"-vel jelöljük. Egy számrendszer alapjához hasonlóan, "r_e" megválasztása is a tervezés során történik, része a szám meghatározásának. Lehet $r_b=r_e=10$, de a legtöbb számítógépnél $r_e=2$.

Az exponens maximális értékét, jegyeinek száma adja meg (melyet "e"-vel jelölünk), amely a rendszer alapszámával együtt meghatározza az ábrázolható értékek tartományát. Láthatjuk, hogy az exponens *e* db r_e alapú számjegyből fog állni. Meg kell még határoznunk az exponens előjelét is (egyszerűen csak figyelni kell a mínusz jel meglétére, vagy hiányára). Lehetséges az, hogy a kitevőt is lebegőpontos formában tároljuk. Ekkor kellene egy előjelbit, amely az exponens előjelét adja meg, majd pedig egy "előjel-hossz" formátum (sign-magnitude), amelyben az exponenst tárolni tudjuk. Azonban ezt a legtöbb számítógép nem alkalmazza, hanem egy kiválasztott kódolási technika segítségével reprezentálja a pozitív és negatív értékeket, erre a feladatra a legjobb módszer az excess kódolás. Az exponens értékét jelöljük V_E–vel. Ahogy a mantisszánál, úgy itt is kell ismernünk a legnagyobb és legkisebb ábrázolható exponens értékét. Ezeket jelöljük $V_{E\,(MAX)}$ –al és $V_{E\,(MIN)}$ –el.

Magának a számnak az előjelét is ismernünk kell, amelyet a tudományos módszerekkel egyértelműen meghatározhatunk. Egy ilyen az "előjel-hossz" (sign-magnitude) technika, amelyet a lebegőpontos számok tárolásánál alkalmaznak.

Végezetül ismernünk kell a tizedespont helyét is, amelyet "p"-vel jelölünk, a legkisebb helyiértékű bitre vonatkozólag, és ennek segítségével tudjuk meghatározni a (előjel nélküli) mantissza értékét:

$$V_M = \sum_{i=0}^{N-1} d_i \times r_b^{i-p}$$

ahol a mantissza N db r_b számjegyből áll (d_{N-1}-től d_0–ig). Ezért a lebegőpontos szám értékét (V_{FPN} -t) a következő módon tudjuk megadni:

$$V_{FPN} = (-1)^{SIGN} V_M \times r_b^{V_E}$$

A mantissza tizedespontjának helye közvetlenül kapcsolódik az exponens értékéhez. Tekintsük a következő szám lebegőpontos értékét:

$$32{,}768_{10} = 3.2768 \times 10^4 = 32.768 \times 10^3 = 327.68 \times 10^2 = 3267.8 \times 10^1$$

Mindegyik megjelenítési módban helyes eredményt kapunk, csupán a matematikai jelölésben van eltérés, és a tizedespont helyét az exponens értéke határozza meg. Ha tehát a tizedespont helye egy adott számon belül jegyről jegyre változik, akkor mindenképpen szükséges p értékét úgy rögzíteni, hogy felhasználható legyen minden egyes későbbi számítás során. Továbbá a legtöbb rendszerben megkötéseket teszünk a tárolt információ minimalizálása és a műveletvégzés megkönnyítése érdekében.

Ezt az eljárást nevezzük **normalizálásnak**, mely segítségével meghatározhatók a megengedett mantissza értékek. A példánkban szereplő feltétel: legyen p=M, és balról a legszélső számjegye a mantisszának ne legyen 0. Így a mantissza egy törtszám, amelynek értéke $1/r_b$ és közelítőleg 1 között váltakozik. Speciálisan, a mantissza maximális értéke: $V_{M\ (MAX)} = 0.d_m d_m d_m \ldots$ egészen a mantissza hosszának végéig, ahol $d_m = r_b$, és ez a szám nagyon közel van az 1-hez ($1- r_b^{-N}$), míg a minimális mantissza értéke $V_{M\ (MIN)} = 0.100 \ \ldots \ = 1/r_b$. Egy normalizált lebegőpontos rendszerben ábrázolt szám tehát megadható a $V_{M\ (MIN)}$-től $V_{M\ (MAX)}$-ig terjedő legális mantissza és a rendelkezésre álló exponens kombinációjával. Ezekből az észrevételekből a következő kifejezések vezethetők le, a nullától különböző értékeket illetően:

- Ábrázolható maximális érték: $V_{FPN(MAX)} = V_{M(MAX)} \times r_b^{V_{E(MAX)}}$
- Ábrázolható minimális érték: $V_{FPN(MIN)} = V_{M(MIN)} \times r_b^{V_{E(MIN)}}$
- Legális mantisszák száma: $NLM_{FPN} = (r_b - 1) \times r_b^{m-1}$
- Legális exponensek száma: $NLE_{FPN} = V_{E(MAX)} + \left|V_{E(MIN)}\right| + 1_{ZERO}$
- Ábrázolható értékek száma: $NRV_{FPN} = NLM_{FPN} \times NLE_{FPN}$

Ezek az értékek segítenek abban, hogy megadhassuk a lebegőpontos rendszer jellemzőit, és meghatározhassuk használatát egy speciális alkalmazásban.

Példa: Normalizált lebegőpontos rendszer tulajdonságai: Legyen $r_b = 10$, $r_e = 10$, m = 3, e = 2, és mind az exponenst, mind pedig a számot tároljuk "előjel-hossz" (sign magnitude) formátumban. Mekkora a legnagyobb, ill. legkisebb ábrázolható törtszám? Mekkora a legnagyobb, ill. legkisebb ábrázolható exponens? Mekkora a legnagyobb ábrázolható szám? Mekkora az ábrázolható legkisebb nullától különböző pozitív szám? Mennyi nullától különböző számot tud ez a rendszer reprezentálni? Ezekből az adatokból a következő egyenletek írhatók fel:

$$V_{M(MAX)} = 0.999 = 1.000 - 10^{-3}$$
$$V_{M(MIN)} = 0.100$$
$$V_{E(MAX)} = (r_e \ \hat{} \ e - 1) = 99$$
$$V_{E(MIN)} = -(r_e \ \hat{} \ e - 1) = -99$$

Fontos tudni, hogy $V_{E\ (MIN)}$ azt a legnegatívabb exponens értéket jelenti, amellyel meghatározható a legkisebb ábrázolható szám, mivel a legkisebb exponens abszolút értékben a 0.

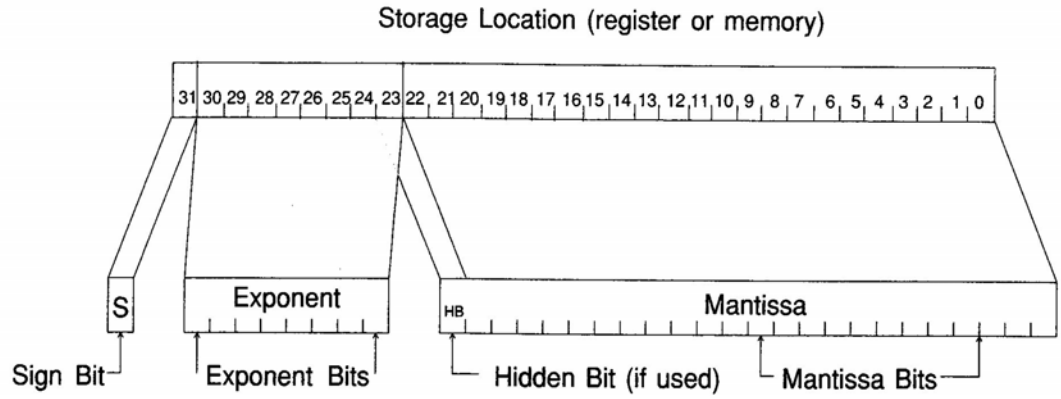
$$V_{FPN(MAX)} = 0.999 \times 10^{99}$$
$$V_{FPN(MIN)} = 0.100 \times 10^{-99}$$
$$NLM_{FPN} = 9 \times 10 \times 10 = 900 = 9 \times 10^2$$
$$NLE_{FPN} = 99 + \left|-99\right| + 1_{ZERO} = 199$$
$$NRV_{FPN} = 2 \times 900 \times 199 = 358{,}200$$

NLE kiszámításakor azért kapunk 199-et, mivel 99 nullánál nagyobb, 99 nullánál kisebb és egy, magával a nullával egyenlő exponenselem található ebben a rendszerben. További kérdésünk, hogy 1.00 és 10,000 összegét (10,001-et) fel tudjuk e írni ebben a számrendszerben? Mivel $1.00 = 0.100 \times 10^1$ és $10{,}000 = 0.100 \times 10^5$ ezért ez nem lehetséges. Ahhoz, hogy ábrázolni tudjuk 1.00-t, a differenciának 1/100-nak kellene lennie, 10,000 esetében pedig Δr=100. Mindez rámutat arra a tényre, hogy a Δr mindenegyes V_E esetén eltérhet.

A fenti két példa is alátámasztja, hogy a lebegőpontos rendszerek közötti főbb különbségek maghatározói a következő tényezők: *m, e, r_e, r_b*. Ha akár csak az egyikben is eltérnek, teljesen más megengedett értékeket kaphatunk. Ezért nagyon fontos, hogy a tervezés során helyesen megválasszuk ezek értékeit, a rendszer feltételezett igényeinek megfelelően.

A számítógépen történő számok tárolásának módszereivel sok hasonlóságot (és különbséget) mutat a **lebegőpontos számábrázolás**. A gépben tárolt információk a következők: előjel, exponens, és mantissza. Ezeket úgy csoportosítják, hogy az előjel legyen a legmagasabb helyiértékű bit, amit az exponens, végül a mantissza követ.

Azokat az adatokat, amelyek a számítások során sohasem változnak, nem tároljuk el. Ilyen például a rendszer alapszáma, és az exponens alapja. Szintén ilyen, bár nem ennyire nyilvánvaló a tizedespont helye, ill. az exponens tárolásának kódolási technikája. Ezeket mindig a rendszer tervezésekor kell megválasztani, ill. konstans értékként definiálni. Ilyen konstans információ lehet egy 2-es alapú normalizált rendszerben a számok elején szereplő 1-esek ("vezető" egyesek). Csak azokat az értékeket kell eltárolni, amelyek megváltozhatnak. Ilyen a *mantissza legmagasabb helyiértékű bitje*, amelyet rejtett (*hidden*) bitnek hívunk, és amely közvetlenül az exponensbitek mögött helyezkedik el. Ha ezt a rejtett bitet beállítjuk, akkor duplájára nő a legális mantisszák, és ezzel együtt az ábrázolható értékek száma. Másrészt a nullát nem tudjuk reprezentálni, mivel a legkisebb ábrázolható érték a 000000, ami a rejtett bit használata miatt $0.1000_2 \times 2^0 = ½$ -nek felel meg.



Felmerül a kérdés: mégis hogyan tudjuk ábrázolni a nullát? A következő példákban pontosan ezt vizsgáljuk meg közelebbről, de egy általános módszer, hogy az exponenst illetően különféle megkötéseket teszünk. A leggyakoribb technika, hogy excess 2^{e-1} kódolást alkalmazunk az exponens ábrázolásánál. Mivel a kód bináris reprezentációja ennek a kódnak 2^e-1 től 1-ig változik, ezért az exponens ennek megfelelően 2^{e-1} től -(2^{e-1} -1) ig terjedhet. Ha az exponens mindegyik bitje csupa 0 - a mantissza mezőben lévő bitek értékeitől függetlenül - akkor a feltételezett szám 0.

Példa: a 2-es alapú DEC lebegőpontos számrendszer tulajdonságai: a fenti összefüggések felhasználásával határozzuk meg a **DEC 32-bites, normalizált lebegőpontos rendszer** főbb jellegzetességeit. Legyen r_b=2, r_e=2, m=23+1=24 rejtett bittel együtt, p=24, e=8, az exponenst tároljuk Excess 128 kódolással, és a számokat tároljuk "előjel-hossz" formátumban (tekintsük a mantisszát pozitívnak). Így a fenti adatokból a következő egyenletek írhatók fel:

$$V_{M(MIN)} = 0.\underbrace{1000\ldots_2}_{24db} = 1/2$$

$$V_{M(MAX)} = 0.1111\ldots_2 = 0.999999940395 = 1.0 - 2^{-24}$$

$$V_{E(MIN)} = -(r_e^{e-1} - 1) = -(2^{8-1} - 1) = -127$$

$$V_{E(MAX)} = r_e^{e-1} - 1 = 2^{8-1} - 1 = 127$$

$$V_{FPN(MIN)} = 0.1000\ldots_2 \times 2^{-127} = 2.9387 \times 10^{-39}$$

$$V_{FPN(MAX)} = 0.1111\ldots_2 \times 2^{+127} = 1.7014 \times 10^{38}$$

$$NLM_{FPN} = 2^{23} = 8,388,608$$

$$NLE_{FPN} = 127 + |-127| + 1_{ZERO} = 255 = (r_e^{\,e} - 1) = 2^8 - 1$$

$$NRV_{FPN} = 2^{23} \times (2^8 - 1) = 2.139 \times 10^9$$

Ez a rendszer a rendelkezésre álló bitminták 99.6%-át használja fel a normalizált lebegőpontos értékek képzésénél. Azonban, ha a számok nagyon nagyok ill. nagyon kicsik, a rendszer nem képes biztosítani a műveletekhez szükséges dinamikus tartományt. Felmerül a kérdés: mennyi értékjegy áll rendelkezésre? Bármely V_E esetén $2^{23} \approx 8.4 \times 10^6$ különböző érték létezik, továbbá ebben a rendszerben egy decimális szám tudományos jelöléséhez 6 számjegyre van szükség (tehát az értékes jegyek száma 6), még akkor is, ha ezekkel a számjegyekkel a mérés során csak 3 tizedesjegy pontossággal ábrázolunk. A DEC rendszer tehát dupla pontosságú aritmetikát biztosít.

Egy dupla pontosságú számrendszer ugyanilyen módon működik, (azonos kezdeti feltételekkel, mint a DEC) azonban itt a mantissza bitek száma 24-től 56-ig terjedhet, aminek következtében 2^{32}-nel megnő az ábrázolható értékek száma (ennyivel növekedett a pontosság), de fontos, hogy az értékhatárok nem változnak meg. Így minden egyes V_E esetén 3.6×10^{16} érték jeleníthető meg, vagy kb. 16 helyiértékű számjegy.

Egy másik elterjedt rendszer az IBM lebegőpontos rendszer, amelyet a következő példában vizsgálunk meg.

Példa: 16-os alapú IBM lebegőpontos rendszer tulajdonságai: Határozzuk meg az **IBM-32 bites normalizált lebegőpontos rendszer** főbb tulajdonságait. Kezdeti feltételek: legyen r_b=16, r_e=2, m=6 hexadecimális számjegy, p=6, e=7, az exponenst tároljuk Excess 64 kódolással, és a számokat tároljuk "előjel-hossz" formátumban (tekintsük a mantisszát pozitívnak). Ekkor a következő egyenletek írhatók fel:

$$V_{M(MIN)} = 0.100000_{16} = 1/16$$

$$V_{M(MAX)} = 0.\text{FFFFFF}_{16} = 0.999999940395 = 1.0 - 16^{-6}$$

$$V_{E(MIN)} = -(r_e^{e-1} - 1) = -(2^{7-1} - 1) = -63$$

$$V_{E(MAX)} = r_e^{e-1} - 1 = 2^{7-1} - 1 = 63$$

$$V_{FPN(MIN)} = 0.100000 \times 16^{-63} = 8.636 \times 10^{-78}$$

$$V_{FPN(MAX)} = 0.\text{FFFFFF}_{16} \times 16^{+63} = 7.237 \times 10^{75}$$

$$NLM_{FPN} = 15 \times 16^5 = 15,728,640$$

$$NLE_{FPN} = 63 + |-63| + 1_{ZERO} = 127 = 2^7 - 1$$

$$NRV_{FPN} = 15 \times 16^5 \times (2^7 - 1) = 1.9975 \times 10^9$$

Ez a rendszer a DEC-nél sokkal nagyobb értékhatárokkal rendelkezik, bár ténylegesen 7%-al kevesebb értéket tud ábrázolni. Mégis, bizonyos alkalmazások esetén éppen erre van szükség. Az IBM rendszer is dupla pontosságú formátumot használ, ami a DEC-hez hasonlóan nem növeli meg a rendszer értékhatárait, de a valós számsorozatok ugyanakkora tartományán belül 16^8 -nal több értéket kezel.

Azonban az egyik legnagyobb hátránya egy normalizált lebegőpontos rendszernek, hogy nagy szakadás van a legkisebb ábrázolható érték és a nulla között, miközben a számok a nullához tartanak. A probléma, hogy a tizedesponttól jobbra lévő első számjegy csak nullától különböző lehet - így az első ábrázolható érték a nullától aránytalanul messze helyezkedik el. Ennek megoldására fejlesztették ki az IEEE lebegőpontos rendszert, amelyet a következő példában vizsgálunk meg.

Példa: Az IEEE lebegőpontos rendszer tulajdonságai: Határozzuk meg az **IEEE-32 és 64-bites normalizált lebegőpontos rendszer** főbb jellegzetességeit.

Tekintsük először a 32-bites rendszert: r_b=2, r_e=2, m=24 a rejtett bittel együtt, de p=23!, e=8, az exponenst tároljuk Excess 127 formátumban, és a számokat tároljuk "előjel-hossz" (sign-magnitude) formátumban (ahol a mantisszát pozitívnak tekintjük). Mivel p=23 (míg m=24), ezért itt a mantisszák tartománya 1-től majdnem 2-ig terjedhet (azt már tudjuk, hogy DEC esetében ez a mantissza tartomány 1/2-től majdnem 1-ig terjed). A másik eltérés, hogy ebben a speciális esetben V_E felveheti a 255-öt, amelynek köszönhetően különleges értékeket tudunk ábrázolni, akár a ∞-t is:

V_F	Előjel	Ábrázolás jelentése
≠ 0	1,0	Nem egy szám (NaN)
0	0	+∞
0	1	-∞

Ebből és a fenti összefüggésekből a következő egyenletek vezethetők le:

$$V_{M(MIN)} = 1.\underbrace{000\ldots_2}_{23db} = 1$$

$$V_{M(MAX)} = 1.111\ldots_2 = 1.99999988 = 2.0 - 2^{-23}$$

$$V_{E(MIN)} = -126$$

$$V_{E(MAX)} = 127$$

$$V_{FPN(MIN)} = 1.000\ldots_2 \times 2^{-126} = 1.1755 \times 10^{-38}$$

$$V_{FPN(MAX)} = 1.111\ldots_2 \times 2^{+127} = 3.4028 \times 10^{38}$$

$$NLM_{FPN} = 2^{23} = 8,388,608$$

$$NLE_{FPN} = 127 + |-126| + 1_{ZERO} = 254$$

$$NRV_{FPN} = 2^{23} \times (2^8 - 1) = 2.131 \times 10^9$$

Mint láthatjuk, ebben a rendszerben a legnagyobb ábrázolható szám ($V_{FPN\ (MAX)}$) pontosan kétszer akkora, mint a DEC-ben. Ennek oka, hogy a $V_{E(MAX)}$ is a kétszeresére változott, megközelítőleg 2 (míg a DEC-nél ~1). Másrészt a legkisebb ábrázolható érték ($V_{FPN\ (MIN)}$) négyszer nagyobb, mint a DEC-nél. Egy korábbi példákban azt mondtuk, hogy egy szám akkor volt nulla, ha az összes exponensbitje nulla. A DEC-el szemben ez az IEEE rendszer már

biztosítja, hogy egy szám normalizálatlanná válhasson, amikor az a nullához linárisan közelít. Ebben az esetben is az összes exponensbitnek nullának kell lennie, de ekkor a mantissza mezőben lévő következő bitnek értékes jegynek kell lennie, valamint az exponens értékét -126-nak kell beállítani. Ez a módszer teszi lehetővé, hogy a számok a nullához lineárisan közelítsenek, és ne legyen szakadás, mint a DEC esetén. Ebben a rendszerben a nullának két különböző reprezentánsa létezik: amikor minden bit nulla (kivéve az előjelbitet) a szám értéke is nulla. Viszont a normalizálatlan ábrázolás miatt az értéktartomány 10^{-45}-re csökken. Az IEEE rendszer tehát megváltoztatja az exponensbitek számát, a dupla pontosságú ábrázolás alkalmazása miatt.

A **64 bites rendszer** esetén: $r_b = r_e = 2$ (mint az előzőnél), de m=53, p=52, e=11, és az exponenst tároljuk excess 1,023 kódolással. Ezek a módosítások valamelyest megváltoztatják a rendszer tulajdonságait: a $V_{M(MIN)}$ mindig 1.0, de a $V_{M(MAX)}$ sokkal közelebb kerül 2-höz ($2-2^{-52}$). Lényeges változás csak a következő egyenletekben van:

$$V_{FPN(MIN)} = 1.\underbrace{000\ldots_2}_{52db} \times 2^{-1022} = 2.255 \times 10^{-308}$$

$$V_{FPN(MAX)} = 1.111\ldots_2 \times 2^{+1023} = 1.798 \times 10^{308}$$

$$NLM_{FPN} = 2^{52} = 4.5 \times 10^{15}$$

$$NLE_{FPN} = 1023 + |-1022| + 1_{ZERO} = 2047 = 2^{11} - 1$$

$$NRV_{FPN} = 2^{52} \times (2^{11} - 1) = 9.214 \times 10^{18}$$

Ez a 64-bites IEEE rendszer 15 értékes jegyet tartalmaz, és az értéktartománya sokkal szélesebb, mint a DEC, vagy az IBM duplapontos rendszereké.

II.) NEM-NUMERIKUS INFORMÁCIÓ KÓDOLÁSA:

Mindeddig a bitmintázatokat egész típusú, vagy lebegőpontos információként értelmeztük, azonban a különféle feladatoknál használt adatok nemcsak számokat tartalmazhatnak. Más típusú információk, mint pl. utasítások, szöveg, címek, állapot információk ugyanúgy tárolhatók egy bitmintázatban, és a formátumra tett megkötéseket itt is, már a rendszer tervezésének kezdetekor definiálni kell. Ebben a részben röviden áttekintjük a nem numerikus információk közül a szöveges- és logikai (boolean)-információt, a grafikus szimbólumokat, és a címeket.

Mind tárolásnál, mind kezelésnél gyakran alkalmazott adattípus a szöveges információ. Kezdetekben a számítógépet, mint számítási műveletek elvégzésére hivatott eszközt fejlesztették ki, azonban az idők során ez megváltozott: manapság pl. az irodákban jelentések, levelek, szerződések és egyéb nyomtatott információk készítésénél is használják. Felmerül a kérdés: hogyan lehet megvalósítani a szöveges információ tárolását és kezelését, ill. mikből áll az ábrázolni kívánt elemek halmaza? A minimális, 14 karakterből álló halmazban lehet számjegy (0-9), tizedes pont, pozitív ill. negatív jel, és üres karakter. Azonban nem szerepelhetnek címkék, kocsi-vissza (CR: Carriage Return), soremelés (LF:Line Feed) és más vezérlőkarakterek. A fenti karakterkészlethez még hozzájön az ábécé (A-Z), a központozás, és a formátumvezérlő karakterek (mint pl. vessző, tabulátor, kocsi-vissza, soremelés, lapemelés (FF: From Feed), zárójel). Így kapjuk, hogy az elemek száma 46 lehet. De tudjuk, hogy 46 különböző érték megjelenítéséhez legalább 6 bitre van szükségünk:

$$\lceil \log_2 46 \rceil = 6 \text{ bit}$$

A halmaz 6 biten már 2^6=64 különböző elemet tartalmazhat, így a legtöbb információ ábrázolhatóvá válik, azonban nem elég nagy ahhoz, hogy mind a kisbetűs, mind pedig a nagybetűs karaktereket is magába foglalja. Ahhoz, hogy az összes karakter és írásjel ábrázolható legyen legalább 7 bitre van szükség. Ezután a bitmintázatból már leképezhetővé válnak a reprezentálni kívánt karakter- és vezérlőinformációk.

A számítógépes kommunikációban az egyik legkorábban alkalmazott eszköz a lyukkártya-olvasó volt, amely sajátos kódolási technikát használt. Mivel az ábrázolni kívánt információban csak nagybetűk, számok, és speciális karakterek szerepelhettek, ezért ezt a csökkentett karakterkészletet az olvasóhoz kifejlesztett kódolásnak tudnia kellett kezelni. Ezt a 6 bites kódolást hívjuk **BCD (Binary Coded Decimal) kódolásnak**, de nem tévesztendő össze a 4-bites BCD ábrázolással, amely csak számokat (0-9) képes megjeleníteni.

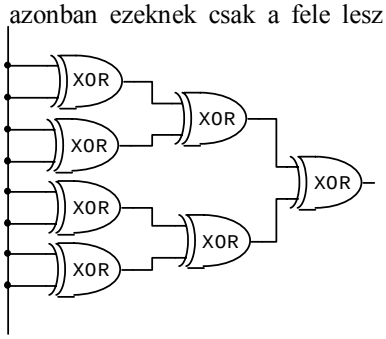
Később, ezt a 6-bites BCD kódolást kibővítették úgy, hogy kisbetűs karaktereket és kiegészítő-információkat is képes volt kezelni. Az így kapott 8-bites kódolást nevezzük **EBCDIC**-nek, melynek jelentése: **Extended Binary Coded Decimal Interchange Code** (kibővített bináris kód decimális számok ábrázolására. Könyv *A* függeléke). A 8-biten reprezentálható 256 értékből nincs mindegyik kihasználva. Fontos továbbá megjegyezni, hogy aritmetikai műveletek esetén, nem biztos, hogy mindig ugyanazt az eredményt kapjuk. Ez akkor történhet meg, ha írunk egy részprogramot, amely kiíratja az összes alfa-numerikus karaktert A-Z –ig, oly módon, hogy először A-nak a kódját rakjuk be egy regiszterbe, majd azt növeljük (inkrementáljuk), hogy megkapjuk B-nek a kódját, és így tovább. Azonban az I betűnél szakadás (üres karakterek) következik be, mivel a számsorozatot alkotó kódolás szerint, utána nem J-nek a kódja következik, sőt az R után is szakadás van. Ez az oka annak, hogy ez a kódolási technika nem terjedt el széles körben.

Egy másik, általánosan használt módszer az **ASCII kód: American Standard Code for Information Interchange** (amerikai szabványügyi hivatal által az információcseréhez kidolgozott kódrendszer. Könyv *A* függeléke). Ez a rendszer 7 biten 128 különböző karaktert képes ábrázolni: a betűk kódjának inkrementálása során már helyesen megkapjuk az ábécében egymást követő összes karaktert, a Z betű kódjának kivételével, amelyet terminálok, nyomtatók és egyéb berendezések használatánál tartanak fenn.

Hibakódolás - Hibadetektálás és Javítás

Mint már említettük, N bit segítségével 2^N féle érték, cím, vagy utasítás ábrázolható, míg más adat ábrázolásához esetleg több bitre van szükség. Ha például egy bittel növeljük az N-bites ábrázolást, akkor a reprezentálható elemek száma megduplázódik, 2^N -ről 2^{N+1} re. De hogyan határozhatók meg ezek a kiegészítő elemek, ill. milyen szabályok (kódolások) léteznek hatékony kihasználásuk érdekében? Először is meg kell vizsgálnunk ezeket a szabályokat, nemcsak azért, hogy *detektálni* tudjuk a hiba helyét, hanem később *javítani* is.

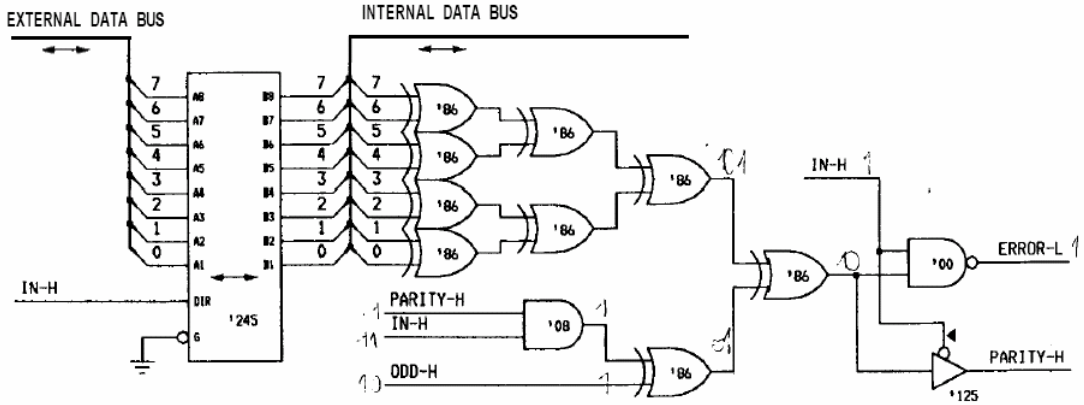
A hibafelismerés egyik legegyszerűbb módszere a **paritás bit ellenőrzés**. Ekkor az N bites információ egy kiegészítő bittel bővül (paritás bittel), amely ugyan megduplázza a reprezentálható minták számát, azonban ezeknek csak a fele lesz használható (a többi redundáns bit): így válik lehetségessé az egyszeres hiba felismerése.



Az ilyen egyszeres hibák többsége a "leragadásból" adódik (0-ból 1-es lesz, vagy fordítva), amikor is a „hibás” bit helytelen értéken áll be, de lehet egy átmeneti állapot is. Tehát ha csak egyetlen bit hibás, a javító kódszó felépítésének köszönhetően detektálni tudjuk a hibát. Az adatok végére paritásbitet kell illeszteni, és értékét úgy kell megválasztani, hogy az 1-esek száma a kódszóban páros (vagy páratlan) legyen. A mellékelt ábrán egy olyan áramkör látható, amely egy 8 bites átvitelnek megfelelő paritás bitet generál. Ha egy bittel bővítjük a XOR kapukból felépülő (Exclusive OR=Kizáró VAGY) áramköri ágakat, akkor lehetővé válik a paritás ellenőrzése 9 bitre, továbbá meg tudjuk határozni, hogy páratlan volt (vagy páros) a paritás.

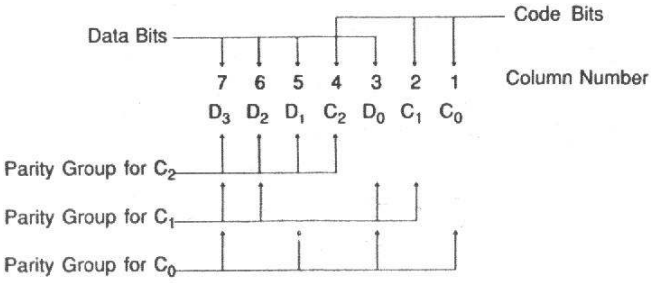
Általában az ilyen fajta hiba-detektálást mind a soros (terminál vonalak) / párhuzamos (buszok) átviteli rendszereknél, mind pedig memóriarendszereknél is használják. Azonban bizonyos esetekben nem alkalmazható paritásellenőrzés Pl: a buszrendszernél az átvitel során fellépő hiba esetén, ha olvasáskor helytelenül csupa 0 jelenik meg (ha páros paritásellenőrzést használunk) nem észleli a hibát. Ha pl. egy 16 bites buszt kiegészítünk további egy bittel (a vonalak száma 17 lesz), hasonló olvasási hiba esetén, amikor csupa 1-est olvasunk, páratlan paritást kell használni.
Egy jó módszer, 16 bites busz esetén, hogy 2 paritásbitet használunk, 8 bites vonalanként egyet-egyet. A busz egyik felének paritásérzékelőjét páratlanra, míg a másik felét párosra állítjuk be. Ekkor, ha csupa 0, vagy ha csupa 1-es érkezik, akkor a busz megfelelő részét (felét) kell átállítani. Így mindegyik esetben, a hiba előfordulásának ténye pontosan azonosítható.

Paritás ellenőrzés és generálás: Konstruáljunk egy **8 bit széles adat utat (Byte-Wide Data Path), kétirányú paritásbit ellenőrzéssel**. Azaz az útvonal egyik fele a 8 bites forrás/cél információ, míg másik része egy háromállapotú adatbusz paritásbittel. A feladat áramköri megvalósítása a következő:



A belső adatbuszról érkező adatnak keresztül kell mennie a kétirányú, háromállapotú adó-vevő egységen (’245 transceiver). Ez az adó-vevő egység egy buszmeghajtó áramkör (jelszint erősítő), amely leválasztja a belső buszt a külsőről. Ha a forgalmat figyelő (IN-H) vonal bemenő adatot észlel, a (PARITY-H) paritás vonal engedélyezi a paritásellenőrző áramkör használatát, amely megvizsgálja az adat helyességét. (Ha IN-H=1, akkor nem ad át a kimenetre adatot, ha 0, akkor a bemenet megjelenik a kimeneten, ha ODD-H=1, akkor páratlan paritást alkalmazunk.) Ha a paritásérzékelő hibát észlel, a hiba vonal (ERROR-L) engedélyeződik. Ha az ERROR-L = 0, akkor hiba történt. Amikor az IN-H vonal érzékeli, hogy az adó-vevő egység adatot tesz a buszra, a paritásgenerátor aktiválódik. A paritás vonal (PARITY-H) hasonlóképpen működik, mint amit a buszvonalnál az előbb megismertünk. A paritásellenőrző áramkört többféle módon is meg lehet valósítani tervezési szempontból.

Megfelelően választott számú „többlet” bittel (redundáns bittel) nem csak a hiba meglétét, hanem a *hiba pontos helyét* (a hibás bitet) is **azonosítani** tudjuk. Egy hibás modellben lehet leragadásos vagy ideiglenes hiba, de megpróbálhatjuk a hibákat egyszeres hibákra korlátozni egy kódszón belül. A **Hamming kód** is egy olyan kód, ahol egy biten tároljuk a bitmintázatok azonos helyiértékű bitjeinek különbségét, tehát egy bites hibát lehet vele javítani.



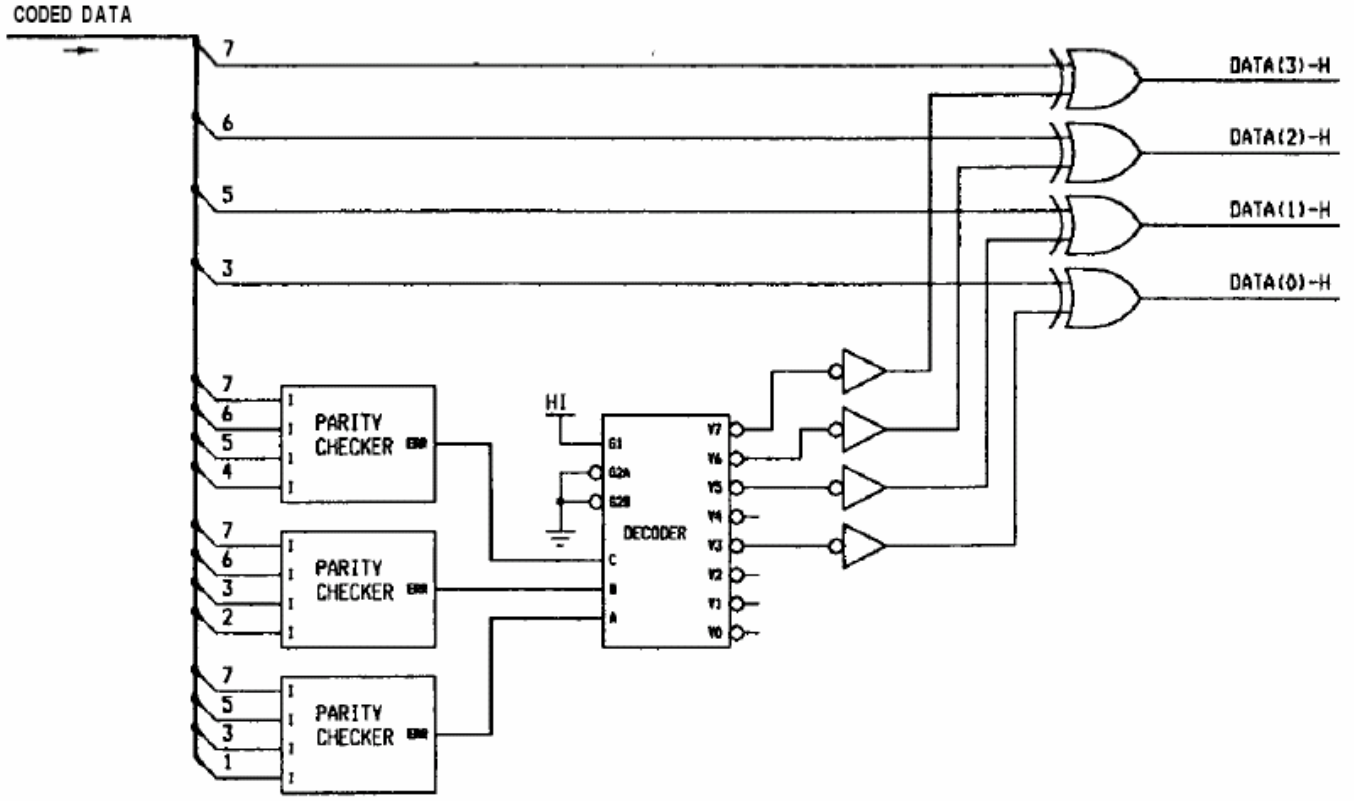
Ezzel a módszerrel próbálunk egy kódszót úgy konstruálni, hogy a hiba pontos helyét azonosítani tudjuk. A módszer bemutatásához felhasználunk *4 adatbitet (D0,D1,D2,D3), 3 kódbitet (C0,C1,C2)* tehát együttesen 7 bitet. Ezeket a biteket a mellékelt ábrán látható formába rendezzük.

Látható, hogy a kódbitek a *bináris súlyuknak* megfelelő bitpozíciókat foglalják el, tehát C0 a 2^0=1. pozícióban, C1 a 2. pozícióban, míg C2 a 4. bitpozícióban. A maradék pozíciókat rendre az adatbitekkel töltjük fel. Ha egy bitpozíció hibás, azt a megfelelő kódbit egyedi módon azonosítja, mivel minden egyes paritáscsoportban egyetlen kódbit található. A következő táblázat írja, le az egyes kódbitekhez tartozó paritáscsoportokat:

Paritáscsoportok	Bit pozíciók	Bitek jelölései
0	1, 3, 5, 7	C0, D0, D1, D3
1	2, 3, 6, 7	C1, D0, D2, D3
2	4, 5, 6, 7	C2, D1, D2, D3

Mivel 3 kódbitünk van, ezért írásnál 3 paritásbit generáló-, míg olvasásnál 3 paritás-ellenőrző áramkör kell. Bemenetüket C0-ás, C1-es, C2-es paritáscsoportba tartozó biteknek megfelelően alakítjuk ki (lenti ábra). A példaként használt bemeneti adatbit-mintázatunk legyen: 0101. Mivel páratlan paritást alkalmazunk, a megfelelő helyen szereplő kódbitekkel kiegészítve a következő szót kapjuk: 0100110. Ha nincs hiba, a paritásellenőrzők kimenete 000, minden egyes paritáscsoportra. Ha a bemeneti mintázatot a példa kedvéért direkt 0100010 -ra változtatjuk, akkor a paritásellenőrző hibát észlel. A C2.paritásbitcsoport rendben, míg C1. és C0. hibás: 011 az azonosított minta a harmadik oszlopban (D0 helyén). Javításként *invertáljuk* a 3. bitpozícióban lévő bitet. 0100010 $\Rightarrow$ 0100110. Ekkor a kódbitek a következőképpen módosulnak a páratlan paritásnak megfelelően: C0=1, C1=1 és C2=0.

A következő ábrán látható, hogy egy *7-bites Hamming kódú hibajavító áramkör* egyszerűen felépíthető három paritásellenőrzőből, négy inverterből, egy dekóderből, és négy XOR kapuból. A dekóder egység segít a hiba pontos helyének azonosításában. Ennél a módszernél tehát összesen $\mathbf{2^N-1}$ bitet használunk fel, amelyből N db kódbitünk, és $\mathbf{2^N-N-1}$ db **adatbitünk** származik. Tehát kis N-értéknél a kódbitek (kiegészítő bitek) számaránya nagy, az adatbitek számához képest, míg nagyobb N-értéknél csökken a kódbitek száma. Ez a rendszer SEC (Single Error Detection) = Egyszeres hibadetektáló.



(DED: Double Error Detection): kétszeres hibát felismerő rendszer. Mivel az előző rendszer kettős hiba esetén rossz bitpozícióban jelezne hibát, és az eredmény is hibás lenne, ezért alkalmazzuk a DEB-et (Double Error Bit). Először egyszeres paritásbit ellenőrzést alkalmazunk mind az adatbiteken, mind pedig a kódbiteken (az adatbiteken lévő hibákat a kódbitek jelzik), majd a kódbitek ellenőrzése után kapjuk meg a DEB-t. A módszer feltételei a következő táblázatból kiolvashatók:

Paritásfeltétel		Jelentése:
DEB	Kódbitek	
jó	Jó	Nincs hiba – normál feltétel
hibás	Hibás	Egyszeres hiba – a hiba helyét a kódbitek bináris súlya határozza meg
hibás	Jó	Egyszeres hiba – DEB hibás
jó	Hibás	Kettős hiba – nem javítható

Példa: 8 bites Hamming-kód (DEB-el): mi a pontos ábrázolása 8 biten a 01011100 adatbit-mintázatnak? Szükséges 8 adatbit (D0-D7), 4 kódbit (C0-C3) és a kettős-hibajelző bit (DEB). *Páratlan* paritást alkalmazunk. (BW-binary weight jelenti az egyes oszlopok bináris súlyát, 1, 2, 4 ill. 8 biten). Általános felírás a következő:

13	12	11	10	9	8	7	6	5	4	3	2	1	Oszlopszám
DEB	D7	D6	D5	D4	C3	D3	D2	D1	C2	D0	C1	C0	
	1	1	1	1	1	0	0	0	0	0	0	0	BW, 8bit
	1	0	0	0	0	1	1	1	1	0	0	0	BW, 4 bit
	0	1	1	0	0	1	1	0	0	1	1	0	BW, 2 bit
	0	1	0	1	0	1	0	1	0	1	0	1	BW, 1 bit

Paritáscsoportok	Bit pozíciók	Bitek jelölései
0	1, 3, 5, 7, 9 ,11	C0, D0, D1, D3, D4, D6
1	2, 3, 6, 7, 10, 11	C1, D0, D2, D3, D5, D6
2	4, 5, 6, 7, 12	C2, D1, D2, D3, D7
3	8, 9, 10, 11, 12	C3, D4, D5, D6, D7

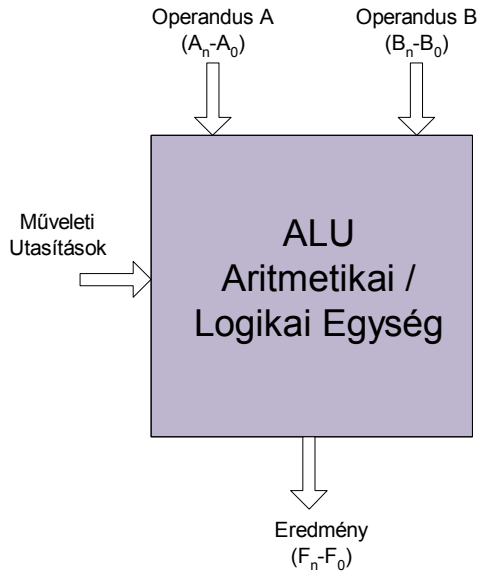
A példa felírása a következő:

13	12	11	10	9	8	7	6	5	4	3	2	1	Oszlopszám
DEB	D7	D6	D5	D4	C3	D3	D2	D1	C2	D0	C1	C0	
	0	1	0	1	_	1	1	0	_	0	_	_	Adatbitek
$\bar{1}$	0	1	0	1	$\bar{1}$	1	1	0	$\bar{1}$	0	$\bar{0}$	$\bar{0}$	Kiegészített kódbitek.

Tehát a helyes ábrázolása 01011100-nek a következő:
1010111101000.

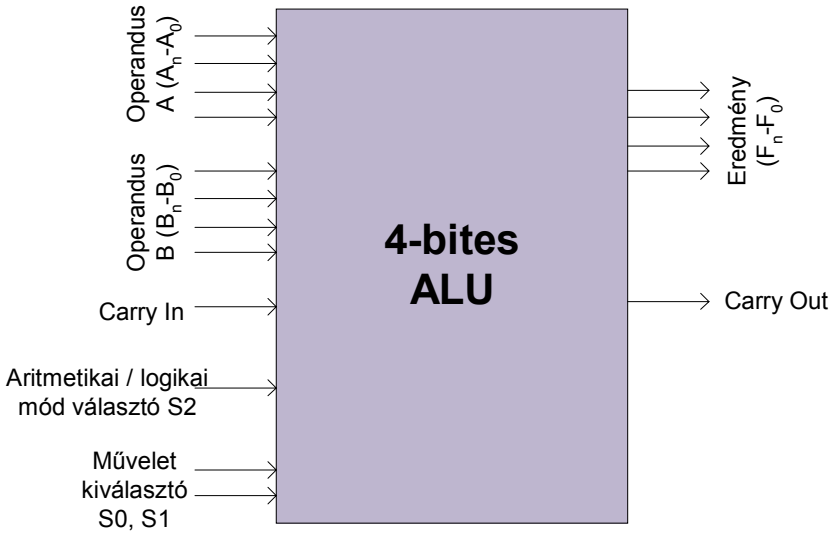
2. ALU felépítése és működése:

ALU (Arithmetic Logic Unit) = Aritmetikai / Logikai Egység
Olyan adatkezelő egység, amely matematikai / logikai műveletek végrehajtását végzi. Matematikai műveletek lehetnek: összeadás, kivonás, szorzás, osztás (különféle eljárások és módszerek felhasználásával). Logikai műveletek lehetnek: AND, OR, NOT, NOR, NAND, XOR, XNOR, EQ stb. amelyeket minimális számú kapu felhasználásával valósít meg, ezáltal minimális késleltetéssel.
Kombinációs logikai hálózatok esetén a NAND ill. NOR logikai alapkapuk univerzálisak, mivel az összes logikai függvény megvalósítható belőlük különböző kombinációval.
Ez a megállapítás igaz az aritmetikai műveletekre nézve is, ahol egyedül az összeadót tekintjük alapvető építőelemnek. Segítségével az összes aritmetikai művelet (kivonás, szorzás, osztás) származtatható.
Utasítások hatására az (S0-Sn) vezérlőjelek kijelölik a végrehajtandó aritmetikai v. logikai műveletet. További adatvonalak kapcsolódhatnak közvetlenül a státusz regiszterhez, amely fontos információkat tárol el: pl. carry-in, carry-out átviteleket, előjel bitet (sign), túlcsordulást (overflow), vagy alulcsordulást (underflow) jelző biteket.



Státuszbitek: az aritmetikai műveletek eredményétől függően hibajelzésre használatos jelzőbitek. Általában 4-féle státuszbitet különböztetünk meg: előjel, zéró, túlcsordulás (overflow), átvitel (carry). Ezeket a státuszregiszterekben tároljuk el. A státuszbitek megváltozása az utasításkészletben előre definiált utasítások végrehajtásától függ.
- *előjelbit*: legkönnyebb generálni. Az eredmény előjelétől függ: 2-es komplemens képzés esetén az MSB (legnagyobb helyiértékű bit) jelöli az előjelbitet. Ha MSB=1, akkor a szám negatív, különben pozitív. Közvetlenül a státuszregiszterbe töltődik. Bizonyos utasítások, aritmetikai műveletek. (pl. összeadás, kivonás, összehasonlítás) módosíthatják csak az előjelbitet. Olyan utasítások amelyek nem aritmetikai jellegűek, nem változtatják meg ezen bit értékét. (pl. ugrás, eljáráshívás).
- *carry vagy átvitelt kezelő bit*: Ha egy aritmetikai művelet átvitelt eredményez egyik helyiértékről a másikra, akkor a státuszregiszter beállítja az átvitelt kezelő bitet.
- *zéró bit*: Ha egy művelet eredménye nulla, akkor a státuszregiszterben beállítjuk a zéró bitet. Általában a zéró bit megváltozását a logikai műveletek befolyásolják (az aritmetikai műveletek mellett). Pl MOVE utasítás teszteli az ALU kimeneti vonalait, hogy azok 0-ák -e, vagy sem.
- *túlcsordulás bit*: jelzi, hogy a rendszer számábrázolási tartományán egy adott aritmetikai művelet eredménye kívül esik-e, vagy sem, tehát a számot tudjuk-e ábrázolni, vagy sem. Ekkor beállítódik ez a jelzőbit a státuszregiszterben. Túlcsordulás egy 2-es komplemens rendszer esetén akkor fordulhat elő, ha két pozitív számot összeadva eredményként negatív számot kapunk, ill., ha két negatív számot összeadva pozitív számot kapunk. (A túlcsordulás detektálását úgy tudjuk megvalósítani, hogy figyeljük a számok előjeleit. Ha a számok előjelbitjei pozitívak, vagyis ’0’-ák, és az eredmény előjele negatív, vagyis ’1’-es, akkor túlcsordulás következett be. Ennek detektálása két AND illetve egy OR kapuval megvalósítható.

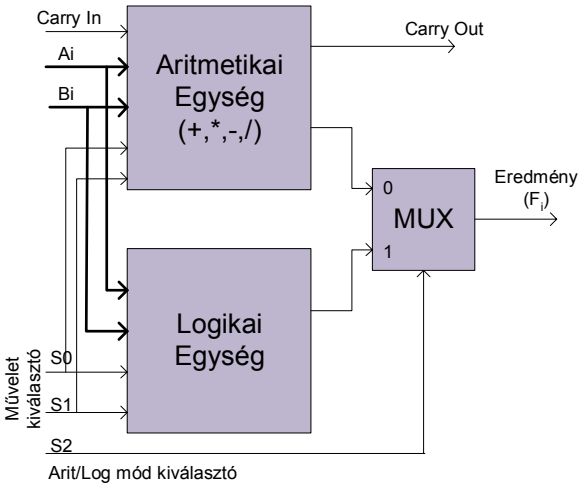
ALU felépítése és működése: Az alábbi ábrán két 4-bites A ill. B számon tudunk aritmetikai / logikai műveletet végrehajtani. Az F eredmény a 4-bites kimeneten képződik. A Carry-in / Carry-out az átvitel kezelésére szolgáló be ill. kimeneti vonal. Az S2 vonal segítségével egy MUX-on (multiplexer) keresztül kiválasztjuk, hogy aritmetikai vagy logikai műveletet szeretnénk végrehajtani (aritmetikai műveleteknél S2=”0”, míg logikainál S2=”1”). Az S1, S0, és Cin megfelelő értékétől függ, hogy pontosan milyen műveletet hajtunk végre. (lásd táblázat). **4-bites ALU szimbólikus rajza:**



ALU működését leíró függvénytáblázat:

Művelet kiválasztás:				Művelet:	Megvalósított függvény:
S2	S1	S0	Cin		
0	0	0	0	F=A	A átvitele
0	0	0	1	F=A+1	A értékének növelése 1-el (increment)
0	0	1	0	F=A+B	Összeadás
0	0	1	1	F=A+B+1	Összeadás carry figyelembevételével
0	1	0	0	$F = A + \overline{B}$	A + 1's komplemens B
0	1	0	1	$F = A + \overline{B} + 1$	Kivonás
0	1	1	0	F=A-1	A értékének csökkentése 1-el (decrement)
0	1	1	1	F=A	A átvitele
1	0	0	0	$F = A \wedge B$	AND
1	0	1	0	$F = A \vee B$	OR
1	1	0	0	$F = A \oplus B$	XOR
1	1	1	0	$F = \overline{A}$	A negáltja

ALU felépítése:

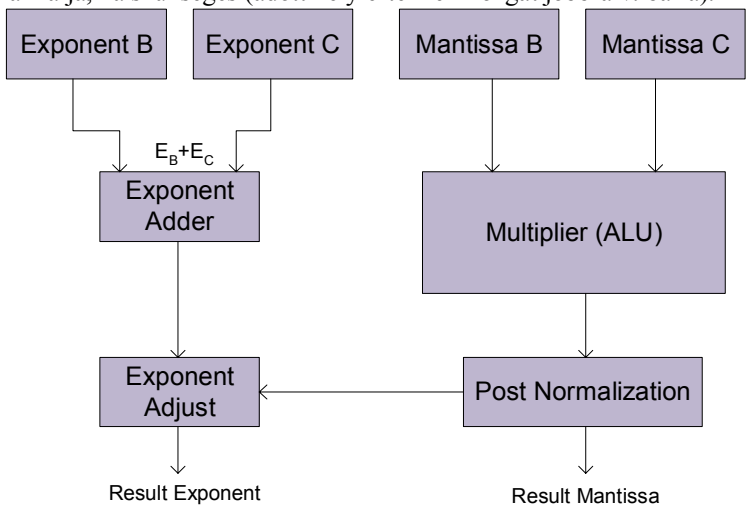


LEBEGŐPONTOS MŰVELETVÉGZŐ EGYSÉGEK:

a.) Lebegőpontos szorzó: A szorzás legegyszerűbben megvalósítható lebegőpontos művelet, mivel nincs szükség az operandusok (műveleti értékek) beállítására, és minimális post-normalizációt kell csak végezni. A=B×C, ahol A szorzat értéke, B szorzandó és C a szorzó (r: a számrendszer alapja).

$$A = B \times C = M_B \times r^{E_B} \times M_C \times r^{E_C} = (M_B \times M_C) \times r^{E_B + E_C}$$

A szorzat mantisszájának értékét tehát egyszerűen csak a két mantissza (M_B, M_C) szorzatából, míg exponensének értékét a két exponens (E_B, E_C) összegéből kapjuk meg. Ezért kell egy Mantissza Multiplier szorzó és egy Exponent Adder összeadó blokk. Az egyedüli nehézség abból adódhat, ha speciális lebegőpontos rendszert alkalmazunk. Pl. IEEE-32 esetén az exponens Excess-127 kódolású, ezért az Exponent Addert a célnak megfelelően kell megtervezni. A Post Normalizaton blokk a szorzás eredményét 0 és 1 közé normalizálja, ha szükséges (adott helyiértékkel mozgat jobbra v. balra).



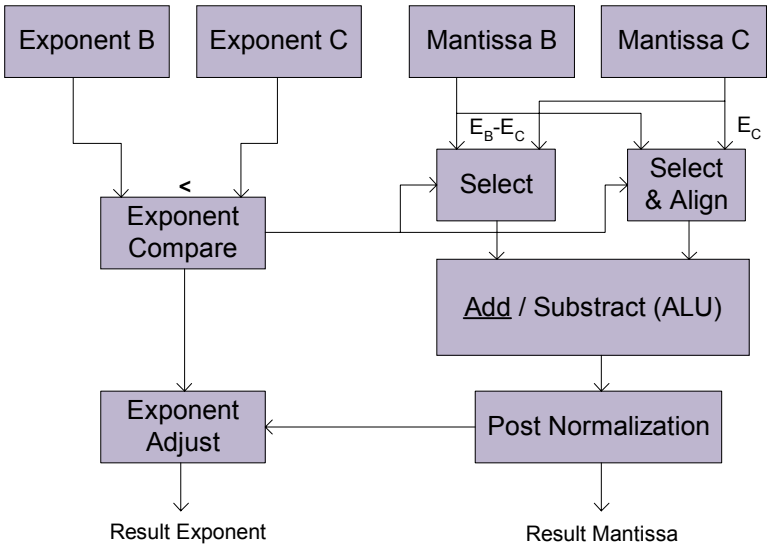
b.) Lebegőpontos összeadó:

Lebegőpontos formában történő adattárolás megnöveli a műveletvégzés összetettségét, komplexitását, és új hardver elemeket kell bevezetni. Lebegőpontos összeadásnál kell egy kivonó egység is, hogy előjeles (sign / magnitude) módszerrel tárolni tudjuk az információt (az előjel független a mantisszától). Ha az egyik szám negatív, akkor azt kivonásként értelmezi! Végezzük el a következő műveletet: A= B+C, ahol B és C lebegőpontos szám (operandus), felírható a következő alakban:

$$A = B + C = M_B \times r^{E_B} + M_C \times r^{E_C} = (M_B \times r^{E_B - E_C} + M_C) \times r^{E_C}$$

A két lebegőpontos szám összeadásakor általában, ha a mantisszák hossza nem egyezik meg, tehát ha az MSB bitek nincsenek azonos helyiértéken, ezért azokat először be kell állítani (align). A beállítást MUX-okkal lehet megvalósítani. Megnézzük melyik operandus értéke kisebb (legyen 0<B<C), majd a mantisszáját a nagyobbikéhoz igazítjuk. Ezt a mantissza jobbra shiftelésével / léptetésével érhetjük el, így a megfelelő helyiértékeken a mantissza bitek egymás alatt sorakoznak fel. A shiftelések számát az exponensbitek (E_C és E_B) különbsége határozza meg: $|E_B - E_C|$-vel jobbra shifteljük a kisebb értékű operandus mantisszáját, ami az exponensben jelent változtatást.

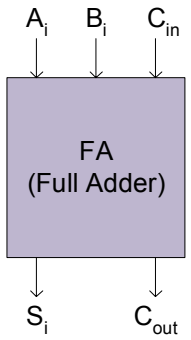
A blokkdiagram aritmetikai részét az Add/Substract egység jelenti. Két exponens összehasonlítását az Exponent Compare végzi el, melynek eredménye a SELECT/ALIGN egységbe kerül, amely helyreigazítja a megfelelő mantisszát. Az összeadás/kivonás eredménye egy utólagos normalizálás után a kimenetre kerül (POST NORMALIZATION). Erre azért van szükség, mivel 0 és 1 között szeretnénk megkapni a végeredményt (normalizáltan), így megfelelő bitpozícióval tud jobbra vagy balra is shiftelni.

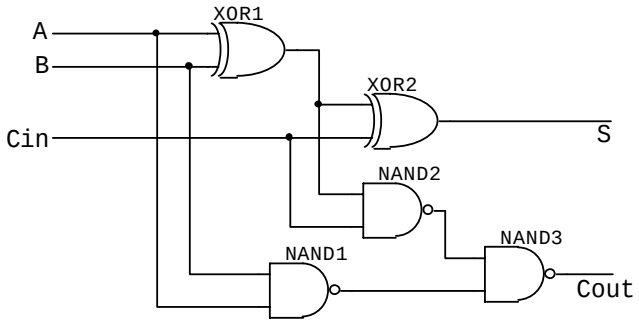


3. Összeadó / Kivonó áramkörök:

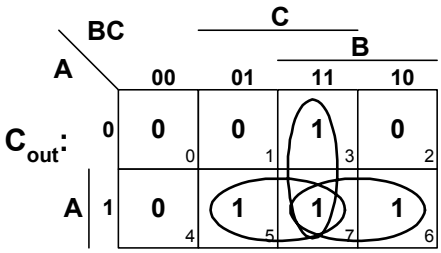
a.) Teljes összeadó (Full Adder - FA): Kapcsolási elrendezését tekintve igen sok változatát dolgozták ki az alkalmazott technológiától, optimalizálástól függően. Ma már elsősorban CMOS technológiát alkalmaznak, amelyek logikai rendszerük szerint lehetnek sztatikusak vagy dinamikusak (fázisjelekkel működő), ill. az optimalizálás történhet minimális kapu felhasználásával (minimális méret), vagy párhuzamosan működő komplex kapuk felhasználásával, ahol a hangsúly a minimális késleltetési időn van (nagy átviteli sebesség, működési sebesség). A következő ábrán egy 1 bites Full Adder / Teljes Összeadó igazságtáblája, blokk diagrammja és egy lehetséges logikai kapcsolási terve látható, amely minimális számú CMOS kapuból épül fel:

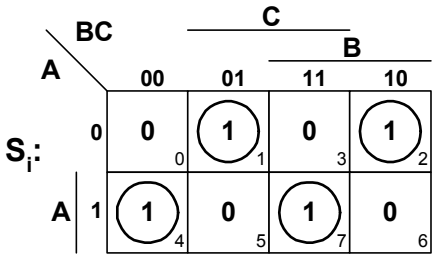
A_i	B_i	Cin	Sum_i	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1





A rajzon bejelölt $\mathbf{A_i}$ és $\mathbf{B_i}$ jelenti a két összeadandó bitet, **Cin** a kisebb helyiértékű állapotból jövő carry in-t (bemeneti átvitelt kezelő bitet), $\mathbf{Sum_i}$ a részletösszeget, míg **Cout** a következő magasabb helyiértékű állapotba történő átvitelt (átvitel ki). Az igazságtáblázatból kiolvasható, hogy a Sum_i vonal értéke akkor '1', ha páratlan számú '1'-est tartalmaznak az A_i ,B_i és Cin bemenő vonalak. A Cout kimeneti átvitel értéke pedig akkor '1', ha az A_i , B_i is '1', vagy pontosan az egyik közülük '1' és Cin értéke szintén '1' (tehát legalább két '1'-est tartalmaz A_i ,B_i és/vagy Cin). Az igazságtáblából felírhatók az Si és a Cout kimeneti vonalakhoz tartozó **Karnough táblák**, amelyekből egyszerűsítések után megkapjuk az Si és Cout értékeinek kiszámítási formuláit.

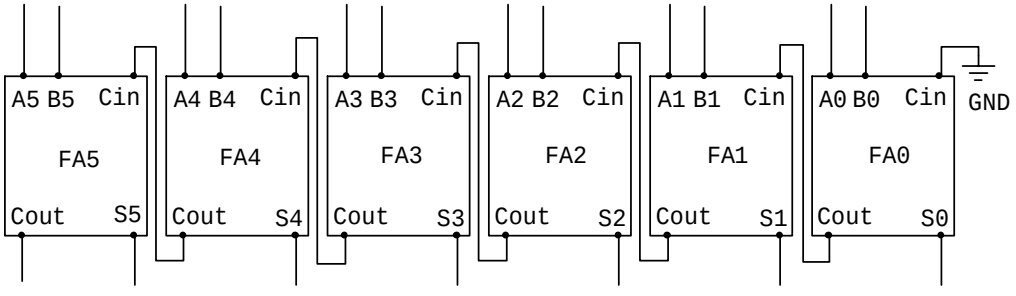




$$C_{out} = A \cdot B + A \cdot C_{in} + B \cdot C_{in} \qquad S_i = \overline{A_i} \cdot \overline{B_i} \cdot C_{in} + \overline{A_i} \cdot B_i \cdot \overline{C_{in}} + A_i \cdot \overline{B_i} \cdot \overline{C_{in}} + A_i \cdot B_i \cdot C_{in} = A_i \oplus B_i \oplus C_{in}$$

Tehát Si részletösszeg - egyszerűsítések után - felírható Ai, Bi és Cin bemenetek kizáró vagy kapcsolataként, mivel két XOR kapu segítségével generáltuk (XOR1 és XOR2 kétszintű logikával megvalósítható). A teljes összeadónak ezen kívül többféle implementációja is létezhet más-más logikai kapukból felépítve!

b.) Átvitelkezelő összeadó (RCA=Ripple Carry Adder): A teljes összeadókat legfőképpen két, több bitből álló szám (bitsorozat) összeadásakor használjuk. Ezt úgy valósíthatjuk meg, hogy több 1 bites full addert egymás után sorba kapcsolunk. Az ábra 6 db 1 bites full adder összekapcsolását mutatja, hiszen így tudunk két 6 bites számon műveletet végezni. Az eredmény szintén 6 bites lesz. Az ilyen kapcsolási módot átvitelt kezelő összeadónak (RCA: Ripple Carry Adder) nevezzük. Hívják még szó összeadónak (word adder) is pl 8 db FA-ból álló RCA-t.
Az RCA-nál a carry a legkisebb helyiértéktől (LSB) a legnagyobb helyiértékű (MSB) bitek felé (C0-C5 ig) „vándorol". Az FA0 összeadó Cin bitjét 0-nak kell tekinteni (földre kell kötni: GND), míg a Cout kimenő átvitelt a szomszédos FA1 összeadó Cin bemenő átvitel vonalára kell kötni. Ezzel biztosítjuk a carry mozgását a magasabb helyiértékek felé, két összeadó között mindig csak egy lépésben. Az Si-k jelentik a részletösszegeket, a „teljes összeadó" végeredménye az S5-S0 értékével egyezik meg.

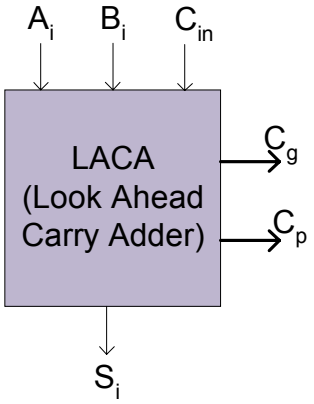


Ez nem a leggyorsabb módja két szám összeadásának, mivel a megfelelő biteket (Ai és Bi, i=0,1...5) nem lehet azonnal összeadni. Az átvitelt minden egyes bitpozíciónál (helyiértéken) meg kell várni, és a bemenő bitekhez (a korábban már említett formulákkal) kell hozzáadni. Az RCA időkésleltetése a következő képlettel számítható ki (itt N=6 esetén):

$$T_{(RCA)} = N*T_{(FA)} = N*(2*G) = 12\ G \text{ (6 bites RCA esetén)}$$

amelyből látszik, hogy két N bites (esetünkben 6 bit) szám összeadásához, N-szer több idő kell, mint két 1 bites szám összeadásához (ehhez ismernünk kell T_{FA} pontos értékét). A G itt egy kapu késleltetését jelenti, amely technológia függő paraméter (CMOS, nMOS, pMOS stb.) Tát az RCA esetében az időkésleltetés egyenesen arányos (lineáris) az összeadandó bitek számával. Kevesebb számú kapuból épül fel, mint a most ismertetésre kerülő LACA, de lassabb a működése.

c.) LACA=Look Ahead Carry Adder:



A fenti két számítási képletet a következő módon alakítjuk át:

$\boxed{Cout} = A\,B + A\,Cin + B\,Cin = \boxed{A\,B + Cin * (A+B)}$

$S = \bar{A}\,\bar{B}\,Cin + \bar{A}\,B\,\bar{Cin} + A\,\bar{B}\,\bar{Cin} + A\,B\,Cin = A \oplus B \oplus Cin$

Az S kiszámítása két kapukésleltetésű (két XOR miatt), míg Cout kiszámításának első formája szintén két kapu, addig második formája három kapukésleltetésű (két OR és egy AND). A Cout második formájában a tagok kétféleképpen csoportosíthatók:

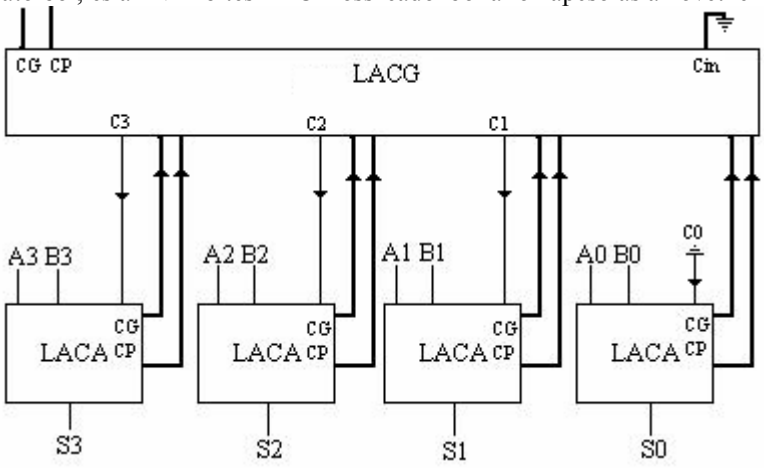
- *CG*=AB az átvitelt generáló (**carry generate**: CG) függvény, mivel ha ez a tag engedélyezett, carry generálódik (az előző állapot Cin-jétől függetlenül)
- *CP*=A+B az átvitelt továbbító (**carry propagate**: CG) függvény, amely az aktuális állapotig létrejött carry-ket továbbítja a következő állapot felé.

Tehát $\boxed{Cout=CG+Cin*CP}$

Látható, hogy egyik sem a Cin függvényeként generálódik. Mind a CG, mind pedig a CP egy kapukésleltetésű. A *LACG*: Look Ahead Carry Generator egység ált. egy b=4 bites ALU, minden egyes állapotban a carry generálásáért felel a CP és CG vonalakon érkező jeleknek megfelelően. Megfelelő számú kapuval megtervezett LACG esetén, minden carry akár két kapukésleltetés alatt generálható. Tehát a LACA számítási időszükséglete a következő:

$$T_{LACA} = 2 + 4 \times (\lceil \log_b(N) \rceil - 1)$$

Ahol „N” jelenti az összeadandó bitek számát (A és B is N bites szám), és „b” a LACG hány db LACA-ból épül fel, és így hány bitet tud kezelni („b” a logaritmus alapja). Minél nagyobb a „b”, annál gyorsabb az összeadás.
A b=4 bites LACG generátorból, és az N=4 bites LACA összeadókból álló kapcsolás a következő:



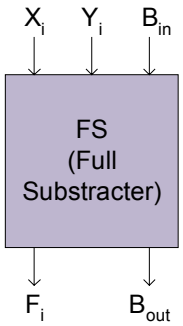
A LACA gyorsabb működésű, mint az RCA, de sokkal több kapuból épül fel, ezért nagyobb lesz a helyszükséglete.
Pl. Legyen N=4 összeadandó bitek száma, b=4 bites LACG. Ekkor az időkésleltés a következő:

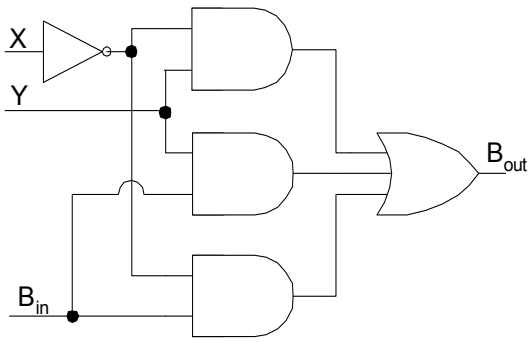
$$T_{LACA} = 2 + 4 \times (\underbrace{\lceil \log_4(4) \rceil}_{1} - 1) = 2$$

KIVONÓ ÁRAMKÖR:

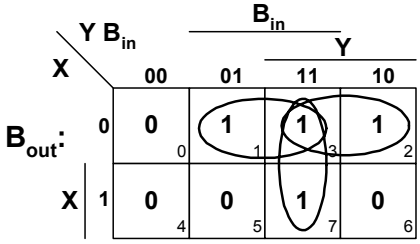
d.) Teljes kivonó (Full Substractor - FS): nem teljesen az összeadókhoz tartozik, de itt kell megemlítenünk a teljes kivonó áramkört is. A következő ábrán egy 1 bites Full Substractor (FS)/ Teljes Kivonó igazságtáblázata, blokk diagrammja, és - itt csak a Bout kimeneti függvényt megvalósító - logikai kapcsolási rajza látható, amely minimális számú kapuból épül fel (lásd. a következő oldal tetején):

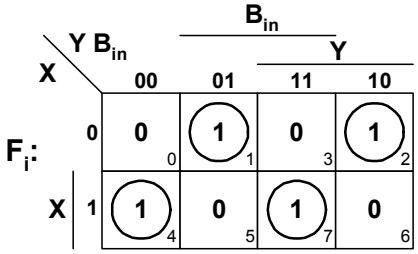
X_i	Y_i	Bin	F_i	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1





A rajzon bejelölt $\mathbf{X_i}$ és $\mathbf{Y_i}$ jelenti a két kivonandó bitet (Xi – Yi), **Bin** a korábbi állapotból érkező „Borrow in”-t (átvitel kivonásnál), $\mathbf{F_i}$ a különbségeket mindenegyes bitre, míg **Bout** a következő állapotba a „Borrow out”-ot jelenti. Az igazságtáblázatból kiolvasható, hogy a F_i vonal értéke akkor ’1’, ha páratlan számú ’1’-est tartalmaznak az X_i ,Y_i és Bin bemenő vonalak. Bout értéke erősen függ az $\overline{X}i$ értékétől. Bout értéke ’1’, ha Xi=’0’ esetén Yi és Bin közül legalább az egyik ’1’, vagy Xi=’1’ esetén Yi és Bin is ’1’. Az igazságtáblából felírhatók az Fi és a Bout kimeneti vonalakhoz tartozó Karnough táblák, amelyekből egyszerűsítések után megkapjuk az Fi és Bout értékeinek kiszámítási formuláit.





$$B_{out} = \overline{X_i} \cdot Y_i + \overline{X_i} \cdot B_{in} + Y_i \cdot B_{in}$$

$$F_i = \overline{X_i} \cdot \overline{Y_i} \cdot B_{in} + \overline{X_i} \cdot Y_i \cdot \overline{B_{in}} + X_i \cdot \overline{Y_i} \cdot \overline{B_{in}} + X_i \cdot Y_i \cdot B_{in} = X_i \oplus Y_i \oplus B_{in}$$

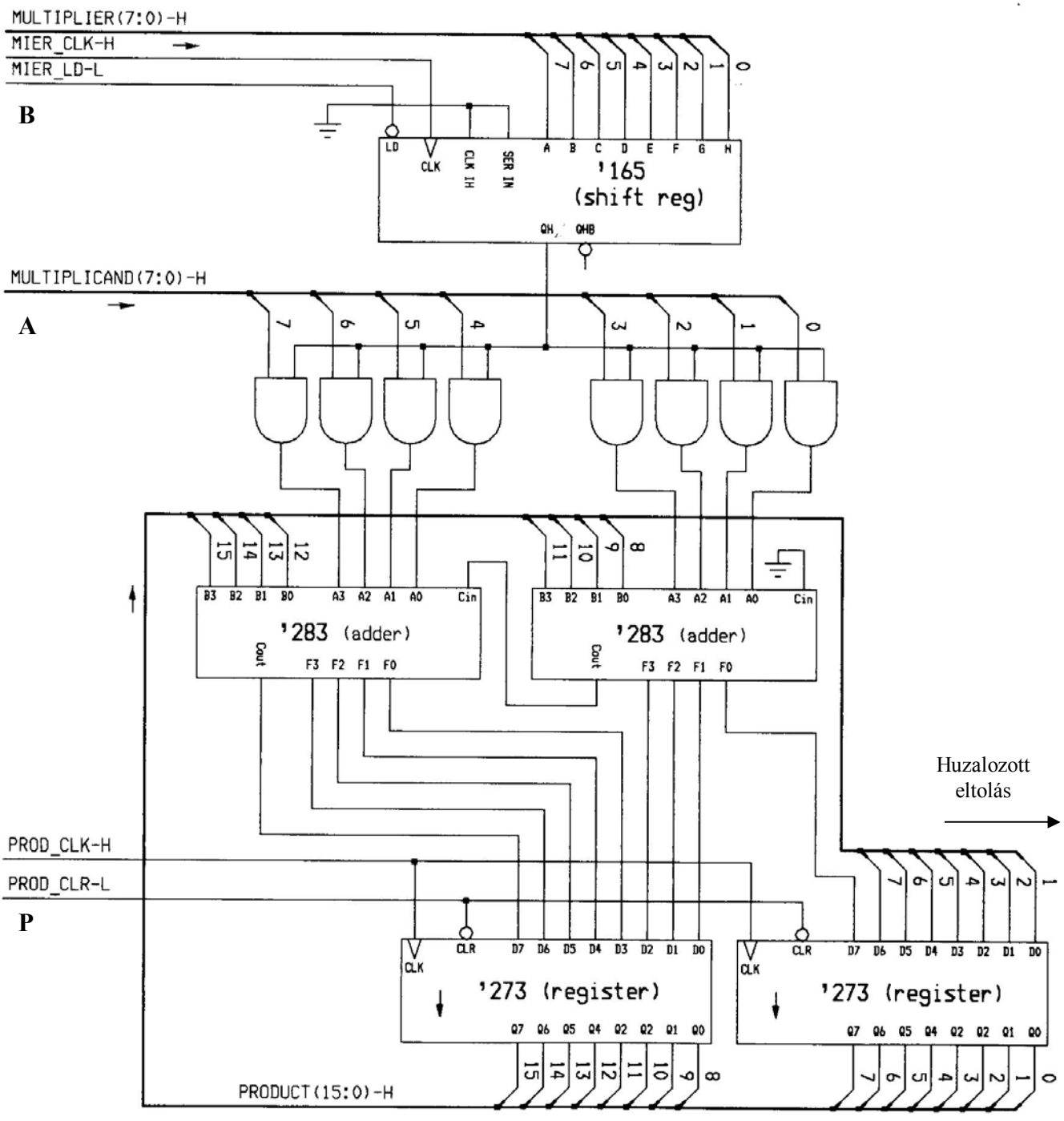
Tehát Fi különbségek (hasonlóan a Full Adderhez) – az egyszerűsítések után – könnyen felírhatók Xi, Yi és Bin bemenetek kizáró vagy kapcsolataként, mivel két XOR kapu segítségével generáltuk (XOR1 és XOR2 kétszintű logikával megvalósítható). A teljes kivonónak is ezen kívül többféle implementációja létezhet más-más logikai kapukból felépítve!

4. Szorzó áramkörök:

I. Iteratív szorzási módszerek:

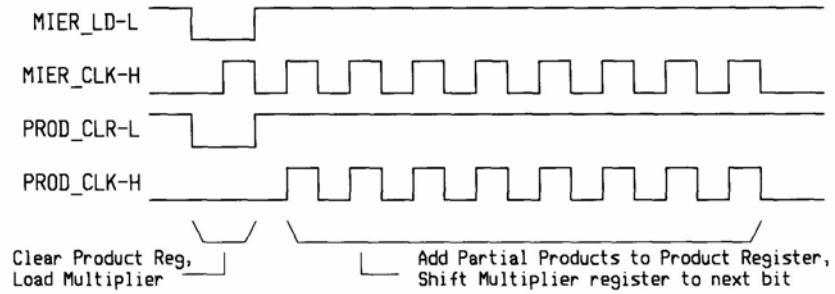
a.) SHIFT&ADD módszerrel működő szorzó:

Ez az egyik legáltalánosabb módszer a szorzás műveletének (P=A*B) elvégzéséhez. Ha két 8 bites számot szorzunk össze (8x8), az eredmény 2*8=16 bites szám lesz (mivel két N bites szám szorzata 2*N bites számként ábrázolható). A ’283 egy 4-bites teljes összeadó: a parciális szorzat-összegeket a legkisebb helyiértékű parciális szorzattól a legnagyobb helyiértékűig adja össze (LSB⇒MSB). Ezért kettő kell belőle: 2*4=8 bites összeadóvá kötjük őket össze. A jobb alsó ’273 regiszter bemenetét 1 bittel jobbra eltoltuk. Így *huzalozott eltolással* valósul meg a részeredmények shiftelése, ezáltal a parciális szorzatok a megfelelő pozícióba kerülnek (visszacsatolás után). A parciális szorzatokat (PPi-ket) 8 AND kapuval generáljuk, amelyekbe a szorzandó szám (A) megfelelő helyiértékű bitjei vannak bevezetve (LSB-MSB). Az eltolt szorzó biteket (B) a 165’ SHIFT regiszterből kapjuk meg, amelyeket a megfelelő AND kapuk bemeneteire kell kötni, szintén (LSB-MSB) helyirétékek szerint! A ’165 egy 8 bites SHIFT regiszter, amely párhuzamosan írható / sorosan kiolvasható.



A MIER_CLK-H (B) magas-aktív órajel vezérli a 165’-ös SHIFT regisztert, amely ciklusonként csak 1 bitet tölt be, de a MIER_LD-L load jel hatására, képes az összes bemenetére érkező jelet egy lépésben betölti, gyorsítva ezzel a folyamatot (mindig a kapu késleltetés határozza meg az órajelet!). A ’273 párhuzamosan írható ill. olvasható 8 bites regiszterek (D flip-

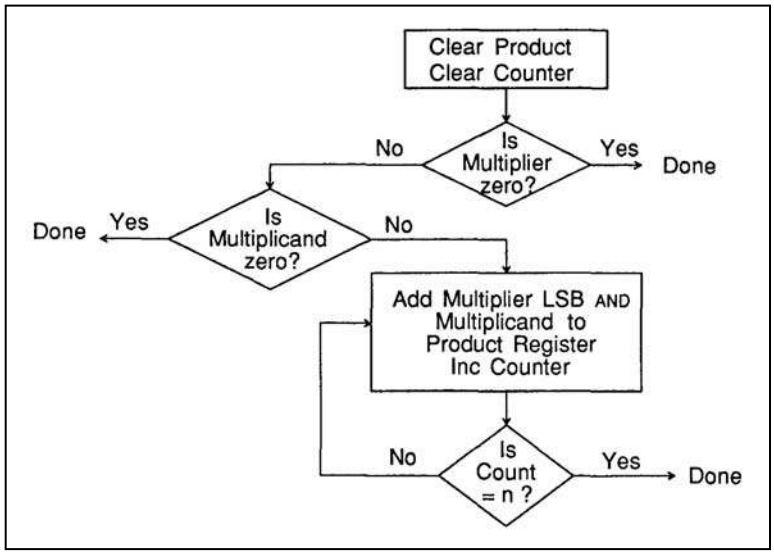
flopok), a részeredmények tárolását szolgálják, amik visszacsatolódnak a 283’ 4-bites összeadók bemeneteire. A PROD_CLK-H itt is magas-aktív órajel, míg a PROD_CLR-L egy törlőjel, amely hozzáadás előtt törli a regiszter tartalmát, nehogy hibás értékkel íródjon felül. Ezt az időzítési diagrammot mutatja a következő ábra:



Ennél a hagyományos shift-add rendszerű módszernél két (A,B) 5 bites szám összeszorzása a következőképpen zajlik:

					A4	A3	A2	A1	A0
				x	B4	B3	B2	B1	B0
PP0					A4*B0	A3*B0	A2*B0	A1*B0	A0*B0
PP1				A4*B1	A3*B1	A2*B1	A1*B1	A0*B1	
PP2			A4*B2	A3*B2	A2*B2	A1*B2	A0*B2		
PP3		A4*B3	A3*B3	A2*B3	A1*B3	A0*B3			
PP4	A4*B4	A3*B4	A2*B4	A1*B4	A0*B4				
PR	az egyes oszlopok összege								

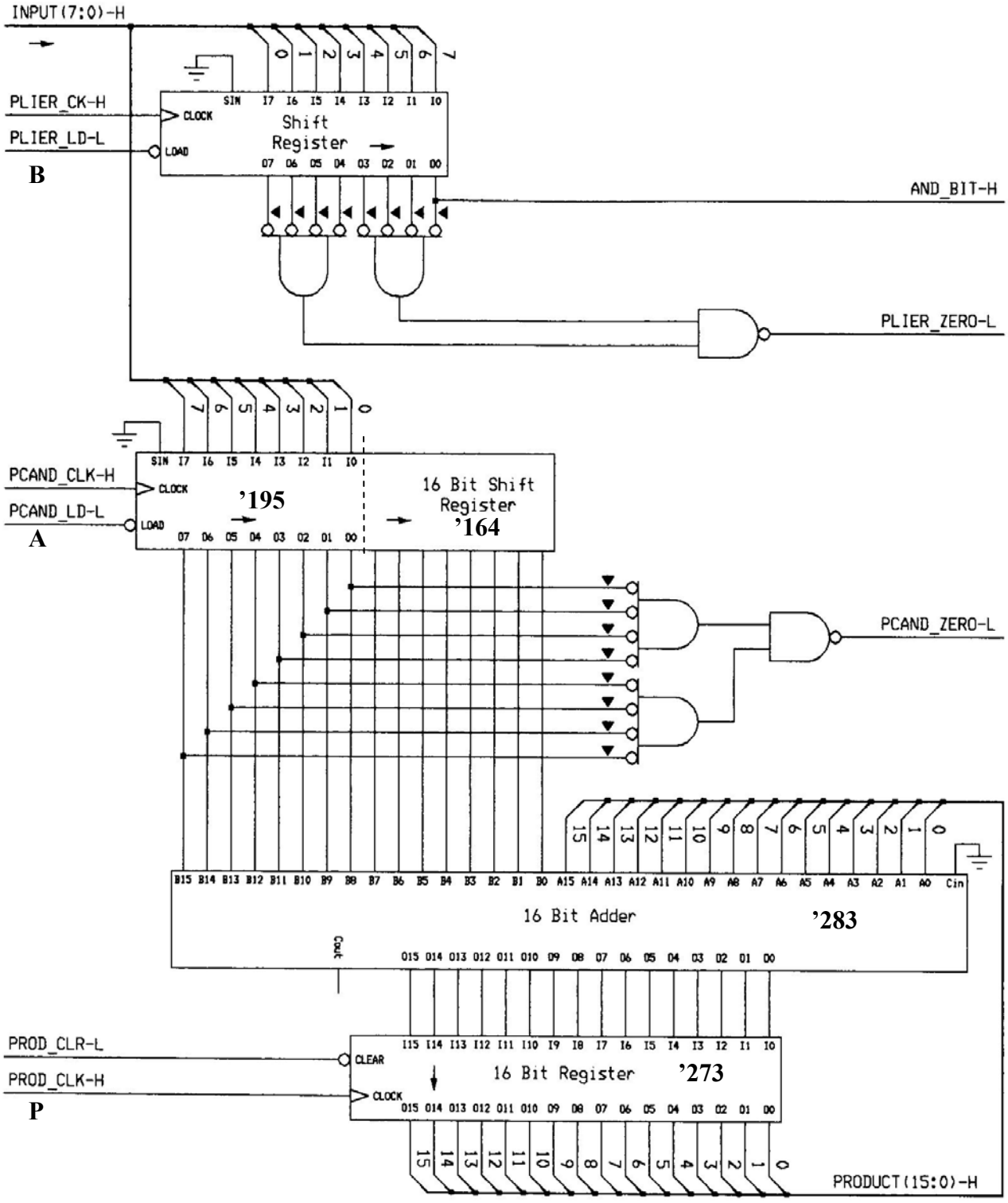
A PPi –k a parciális szorzatokat jelölik egy sorban. A módszer alapvetően adatfüggetlen algoritmus. Azonban megfelelő vezérlő áramkörök generálásával adatfüggő szorzási algoritmust is megvalósítható. Ha tesztelési feltételeket adunk, az algoritmus futási ideje felgyorsítható. (Például ha a két szám közül valamelyik 0, akkor a szorzat eredménye is 0 lesz, így azt nem kell kiszámolni. Az eredményt azonnal megkaphatjuk. De a bemenetek értékeit figyelnünk kell!) Ezt szemlélteti a következő adatfolyam diagramm:



b.) Fordított sorrendű szorzó:

P=AxB végrehajtása, ahol A és B két 8-bites szám, az „A” szorzandó, „B” szorzó. Azért fordított sorrendű, mivel a parciális szorzatokat a SHIFT&ADD módszerhez képest *fordított sorrendben* adja össze: a B szorzó biteket a legnagyobb helyiértékűtől a legkisebbek felé haladva (MSB⇒LSB)! Ehhez a módszerhez egy olyan bitszélességű összeadó kell, mint amilyen eredmény szorzat lesz: ilyen a ’283 jelű 16 bites gyors összeadó átvitelgyorsító (Look Ahead) képességgel. Az adderből érkező szorzatot egy ’273 –as 16 bites él-vezérelt regiszter tárolja. Ebből visszacsatolással visszatölthetők a bitek a 16 bites összeadó megfelelő bemeneteire, de itt ugyanarra a bemeneti bitpozícióra A0-A15 bitig (tehát nem használunk huzalozott eltolást). Ekkor kapjuk meg a P szorzatot, amely a PROD_CLR-L törlőjel segítségével törölhető.

A felső 8 bites párhuzamosan írható / olvasható SHIFT regiszter a „B” szorzóregiszter (a szorzó bemeneteire a biteket *fordított sorrendben* adjuk: B7-B0). A PLIER_CLK-H aktív magas órajel (a szorzóregisztert egy bittel eltolja), míg PLIER_LD-L egyszerre betölti a bemenetre érkező szorzótagokat. A regiszter kimenetén található a PLIER_ZERO-L vonal állandóan ellenőrzi, hogy a bemenetén érkezik-e 0, vagy sem, mert ilyenkor a szorzás egyszerűsíthető, nem kell elvégezni minden egyes bitre. A kimenetén található másik AND_BIT-H jel határozza meg, hogy a szorzó és szorzandó-regiszterek tartalma a szorzatregiszterbe másolható-e, amely az MSB-től függ: ha az AND_BIT-H értéke ’1’, akkor átmásolható. Ellenkező esetben a szorzandó- és szorzóregiszterek tartalmát shiftelni kell egy bitpozícióval.

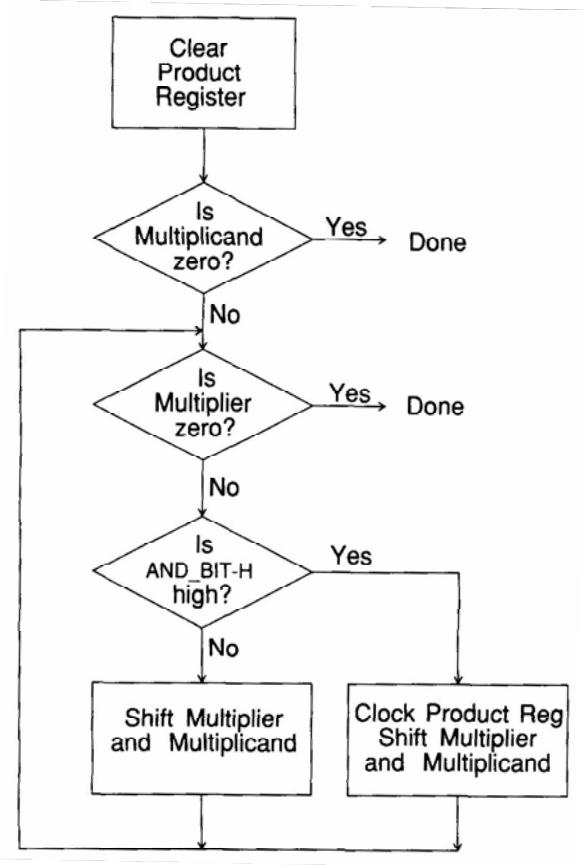


Az középső 16 bites SHIFT regiszter az „A” szorzandó regisztere, amely egy ’195-ös és egy ’164-es SHIFT regiszterből épül fel. A ’195 tölti be a szorzandó értékeket a regiszterbe (PCAND_LD-L), és ugyanekkor törli a ’164 regiszter tartalmát (PCAND_CLR-L -re alacsony jelszint érkezik). Ahogyan a szorzandó „ki-shiftelődik” a 195’-ből (MSB helyiértékek felől), úgy shiftelődik át a 164’esben az LSB helyiértékek felé. A szorzandó regiszterének kimenetén a PCAND_ZERO-L szintén ellenőrzi, hogy 0 érkezett-e a bemenetre.

	A4	A3	A2	A1	A0				
x	B4	B3	B2	B1	B0				
PP0	A4*B4	A3*B4	A2*B4	A1*B4	A0*B4				
PP1		A4*B3	A3*B3	A2*B3	A1*B3	A0*B3			
PP2			A4*B2	A3*B2	A2*B2	A1*B2	A0*B2		
PP3				A4*B1	A3*B1	A2*B1	A1*B1	A0*B1	
PP4					A4*B0	A3*B0	A2*B0	A1*B0	A0*B0
PR			az egyes oszlopok összege						

Ennél a fordított szorzásnál a műveletvégzési idő erősen adatfüggő, mivel ha 0-k érkeznek, akkor a kisebb helyiértékeken már nem kell elvégezni a műveletet, azonnal megkapjuk az eredményt. Azonban kénytelenek vagyunk nagyobb összeadót és „P szorzat regisztert használni. Mindig az adott cél határozza meg, hogy fordított vagy SHIFT-ADD módszert használunk.

Az *adatfolyam diagramm* a szorzási algoritmus pontos végrehajtódását mutatja. Első lépésként a szorzatregiszter törlődik. Majd ellenőrzi hogy a szorzandó zérus-e, ill. hogy a szorzó zérus-e. Az **AND_BIT_H** jel ellenőrzi, hogy az „A” szorzandó regiszter hozzáadható-e a „B” szorzóregiszterhez, és az eredmény betehető-e a „P” szorzatregiszterbe. Ha az MSB=’1’, akkor betehető. Ha nem, akkor a szorzó és szorzandóregiszter 1 bitpozícióval shift-elődik. Továbbá az iterációs lépések számának annyinak kell lennie, ahány bites számaink voltak.



Ebben az esetben az algoritmus időszükséglete is erősen adatfüggő! Továbbá az AND kapuk hiánya (lásd Shift-Add módszer) itt felgyorsítja a működést, igaz ugyan, hogy nagyobb összeadó és nagyobb szorzandó regisztert kell helyette alkalmazni. A szorzás időszükséglete a következő:

$$T_{Mult} = T_{Setup} + N \times T_{Iter}$$

,
ahol

$$T_{Mult} = T_{Setup} + N \times T_{Iter}$$

T(SETUP): kezdeti ellenőrzések, inicializálás ill. szorzat regiszter törlése
T(AND): AND függvények végrehajtása, parciális szorzatok képzése
T(SUM): parciális szorzatok összeadása
T(REG): betöltésük a regiszterbe

c.) Előjeles szorzás Booth algoritmussal:

Nagyon fontos tulajdonsága, hogy *negatív számokkal* is lehet szorzást végezni! Míg az iteratív SHIFT-ADD szorzó és a fordított sorrendű szorzó nem képes kezelni a negatív számokat, addig a Booth algoritmust kifejezetten erre a célra fejlesztették ki. Mindig az MSB bit határozza meg egy 2-es komplemens szám előjelét., ha az MSB='1' akkor a szám negatív, ellenkező esetben pozitív. Példaként 6 bites számot alakítunk át Booth algoritmussal.

$B_{(2'\ komplemens)} = B_5\ B_4\ B_3\ B_2\ B_1\ B_0 = B_5\times(-2^5) + B_4\times2^4 + B_3\times2^3 + B_2\times2^2 + B_1\times2^1 + B_0\times1$.

Előállítjuk a 2' komplemens szám egy új formáját: Újrakódolási technika (séma)

$B_{(2'\ komplemens)} = B_5\times(-32) + B_4\times16 + B_3\times8 + B_2\times4 + B_1\times2 + B_0\times1$ = TRÜKK!!

$= B_5\times(-32) + B_4\times(32-16) + B_3\times(16-8) + B_2\times(8-4) + B_1\times(4-2) + B_0\times(2-1)$ = (azonos numerikus értékek összerendelése)

$= B_5\times(-32) + B_4\times32 - B_4\times16 + B_3\times16 - B_3\times8 + B_2\times8 - B_2\times4 + B_1\times4 - B_1\times2 + B_0\times2 - B_0\times1$=

$= -32\times(B_5 - B_4) - 16\times(B_4 - B_3) - 8\times(B_3 - B_2) - 4\times(B_2 - B_1) - 2\times(B_1 - B_0) - 1\times(B_0 - 0)$.

Ezzel az *átzárójelezéssel* a megfelelő értékeket két bitpár kivonásával kapjuk (ha a zárójeles kifejezés értéke **+**: akkor kivonás,/ ha **0**: akkor áteresztés / ha **–** : akkor pedig összeadás történik). A súlytényezők 2 hatványai, és a szorzást a súlytényezők shiftelésével oldják meg ('165 shift regiszterrel). Egy összeadást mindig egy kivonás követ alternáló jelleggel.

Példa: 543210 (bitpozíciók)

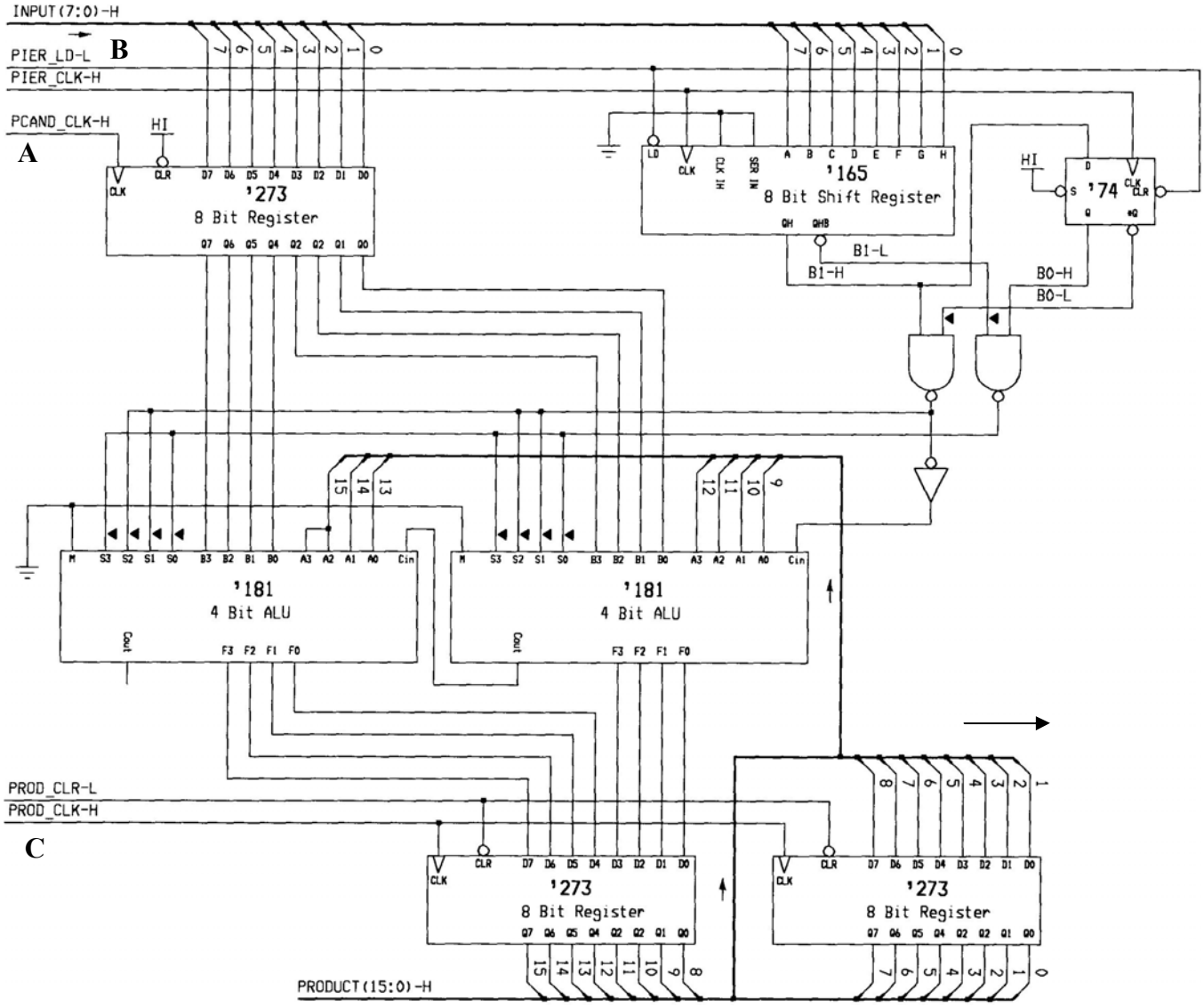
011001= 25 „A"

101101=–19 „B" Végrehajtjuk „**C=A*B" -t!**

Az újrakódolást bitpárokon végezzük el.

$-1\times(B_0-0) = -1$	Co=0–1*A	Mivel – volt az érték, ezért kivonjuk a 0-ból az A-t.
$-2\times(B_1-B_0) = +2$	C1=Co+2*A	Mivel + volt az érték, ezért hozzáadjuk C0-hoz a 2*A-t.
$-4\times(B_2-B_1) = -4$	C2=C1–4*A	
$-8\times(B_3-B_2) = 0$	C3=C2	Mivel 0 volt a kifejezés értéke, ezért Áteresztés történik, nem változik.
$-16\times(B_4-B_3) = +16$	C4=C3+16*A	
$-32\times(B_5-B_4) = -32$	C5=C4–32*A	

A Booth algoritmust megvalósító áramköri modell a következő:

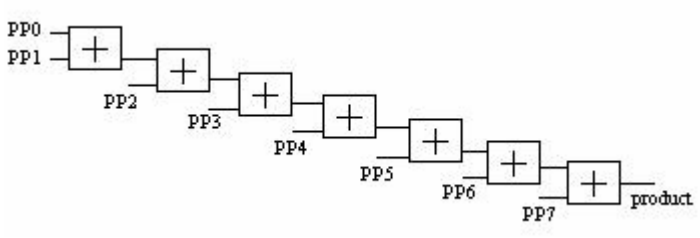


A Booth algoritmust megvalósító szorzó felépítése: két 8 bites számot szeretnénk összeszorozni. A szorzandó „A” ill. az eredményszorzat „C” 8-bites 273’ regiszterek. A szorzó „B” betöltődik a 165’-ös párhuzamosan írható / sorosan olvasható regiszterbe. Ezután a D-tároló 74’ törlődik, hogy a sorozó következő bitjét tárolni tudja. Az összeadás / kivonás / áteresztés műveleteket két 4-bites ALU valósítja meg, a megfelelő S0-S3 jelek választják ki a kívánt műveletet. Az ALU-kból az elvégzett műveleti eredmények egy 273’ regiszterbe töltődnek. Ennek kimenete visszacsatolódik az ALU-k bemeneteire, ill. átkerül a másik mellette lévő 273’ regiszter bemenetére, amely huzalozott eltolással 1-el jobbra tolja a bemenetére érkező biteket.
A ’74 D tároló kimenetéről B0_H ill. a „B” szorzóregiszter kimenetéről B1_H jelek a szorzó shiftelődésének megfelelően generálódnak. Ezek állítják elő megfelelő kombinációs hálózat (2 NAND kapu, és 1 Inverter) segítségével az S0-S3 kiválasztó jeleket az ALU-nál.

II. Közvetlen szorzási módszerek:

Ezeknél a szorzási módszereknél is - hasonlóan az iteratív technikákhoz – először a részlet szorzatokat (partial products, PPi) állítják elő, majd azokat összeadják. A részletszorzatok eltolását egységnyi kapu késleltetéssel oldják meg.

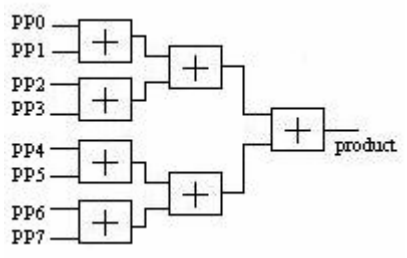
a.) Lineáris modell



Időszükséglet: $T_{(DIRECT-LINE)}=(N-1)xT_{(SUM)}$

A parciális szorzatképzés után azonnal összeadhatók, így gyorsabban megkapjuk az eredményt. N bites számok esetén (N-1) db összeadóra van szükségünk. Lassabb, mint a következő FA modell, mivel több összeadó szintű a késleltetés.

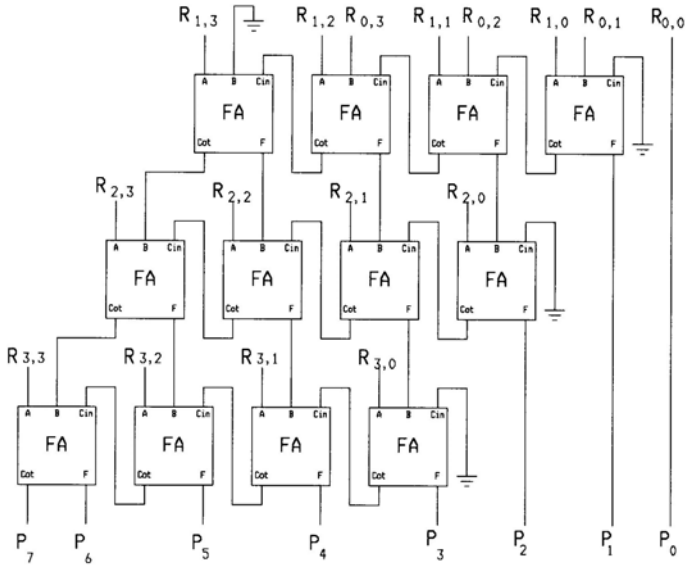
b.) Fa modell



$$T_{DIRECT-TREE} = \lceil \log_2(N) \rceil * T_{SUM}$$

A parciális szorzatok, képzésük után szintén azonnal összeadhatók, gyorsabban megkapjuk az eredményt, mint a lineáris modellnél, mivel (N=8 bit esetén) csak 3 szintű a hierarchia, így kevesebb a késleltetés. N bites számok esetén (N-1) db összeadóra van szükségünk.

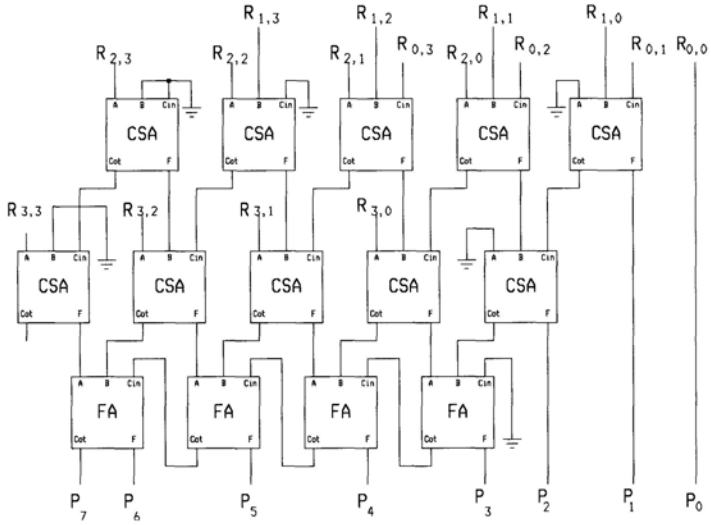
c.) Full Adder felhasználásával részletszorzat képzés:



A mellékelt ábra két 4 bites szám szorzását valósítja meg. A sorokat, mint parciális szorzat tömböket emeljük ki. Jel: $R_{X,Y}$, ahol X: a sor száma, Y: a sor eleme (oszlop).

P=AxB				(A3 (B3	A2 B2	A1 B1	A0) B0)	
				R 0,3	R 0,2	R 0,1	R 0,0	PP0
			R 1,3	R 1,2	R 1,1	R 1,0		PP1
		R 2,3	R 2,2	R 2,1	R 2,0			PP2
	R 3,3	R 3,2	R 3,1	R 3,0				PP3
P7	P6	P5	P4	P3	P2	P1	P0	

d.) CSA=CARRY SAVE ADDER: Olyan Full Adder, amely az előző szint átvitelét (C_{out}) eltárolja, és a következő szint C_{in}-jének továbbítja. Ezzel a módszerrel a szorzás tovább gyorsítható. A késleltetés mindig 2G (2 egységnyi) lesz. A CSA-val történő megvalósítás az előző ábra szerint történik, de az utolsó sorban Full Addereket használunk, míg az első két sorban 5-5 db CSA található. A CSA csökkenti az összeadandó sorok számát (3-2 sorcsökkentő egységnek felel meg).



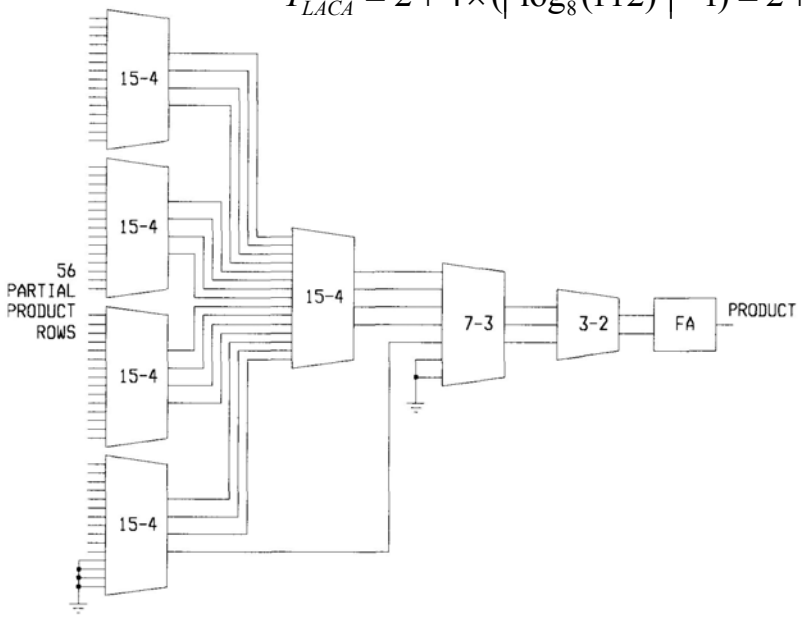
P=AxB				(A3 (B3	A2 B2	A1 B1	A0) B0)	
				R 0,3	R 0,2	R 0,1	R 0,0	PP0
			R 1,3	R 1,2	R 1,1	R 1,0		PP1
		R 2,3	R 2,2	R 2,1	R 2,0			PP2
	R 3,3	R 3,2	R 3,1	R 3,0				PP3
P7	P6	P5	P4	P3	P2	P1	P0	

e.) Sorcsökkentős megvalósítás (Row Reduction):

Két N=56 bites számot szeretnénk összeszorozni ezzel a megoldással (eredmény P= 2*N=112 bites lesz).

Egy k kimenetű sorcsökkentő egység 0 – 2^{k-1} értéket tud reprezentálni, ezért 2^{k-1} bemenetet tud kezelni. Így definiálhatók 31-5, 15-4, 7-3, 3-2 sorcsökkentő egységek. Nagy előnyük, hogy a parciális részletszorzatok (PPi) összeadását párhuzamosan végzik. Így egy N bites bemenetet végül 2 bitesre tudunk redukálni, amely után egy egyszerű Pl. teljes összeadó (FA) vagy LACA összeadó használható. Most 56x56 bites szorzást végzünk: egy megkötésünk, hogy a legnagyobb alkalmazható sorcsökkentő egység 15-4. Az utolsó előtti 3-2 sorcsökkentő egység a carry save adder (CSA), amelyet végül egy LACA (tekintsük egy b=8 bites CP-t propagáló és CG-t generáló LACG egységnek), amelyik a 2*56=112 bites eredményt számolja ki. Így LACA számítási szükséglete a következő:

$$T_{LACA} = 2 + 4 \times (\lceil \log_8(112) \rceil - 1) = 2 + 4 \times (3 - 1) = 10G$$

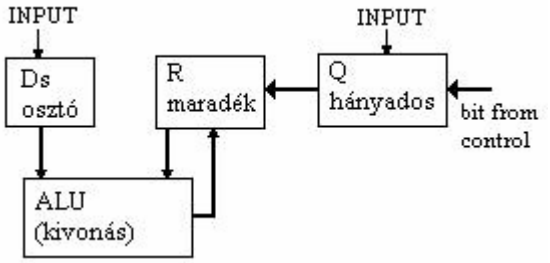


Ha mindenegyes sorcsökkentőnek 2G a késleltetése, amelyet beszorzunk a szintek számával akkor 8G-t kapunk. Ehhez hozzájön, az inputot terhelő 1G késleltetés. Így összesen 9G+10G=19G a kapukésleltetése az eszköznek, ezzel a sorcsökkentős megoldással.

5. Osztó áramkörök:

a.)Hagyományos közvetlen osztási algoritmus:

Kérdés: hányszor van meg egy adott érték (hányados) egy számban (osztandó) ? Az osztót jelöljük Ds-el, az osztandó számot Dd-vel, a hányadost Q-val, míg a maradékot R-el. Az osztást a következő formula szerint végezzük:



Ez az osztási folyamat igen lassú eljárás. **Lépései**:

1. az osztót a Ds regiszterbe rakjuk, az osztandót a Q regiszterbe.
2. töröljük az R regisztert
3. <u>iterációs lépés</u>: kivonjuk az R-ből a Ds osztót. Ha R-Ds>0 akkor folytatódik, tehát ezt a megváltozott értéket visszatesszük az R-be, és egy ’1’-est teszünk a Q regiszterbe. Ha R-Ds<0 akkor R regiszter tartalma nem változik, és egy ’0’-át teszünk a Q regiszterbe (vagy hogyha nincs több osztandó bit, akkor vége az osztásnak).
4. Minden iterációs lépésben egy-egy új bit jön létre, amelyet a Q regiszterbe shiftelünk, ahogyan az R regiszterbe az osztandót
5. Az osztandó legnagyobb helyiértékű (MSB) bitjével kezdjük az összehasonlítást (míg a legkisebbtől a legnagyobb helyiértékek felé, balra haladva shiftelünk a visszaszorzásnál)
6. A hányados generálódik elsőként az MSB felől, és a Q-ba shiftelődik 1 bittel balra
7. A folyamat végén a maradék Az R-ben, a hányados pedig a Q-ban lesz

Dd=Q*Ds + R

Egy kikötésünk van: R<Ds esetén leáll az osztás!

Decimális számok esetén:

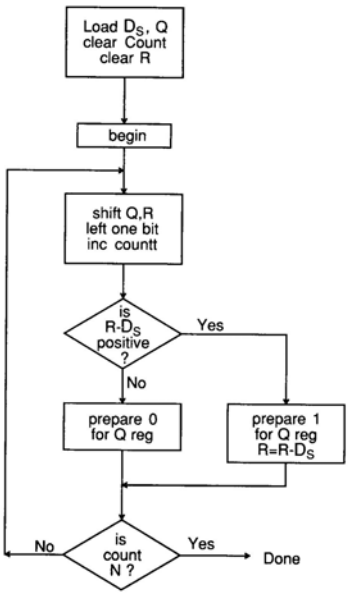
5|8 : 5=1|1
0 8
3

PÉLDA: Bináris számok esetén hasonlóan

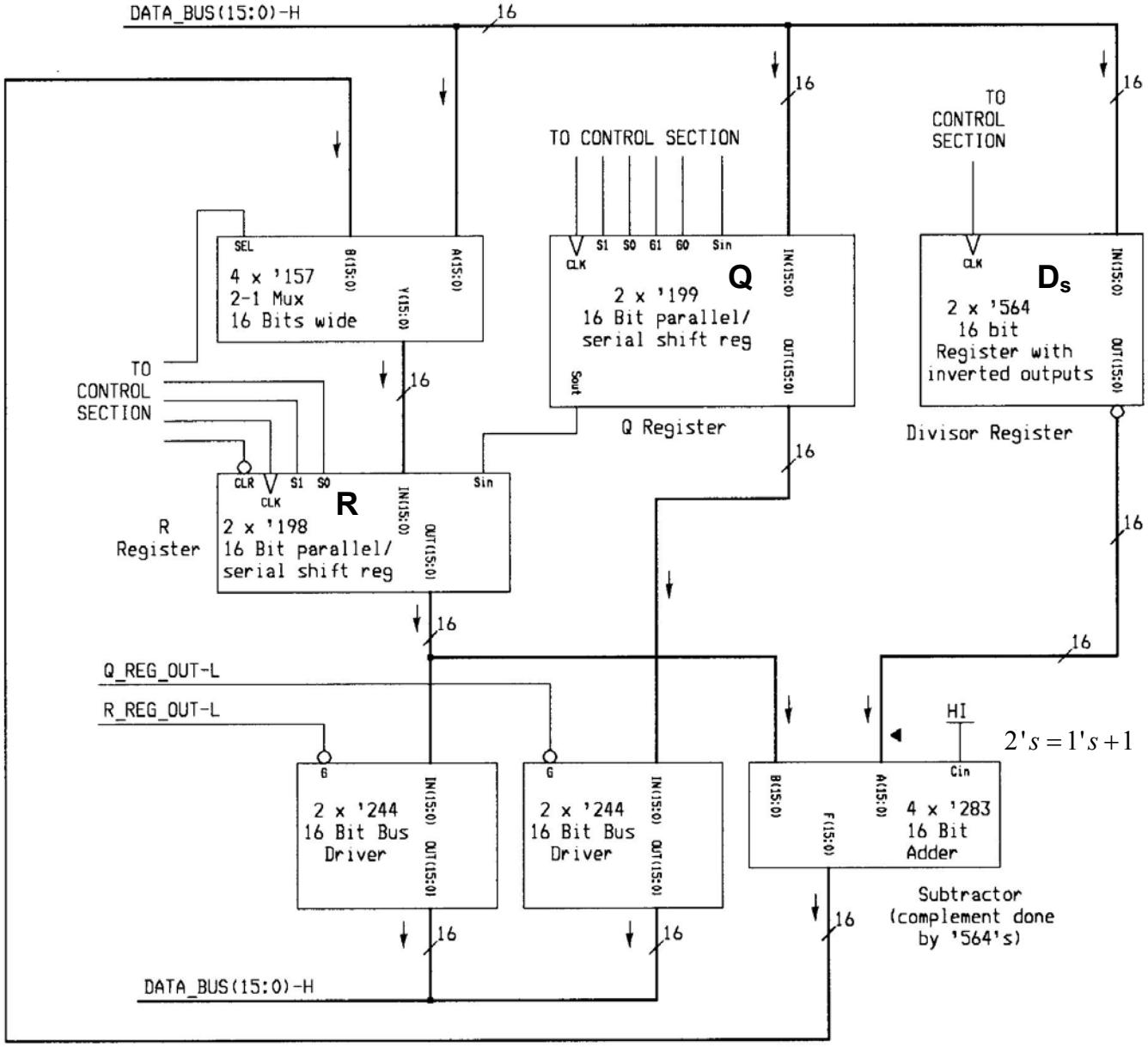
1011	Q hányados (Ds hányszor van meg Dd-ben)
111010	Dd osztandó
101/	Ds osztó
111|010	111-ben megvan „101” ezért 1→Q
101	Visszaszorzás 1*”101”-el
10	Ez a kivonás eredménye 111-101=10
100|00	100-ban nincs meg az ’101’, ezért 0→Q
000	Visszaszorzás 0*’101’-el
100	Ez a kivonás eredménye 100-000=100
1001|0	1001-ban megvan ’101’, ezért 1→Q
101	Visszaszorzás 1*’101’-el
100	Ez a kivonás eredménye: 1001-101=100
1000|	1000-ban megvan az ’101’, ezért 1→Q
101	Visszaszorzás 1*’101-el
11	Kivonás eredménye: 1000-101=11
	Ez a maradék R!

Az osztó áramkör felépítése a következő: 16 bites buszon keresztül érkeznek az adatok. Három regiszterből áll: Ds, Q, és R. A Ds osztóregiszter két 16 bites 564’ –es invertált kimenettel rendelkező regiszterből áll, amely 2-es komplemens kivonást hajt végre (R-ből). Az R maradék regiszter is két 198’-as 16 bites párhuzamosan írható / sorosan olvasható SHIFT regiszterből áll (tölthető és törölhető is). Az R maradék regisztert a buszról érkező osztandó (Dd) értékével írjuk felül, a 4 db 2-1 MUX segítségével. Szintén két 198’-as 16 bites paralel írható / sorosan olvasható SHIFT regisztert használunk a Q hányados értékének tárolásához. Az „R−Ds” kivonáshoz 4 db 283’as 16 bites 2-es komplemens összeadót használunk, amelynek egyik bemenetét közvetlenül az R-regiszter képezi, míg a másikat a Ds regiszter negálásával kapjuk. Ha a Cin-H jel magas-aktív, akkor inkrementálódik 1-el. (így érjük el a kivonást bitenkénti negálással és inkrementálással). Végül a kivonás eredménye az R regiszterbe töltődik vissza MUX-okon keresztül.

Osztási algoritmus **folyamatábrája** a következő:



Az alábbi ábrán a hagyományos módszerű osztó áramköri modellje látható:



b.) Iteratív osztási műveletek I.: gyors osztás Newton Raphson módszerrel

Az előzőnél gyorsabb osztási művelet reciprokképzéssel valósul meg. Szorzó segítségével végezzük el az osztást. A Newton-Raphson iteráció alapformulája a következő:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Van egy megfelelő f függvényünk és egy x0 kezdeti értékünk. Iterációs lépésekkel megkapjuk az osztás eredményét az f(x)=0 egyenlet megoldásaként. Az f-et úgy kell (jól) megválasztanunk, hogy a reciprok gyökkel rendelkezzen. Legyen

$$f(x) = \frac{1}{x} - w$$

Az fenti egyenlet gyöke, f(x)=0 esetén az x= 1/w. Ha f(x)=1/x-w, akkor

$$f'(x) = -\frac{1}{x^2}$$

Ekkor visszahelyettesítve az eredeti Newton-Raphson iterációs képletbe a következőt kapjuk:

$$x_{i+1} = x_i - \frac{\frac{1}{x_i} - w}{-\frac{1}{x_i^2}} = x_i + (x_i - wx_i^2) = 2x_i - wx_i^2 = x_i(2 - wx_i)$$

Tehát az A/B műveletet A*(1/B) alakra írtuk át, és az 1/B reciprokképzést egy szorzóval és egy kivonóval valósíthatjuk meg. A függvény Taylor sorának kiterjesztésével (négyzetes konvergencia) belátható, hogy minden egyes iterációs lépésben a helyes bitek száma megduplázódik. Tehát megfelelő iterációs lépés kiválasztásával a kívánt pontosság elérhető!

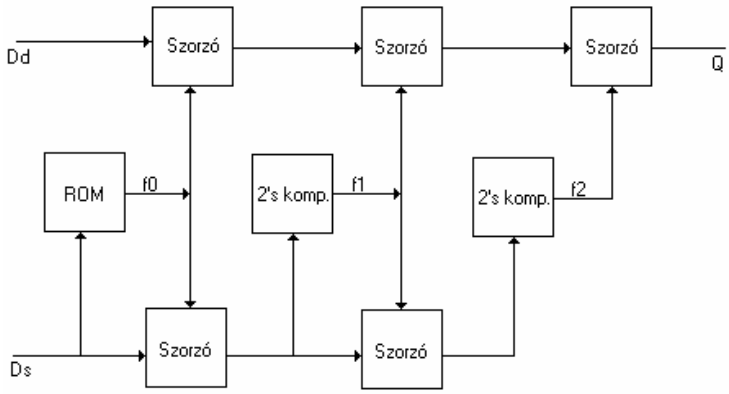
c.) Iteratív osztási műveletek II.: közvetlen gyors osztó

Az iteratív osztási művelet másik módszere a következő: $Q=D_D/D_S$ kiszámolható a következő egyenlettel, ha a successive (egymást követő) f_k –k úgy vannak megválasztva, hogy a nevező az 1-hez konvergáljon.

$$Q = \frac{D_D \times f_0 \times f_1 \times f_2 \ldots}{D_S \times f_0 \times f_1 \times f_2 \ldots}$$

A számláló iterációja ennél a módszernél $D_{Dn+1} = D_{Dn} \times f_n$. A nevező iterációját a megismert módon használjuk a következő f-ek meghatározására. Tegyük fel, hogy szorzóink, 2'komplemens képző egységeink vannak, valamint a kezdő értéket tartalmazó ROM, vagy bármilyen regiszter áll rendelkezésünkre.

Ezzel az iteratív osztási módszerrel az eredményt *közvetlenül* megkapjuk. Feltételezzük, hogy a számok itt normalizált lebegőpontos számok, az osztót és osztandót egy törtkifejezésként írjuk fel (mantissza egy normalizált tört). Keressük a Q hányados (quotient) értékét. Hogy megkapjuk, mind az osztó, mind pedig az osztandó értékét ugyanazokkal az $\mathbf{f_K}$ értékekkel kell megszorozni, amelyet úgy határozunk meg, hogy a nevező egységnyi legyen az iterációk elvégzése után. Így később a számláló értékéből megkapjuk a Q pontos értékét. Tudjuk, hogy D_S normalizált tört, ezért így ábrázoljuk: $D_S = 1-x$, ahol x-et D_S határozza meg, és mivel D_S kisebb 1-nél, így az x is kisebb 1-nél.



Az osztás művelete fo kiszámolásával kezdődik. Válasszuk **fo** = 1+x = 1+(1–D_S) = **2–D_S** . Így $D_S \times fo = (1-x)(1+x) = 1-x^2$. Így sokkal közelebb kerültünk 1-hez, mintha csak a D_S –et használtuk volna. Minden iterációs lépésben a számláló és a nevező is f_K tényezőkkel szorzódik, és közelebb kerülünk a Q pontos értékéhez. Legyen $f_1 = 1+x^2$. Így $D_S \times fo \times f_1 = 1- x^4$ és ez tovább ismételhető iteratív módon.

Tehát azt kapjuk, hogy $\mathbf{D_{Dn+1} = D_{Dn} \times fn.}$

Egy kérdés vetődik fel: hogyan válasszuk meg f_k következő értékét. $\mathbf{f_1} = 1+x^2 = 1+(1-D_S \times fo) =$ **2– $\mathbf{D_S \times fo.}$** Tehát minden egyes új f_k –t úgy kapunk meg, hogy vesszük az f_{k-1} és a Ds (nevező) szorzatának 2's komplemensét. Az iterációs lépéseket a kívánt pontosság eléréséig kell ismételni, amelyet f_k értéke határoz meg. Amikor f_k *közelítőleg 1*, akkor a Q eredmény

elegendően közel lesz a kívánt eredményhez (amely az alkalmazástól és a bitek számától függ). Általában előre definiált fix számú iterációs lépést végzünk el. Ezért kell ROM-ot használni, amelyben az fo megfelelő kezdeti értékét tároljuk. Vegyünk két egyszerű példát:

Példák:

(1) Legyen az osztandó $D_D = 0.4$, osztó $D_S = 0.7$, és 6 iterációs lépésig számoljunk (7 tizedesjegy pontosságú számokkal). Ekkor fo=2– D_S =2–0.7=1,3000000. Kérdés Q= D_D/D_S? /$D_{Dn+1} = D_{Dn} \times$ fn./

0.	D_{DO}=0,4000000	D_{SO}=0,7000000	fo=1,3000000
1.	D_{D1}=0,5200000	D_{S1}=0,9099999	f1=1,0900000
2.	D_{D2}=0,5668000	D_{S2}=0,9918999	f2=1,0081000
3.	D_{D3}=0,5713911	D_{S3}=0,9999344	f3=1,0000656
4.	D_{D4}=0,5714286	D_{S4}=0,9999999	f4=1,0000000
5.	D_{D5}=0,5714286	D_{S5}=1,0000000	f5=1,0000000
6.	D_{D6}=0,5714286	D_{S6}=1,0000000	

Látható, hogy már a 4. Iterációs lépésben megkaptuk a helyes eredményt (D_{D4}=0,5714286), mivel Ds elég közel volt az 1-hez, és x=0.3 volt. (x=1-Ds).

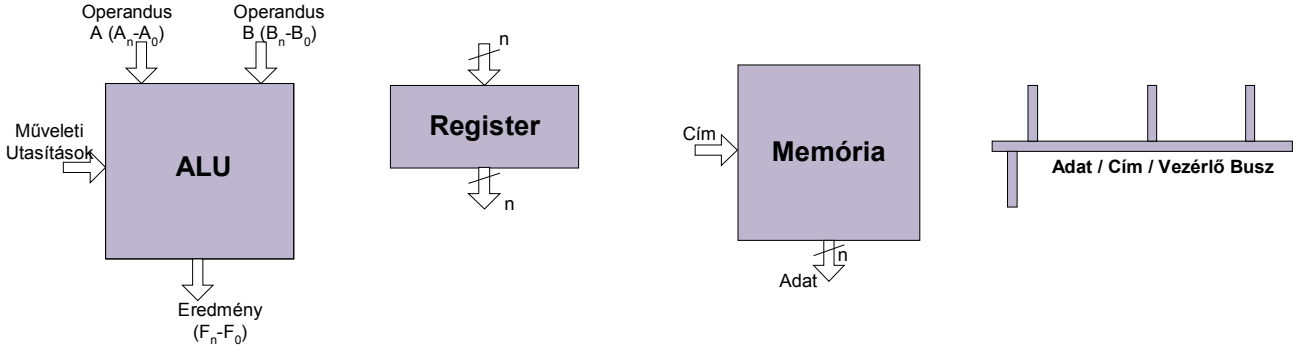
(2) Legyen az osztandó $D_D = 0.1$, osztó $D_S = 0.15$, és 6 iterációs lépésig számoljunk (7 tizedesjegy pontosságú számokkal). Ekkor fo=2– D_S =2–0.15≈1,8499999. Kérdés Q= D_D/D_S? /$D_{Dn+1} = D_{Dn} \times$ fn./

0.	D_{DO}=0,1000000	D_{SO}=0,1500000	fo=1,8499999
1.	D_{D1}=0,1850000	D_{S1}=0,2775000	f1=1,7224999
2.	D_{D2}=0,3186625	D_{S2}=0,4779938	f2=1,5220062
3.	D_{D3}=0,4850063	D_{S3}=0,7275094	f3=1,2724905
4.	D_{D4}=0,6171659	D_{S4}=0,9257489	f4=1,0742511
5.	D_{D5}=0,6629912	D_{S5}=0,9944868	f5=1,0055132
6.	D_{D6}=0,6666464	D_{S6}=0,9999696	....

Látható, hogy itt nem kapjuk meg a kívánt értéket (D_{D6}=0,6666464) 6 iterációs lépés alatt. Ezért hogy elérjük a kívánt pontosságot véges számú lépés alatt, ROM-ot kell használni (ahol fo kezdeti értékét tároljuk).

6. Digitális építőelemek (regiszter, ALU, MUX, kódolók):

(ALU: Lásd bővebben még 2. tétel)
Már a korai számítógépeknél is léteztek a következő építőelemek: regiszterek, ALU-k, memóriák, adatbuszok. Az ***ALU*** két különböző input résszel rendelkezik, és egy kimeneti vonallal. A szelektáló jelek segítenek a megfelelő műveletek kiválasztásában. Az ALU egység egy algoritmus utasításainak megfelelően aritmetikai ill. logikai műveleteket hajt végre.



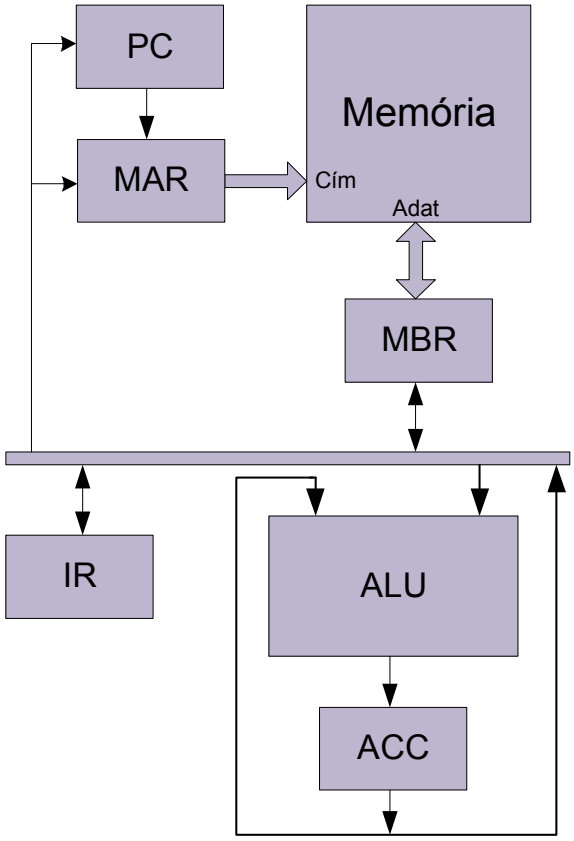
Az ALU által kezelt adatok a ***memóriában*** (tároló rekeszek lineáris tömbje) tárolódnak el. A memória rekeszei általában olyan szélesek, amilyen széles az adatbusz. Például, legyen n-bit széles egy rekesze, és álljon M számú rekeszből. Ekkor $\log_2(M)$ számú címvezetékkel címezhető meg a memória. Az adatbuszon kétirányú (írás/olvasás) kommunikáció is megengedett. Memória *Neumann* architektúrát követ: tehát az utasítások (program) és az adatok egy helyen tárolódnak, nem pedig külön-külön (Harward architektúra). A programot is adatként tárolja a memória.

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

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

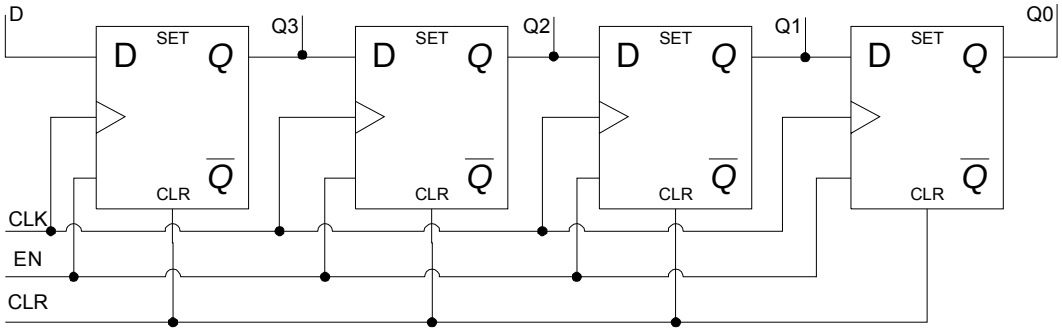
Egyszerű számítógép blokkdiagramja (Egycímű gép), a benne lévő komponensek együttes működésének leírásával.

- *MAR: Memory Address Register (Memória-cím Regiszter)*: a memóriából érkező, vagy a memóriában lévő információ helyét azonosítja adott memóriacím alapján.
- *MBR: Memory Buffer Register (Memória Puffer Regiszter)*: tárolja a memóriába bevitt, ill. az onnan érkező információt. Egy adott memóriacímen lévő adat kiolvasásakor az ott lévő bejegyzés törlődik (destruktív memória), és ahhoz hogy ezt megőrizzük vissza kell írni a bejegyzést. (Ez csak a korai ferritgyűrűs memória esetén igaz). Ezt a bejegyzést őrzi meg az MBR, amelyet aztán az áramkör többi része is elérhet. Félvezető memóriák használata esetén nem szükséges MBR-t alkalmazni.
- *PC: Program Counter (Programszámláló)*: a soron következő (végrehajtandó) utasítás helyét azonosítja. Azon gépeknél, amelyek egy utasítást tárolnak memóriaterületenként, az utasítás végrehajtása után a PC értékét 1-el kell növelni, tehát úgy működik, mint egy számláló. Ezért is kapta a nevét.
- *IR: Instruction Register (Utasítás Regiszter)*: tárolja az éppen végrehajtás alatt álló utasítást. Engedélyezi a gép vezérlő részeinek, hogy a regiszterek, memóriák, aritmetikai egységek vezérlő vonalait a végrehajtáshoz szükséges működési módba állítsák. Az IR olyan széles, hogy az utasítás műveleti kódja ill. a hozzá tartozó egyéb utasítások ideiglenes másolatai eltárolhatók legyenek.
- *ACC: Accumulator (tárolóregiszter):* eredmény ideiglenes tárolására használjuk (összes adatkezeléshez tartozó utasítás tárolása).

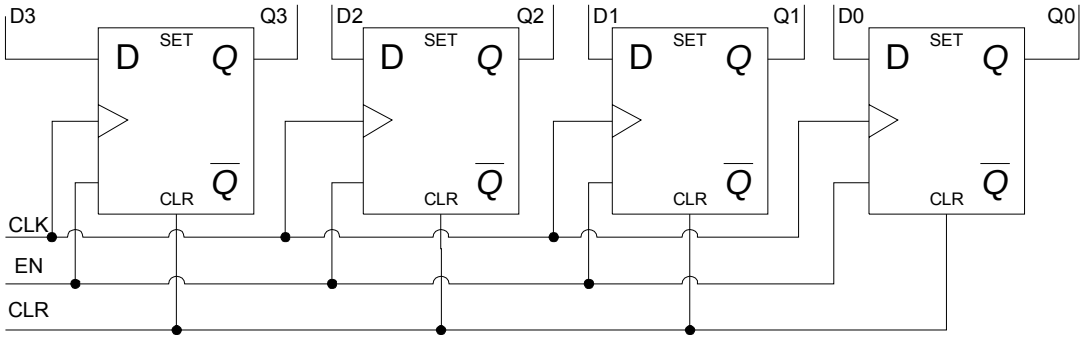


Regiszterek: Általánosan használt regiszterek a következők: (legyen n=4 bit)

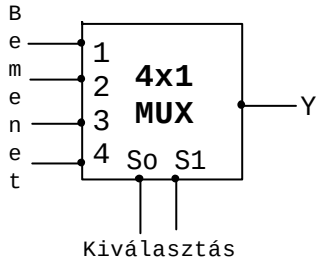
a.) 4-bites Serial In/Parallel Out SHIFT/léptető - regiszter D tárolókból felépítve:



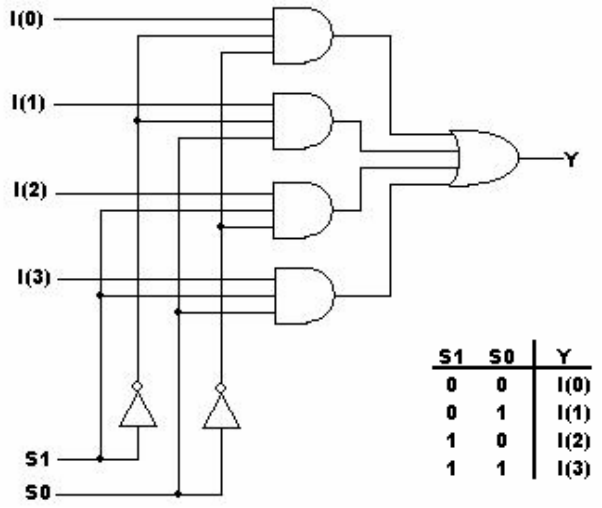
b.) 4-bites Parallel In/ Parallel Out regiszter D tárolókból felépítve:



4–1 Multiplexer:

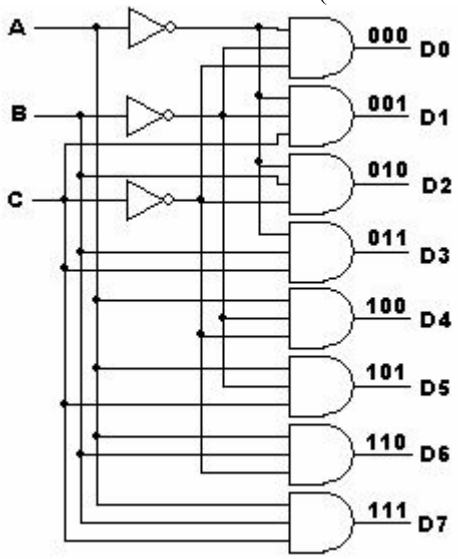


N kiválasztó jel -> 2^N számú bemenet közül választ egyet -> kimenet (Y)
Működése: egyszerű kapcsoló áramkör (switch)

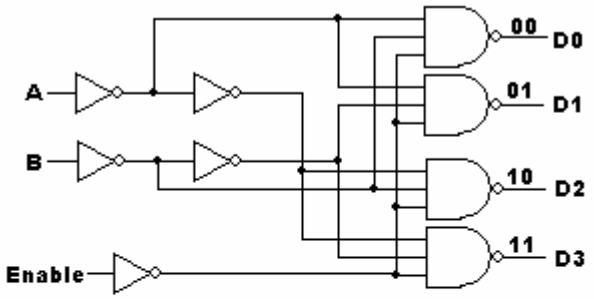


S1	S0	Y
0	0	I(0)
0	1	I(1)
1	0	I(2)
1	1	I(3)

Dekódoló áramkörök: 3 bemenet esetén 2^3=8 kimenet van. Jele: 3x8 (N bemenet esetén 2^N kimenete van!)



Dekódoló 2x4 – Engedélyező (alacsony-aktív) bemenettel:



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

7. Utasítás kódok:

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

Egy utasítás végrehajtásának három fő lépését a **Fetch-Decode-Execute** (FDE) mechanizmussal definiálhatjuk:

1. Fetch: az utasítás betöltődik a memóriából az utasításregiszterbe (regiszter-transzfer művelet)
2. Decode: utasítás dekódolása (értelmezése), azonosítja az utasítást
3. Execute: a dekódolt utasítást végrehajtjuk az adatokon, aminek eredménye visszakerül a memóriába

Egy utasítás végrehajtása az *RTL leírás (Regiszter-Transzfer Nyelvvel*) segítségével írható le. A szükséges adatátviteleket ezzel a nyelvvel specifikáljuk az egyik fő komponenstől a másikig. Továbbá megadható az engedélyezett adatátvitelekhez tartozó blokkdiagram is, az éleken adott irányítással, amelyek az adatátvitel pontos irányát jelölik. Az RTL leírások specifikálják az akciók pontos sorrendjét. Az egyes utasításokhoz megadhatók a szükséges végrehajtási idők (ns-ban), amelyek erősen függenek a felhasznált technológia tulajdonságaitól. Ezek összege megadja a teljes tranzakció időszükségletét.

Néhány alapvető akció specifikációja a következő:

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

Utasítás formák:

- **Zéró-című (0 című):** STACK, vagy verem (PUSH, POP, ACC) [operátor]
- **1-című:** [operátor], [operandus] (Példa: Egyszerű számítógép blokkdiagramja – Lásd még 6. tétel!)
- **2-című:** [operátor], [operandus1], [operandus2]
- **3-című:** [operátor], [operandus1], [operandus2], [eredmény]
- **4-című:** [operátor], [operandus1], [operandus2], [eredmény], [következő utasítás] …

a.) 1-Című gép v. Egyszerű számítógép (pl. PDP-8, ISA) A műveletekhez csak 1 operandus szükséges. Ilyen művelet lehet például: komplemens képzés, inkrementálás, törlés, keresés az ACC-ben (akkumulátor). Az eredmény az ACC-ben tárolódik. (ACC egy olyan speciális regiszter, amelyben az aritmetikai és logikai műveletek eredménye ideiglenesen tárolódik.) Két operandus esetén az első operandus ACC-ben tárolt értékét használjuk fel, és a másik operandust egyetlen címmel azonosítjuk.

- Pl: ***Negálás/1's komplemens* képzés:** RTL leírása

Fetch: (regiszterek feltöltése, utasításhívások):

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

Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute:(végrehajtás)

complement ACC	→ ACC	ACC komplemensét az ACC-be töltjük
ACC+1	→ ACC	ACC-t 1-el inkrementáljuk

- Pl: ***Kivonás* egy operandus esetén** (SUB X= kivonjuk az X helyen tárolt értéket az ACC-ből, és az ACC-be visszatöltjük).

Fetch: (regiszterek feltöltése, utasításhívások):

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

Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute:(végrehajtás)

X	→ MAR	X címét a MAR-ba töltjük
M[MAR]	→ MBR	X címen lévő értéket az MBR-be tesszük
ACC – MBR	→ ACC	ACC-ből kivonjuk az X-et, és ACC-be rakjuk

b.) 2- és több-című gépek (regiszter nélküli eset):

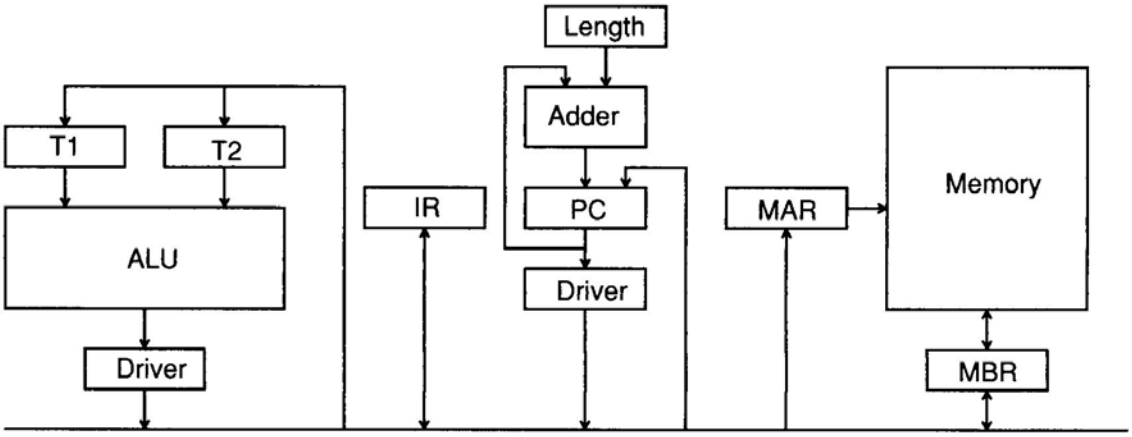
- [operátor], [operandus1], [operandus2]
- [operátor], [operandus1], [operandus2], [eredmény]…

Mivel itt egy utasítással több műveletet lehet megadni, kevesebb utasítás-sorral, de összetett módon írhatók le az RTL nyelv segítségével az egyes folyamatok, (az egycímű gépekkel ellentétben). A többcímű utasítások meghatározzák, mind a forrás, mind pedig a célinformációt. A célinformáció helyét az utolsó operandus címe adja meg!

Jelölés: ***ADD2 X, Y*** kétcímű utasítás (műv, op1, op2). Az X cím által azonosított helyen tárolt értéket hozzáadjuk az Y cím által azonosított helyen lévő értékhez, és az összeadás eredményét az Y címmel azonosított helyen tároljuk el.

Jelölés: ***ADD3 X,Y,Z*** háromcímű utasítás (műv, op1, op2, eredmény): hasonló az előzőhöz, csak az összeadás eredménye egy új helyen, a Z cím által azonosított helyen tárolódik el.

Fontos megjegyezni hogy itt a T1, ill. T2 regiszter nem az utasítás-készlet architektúra része! (ezért nem keverendő össze a később említésre kerülő regiszteres címzéssel!) Ebben az esetben csak az adatok tárolásához használjuk, nem pedig a rendszer gyorsítását kívánjuk alkalmazásukkal elérni.



Példa 1 – Kétcímű gép:

ADD2 X, Y (RTL idők: Memória idő=200ns, Regiszter idő=30ns, Összeadási idő=70ns. ADD2 X, Y direkt címzéses (direct addressing) összeadás RTL-leírása, időzítése)

Fetch: (regiszterek feltöltése, utasításhívások):

PC	→ MAR	(30ns)	Elsőként a PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]	→ MBR	(200ns)	Memóriában lévő utasítás beírása az MBR-be (később visszaírjuk)
PC+I_len	→ PC	(30ns)	Az utasítás hosszával (I_len) növeli a PC értékét
MBR	→ IR	(30ns)	Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute:(végrehajtás)

PC	→ MAR	(30ns)	MAR-ben eltároljuk **X** címét
PC+XA_len	→ PC	(30ns)	Megnöveljük a PC értékét az X címének hosszával
M[MAR]	→ MBR	(200ns)	MBR beírjuk az X címét
MBR	→ MAR	(30ns)	Majd betesszük a MAR-ba
M[MAR]	→ MBR	(200ns)	Kinyerjük az X értékét
MBR	→ T1	(30ns)	X operandust a T1 regiszterbe töltjük
PC	→ MAR	(30ns)	MAR-ben eltároljuk **Y** címét
PC+YA_len	→ PC	(30ns)	Megnöveljük a PC értékét az Y címének hosszával
M[MAR]	→ MBR	(200ns)	Beírjuk az Y címét az MBR-be
MBR	→ MAR	(30ns)	Majd betesszük a MAR-ba
M[MAR]	→ MBR	(200ns)	Kinyerjük az Y értékét
MBR	→ T2	(30ns)	Y–t a T2 regiszterbe töltjük
T1 + T2	→ MBR	(70+30ns)	Elvégezzük az összeadást (két regiszter tartalmát MBR-be írjuk)
MBR	→ M[MAR]	(200ns)	Eredményt a MAR-ban tároljuk el (ahol Y volt)
		(∑ 1630ns)	A teljes (F-D-E) idő.

Példa 2 – Háromcímű gép:

ADD3 X, Y, Z (RTL idők: Memória idő=200ns, Regiszter idő=30ns, Összeadási idő=70ns. ADD3 X, Y, Z direkt címzéses (direct addressing) összeadás RTL-leírása, időzítése)

Fetch: (regiszterek feltöltése, utasításhívások):

PC	→ MAR	(30ns)	Elsőként a PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]	→ MBR	(200ns)	Memóriában lévő utasítás beírása az MBR-be (később visszaírjuk)
PC+I_len	→ PC	(30ns)	Az utasítás hosszával (I_len) növeli a PC értékét
MBR	→ IR	(30ns)	Majd az MBR-ben lévő adatot az IR-be tesszük

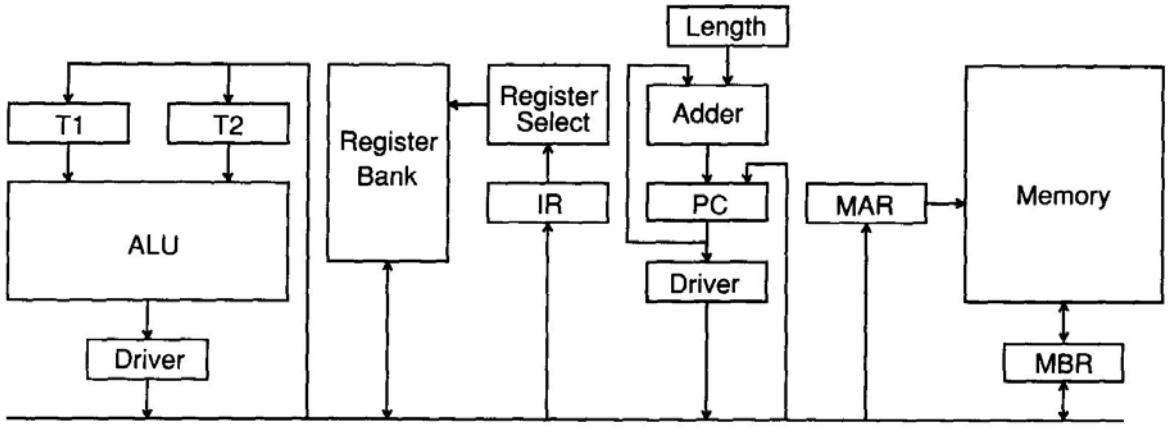
Decode: (a dekódolást általában 0 idejűnek feltételezzük)
Execute:(végrehajtás)

PC	→ MAR	(30ns)	MAR-ban eltároljuk **X** címét
PC+XA_len	→ PC	(30ns)	Megnöveljük a PC értékét az X címének hosszával
M[MAR]	→ MBR	(200ns)	MBR beírjuk az X címét
MBR	→ MAR	(30ns)	Majd betesszük a MAR-ba
M[MAR]	→ MBR	(200ns)	Kinyerjük az X értékét
MBR	→ T1	(30)	X operandust a T1 regiszterbe töltjük
PC	→ MAR	(30ns)	MAR-ban eltároljuk **Y** címét
PC+YA_len	→ PC	(30ns)	Megnöveljük a PC értékét az Y címének hosszával
M[MAR]	→ MBR	(200ns)	Beírjuk az Y címét az MBR-be
MBR	→ MAR	(30ns)	Majd betesszük a MAR-ba
M[MAR]	→ MBR	(200ns)	Kinyerjük az Y értékét
MBR	→ T2	(30)	Y–t a T2 regiszterbe töltjük
PC	→ MAR	(30ns)	MAR-ban eltároljuk **Z** címét
PC+ZA_len	→ PC	(30ns)	Megnöveljük a PC értékét a Z címének hosszával
M[MAR]	→ MBR	(200ns)	Beírjuk az Z címét az MBR-be
MBR	→ MAR	(30ns)	Majd betesszük a MAR-ba
T1 + T2	→ MBR	(70ns)	Elvégezzük az összeadást (két regiszter tartalmát MBR-be írjuk)
MBR	→ M[MAR]	(200ns)	Eredményt a MAR-ban tároljuk el (ahol Z volt)
		(∑ 1690ns)	A teljes (F-D-E) idő.

Az **ADD3** több időt vesz igénybe, mint az ADD2, mivel egy címmel nőtt a végrehajtás. Azonban az ADD3 jelentősége a komplexebb műveletek elvégzésekor mutatkozik meg:
Legyen X=Y*Z+W*V

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

c.) 2- és többcímű gépek (regiszteres címzési mód):
A regiszterek használata csökkenti a végrehajtási időt, mivel a lassú memória-intenzív műveletek helyett gyorsabb regiszterműveleteket használnak. (A regiszterbank 8 regisztert tartalmazhat.)



ADD2 R_X ,R_Y (RTL idők: Memória idő=200ns, Regiszter idő=30ns, Összeadási idő=70ns. ADD2 R_X, R_Y direkt regisztercímzéses (direct register addressing) összeadás RTL-leírása, időzítése.)
Fetch: (regiszterek feltöltése, utasításhívások):

PC	→ MAR	(30ns)	Elsőként, a PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]	→ MBR	(200ns)	Memóriában lévő utasítás beírása az MBR-be (később visszaírjuk)
PC+I_len	→ PC	(30ns)	Az utasítás hosszával (I_len) növeli a PC értékét
MBR	→ IR	(30ns)	Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (a dekódolást általában 0 idejűnek feltételezzük)
Execute:(végrehajtás)

R_X	→ T1	(30ns)	T1-ben közvetlenül eltároljuk **R_X** operandust
R_Y	→ T2	(30ns)	T2-ben pedig eltároljuk **R_Y** operandust
T1 + T2	→ R_Y	(70ns)	Elvégezzük az összeadást (két regiszter tartalmát R_Y helyére írjuk)
		(∑ 420ns)	A teljes (F-D-E) idő.

Ennél a címzési módnál csak egyszer kell a memóriára hivatkozni, mégpedig a regiszterek feltöltésének fázisában. A regiszterből közvetlenül kinyerhető az operandus címe, később már nem kell a végrehajtás fázisában a memóriát használni. Így a címzési módok közül ez a leggyorsabb. A regiszterek alkalmazásával nő az algoritmus végrehajtásának sebessége. A megfelelő regisztert egy MUX választja ki, az IR utasításregiszterben tárolt bitnek megfelelően.

ADD3 R_X ,R_Y, R_Z (RTL idők: Memória idő=200ns, Regiszter idő=30ns, Összeadási idő=70ns. ADD3 R_X, R_Y, R_Z direkt regisztercímzéses (direct register addressing) összeadás RTL-leírása, időzítése.)

Fetch: (regiszterek feltöltése, utasításhívások):

PC	→ MAR	(30ns)	Elsőként, a PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]	→ MBR	(200ns)	Memóriában lévő utasítás beírása az MBR-be (később visszaírjuk)
PC+I_len	→ PC	(30ns)	Az utasítás hosszával (I_len) növeli a PC értékét
MBR	→ IR	(30ns)	Majd az MBR-ben lévő adatot az IR-be tesszük

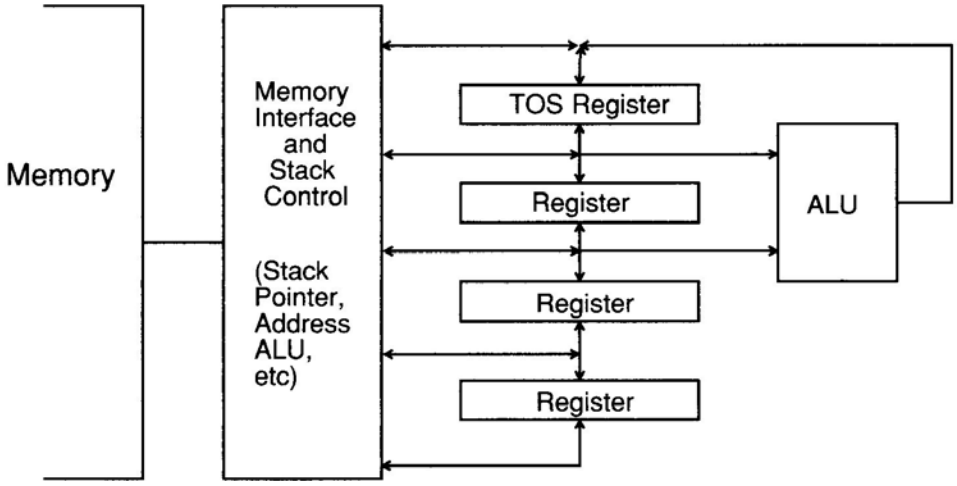
Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute:(végrehajtás)

R_X	→ T1	(30ns)	T1-ben közvetlenül eltároljuk $\mathbf{R_X}$ operandust
R_Y	→ T2	(30ns)	T2-ben pedig eltároljuk $\mathbf{R_Y}$ operandust
T1 + T2	→ R_Z	(70ns)	Elvégezzük az összeadást (két regiszter tartalmát $\mathbf{R_Z}$ helyére írjuk)
		(∑ 420ns)	A teljes (F-D-E) idő.

d.) 0-című vagy Zéró-című gépek:

Ebben az esetben egy vermet (stack) használunk: LIFO- típusú tároló, amelyből az utoljára betett adatot vesszük ki elsőként. A stack a memóriában található egy elkülönített részen. Különböző aritmetikai kifejezések hajthatók végre elég hatékonyan stack használatával: a szükséges operanduskat a stack egy-egy rekeszében tároljuk. A megfelelő operandusokat (felső kettő) kivesszük, elvégezzük rajtuk a műveleteket, és az eredményt a verem tetejére tesszük. Azért nevezzük zéró címűnek, mivel az operandusok azonosítására szolgáló utasításhoz nem használunk címeket. Az ábra a HW orientált stack rendszert ábrázolja.



*F=A+[B*C + D*(E / F)]* az aritmetikai kifejezés, F-et akarjuk kiszámolni verem segítségével és eltárolni az eredményt.

A következő műveletek szükségesek a végrehajtáshoz: **PUSH, POP**, ADD, DIVIDE, MULTIPLY

1. módszer: (arit. kif. elejétől haladva)

```
PUSH A
PUSH B
PUSH C
MULT [B*C]
PUSH D
PUSH E
PUSH F
DIV [E/F]
MULT [D*(E/F)]
ADD [B*C+D*(E/F)]
ADD [A+(B*C+D*(E/F))]
POP F
```

2. módszer: (arit. kif. végétől visszafelé haladva)

```
PUSH E
PUSH F
DIV [E/F]
PUSH D
MULT [D*(E/F)]
PUSH C
PUSH B
MULT [B*C]
ADD [B*C+D*(E/F)]
PUSH A
ADD [A+(B*C+D*(E/F))]
POP F
```

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

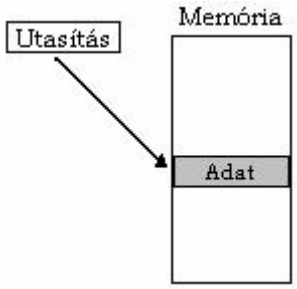
Az 1. módszer kiértékelésénél az aritmetikai kifejezés elejétől haladunk, és amint lehetséges a verem tetején lévő két értéken végrehajtjuk a soron következő műveletet, az eredményt, pedig a verem tetejére pakoljuk. A veremben max. 5 értéket tárolunk el, (mélysége 5 lesz) ezért lassabb, mint a második módszer.

A 2. módszernél az aritmetikai kifejezést hátulról előrefelé haladva értékeljük ki. Itt is elvégezzük a soron következő műveletet, és az eredményt a verem tetejére rakjuk. De ez gyorsabb módszer, mivel a veremben max. csak 3 értéket tárolunk el.

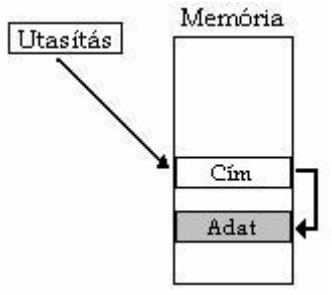
8. Címzési módok: operandus címzési módok

Utasítás végrehajtásakor a kívánt operandust el szeretnénk érni, a címmel hivatkozhatunk a pontos helyére, ill. azonosítjuk őt. Többféle címzési mód is létezik: közvetlen (direct), közvetett (indirect), indexelt (indexed), regiszteres megvalósítású (register relative). Ezek kombinációja igen sokféle, összesen 10-féle azonosítási módot tesz lehetővé.

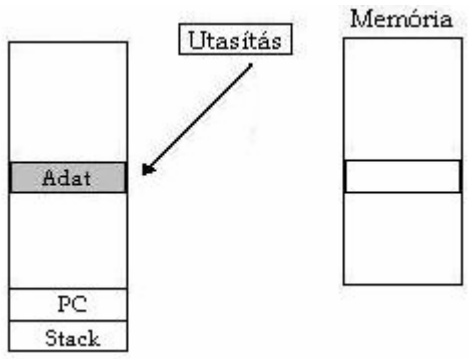
1. Direkt címzés: az utasítás egyértelműen, közvetlenül azonosítja az operandus helyét a memóriában. (effektív cím EA= valódi címén). Pl. **ADD2 X,Y** (X-ben tárolt op1 értéket hozzáadjuk az Y-ban tárolt op2 értékhez, az eredmény az Y-ban lesz.)
EA op1 = X
EA op2 = Y



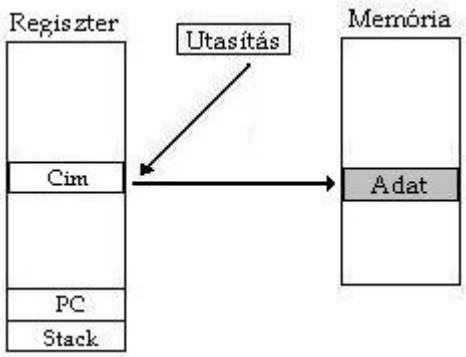
2. Indirekt címzés: az utasítás közvetett módon, (nem közvetlenül az operandusra), hanem az operandus helyére mutat cím segítségével a memóriában. Ezen a helyen pedig az operandus értékét kapjuk meg. Ez sokkal hatékonyabb megvalósítás. Jel: **ADD2 *X,*Y**
EA op1 = MEM[X]
EA op2 = MEM[Y]
Ezt különböző gyártók többféleképpen jelölik. Általában az indirekt címzési módot (*)-al jelölik:
Példa: **ADD2 *X,*Y** (az első op1 értékének címe az X-ben található, a második op2 értékének címe az Y-ban lesz, és az eredmény is az Y-ban tárolódik el.) Az 1, 2, 3, 4, közül ez a leglassabb megvalósítás, de az indirekt címzéssel egy memóriatömb összes eleme elérhető. (Lehetséges direkt-indirekt kombináció is. Például: **ADD2 *X,Y. stb.)**



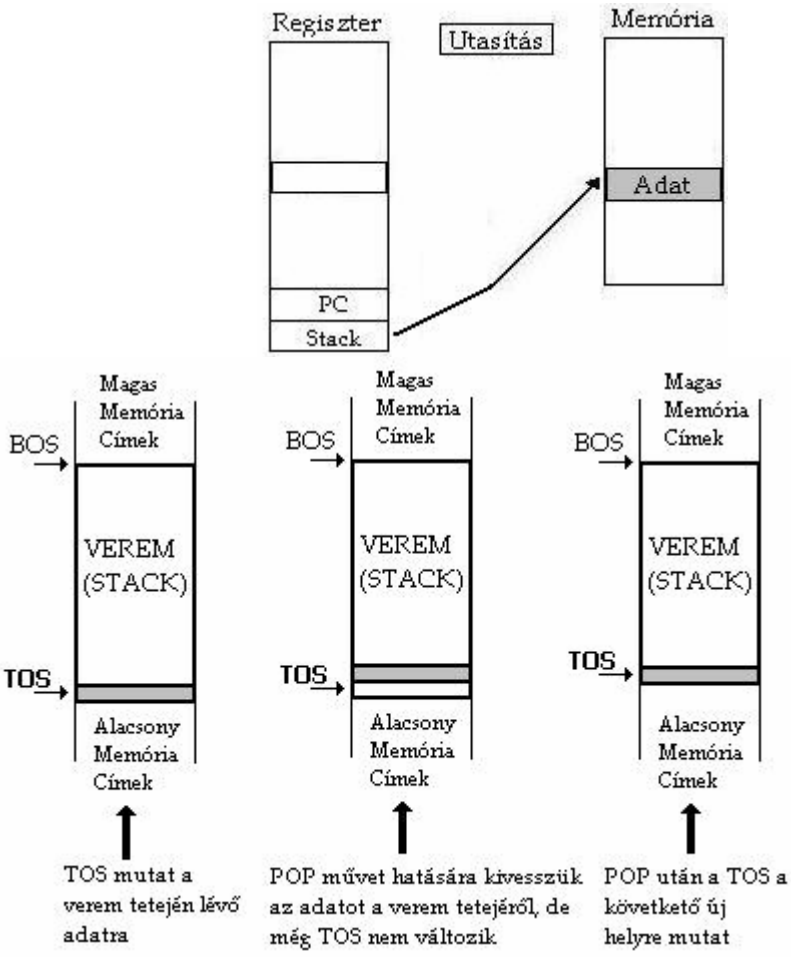
3. Regiszteres direkt címzés: hasonló, mint a direkt címzés, de sokkal gyorsabb, mivel a memória intenzív műveletek helyett a köztes eredményeket a gyors regiszterekben tárolja, és csak a végén tölti át a memóriába. Az 1), 2), 3), 4) közül ez a leggyorsabb módszer.



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



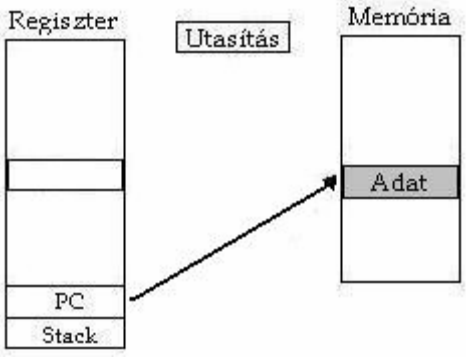
5. Verem (STACK) címzés vagy regiszteres indirekt autoinkrement címzési mód: *indirekt* módszerrel az operandus memóriában elfoglalt helyét a címével azonosítjuk, és akár az összes memóriatömbben lévő elem megcímezhető. *Autoincrement*, is mivel a címeket automatikusan növeli. Ezt (+) jellel jelöljük.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



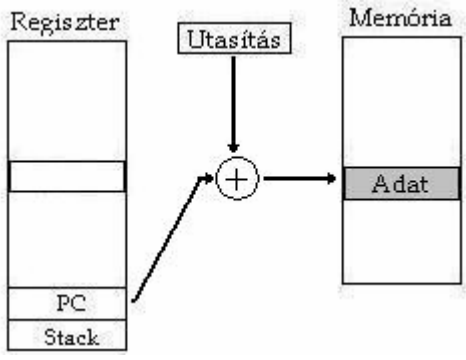
POP művelet kivesszük a stack tetején lévő adatot (TOS), és egy Rx regiszterbe rakjuk. Ezután a stack pointer automatikusan inkrementálja a címet, amivel a következő elemet azonosítja a verem tetején.
MOVE *R (stack pointer)+, Rx

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

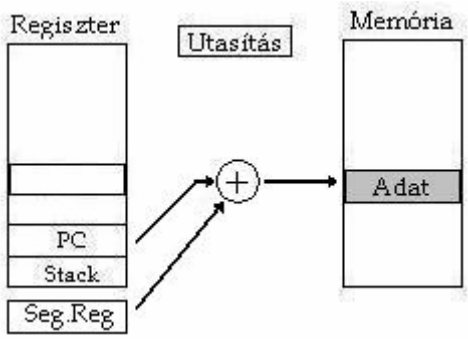
6. Instruction Stream címzés: PC közvetlenül azonosítja a memóriában lévő adatot és címet. Az utasítás végrehajtásakor visszakapjuk az utasításfolyamból magát az utasítást, amelyet a PC azonosít. Nevezik még azonnali módszernek is, mivel az adatok és címek azonnal a rendelkezésünkre állnak. Konstansok, előredefiniált címek szerepelhetnek az utasítás-folyamban.



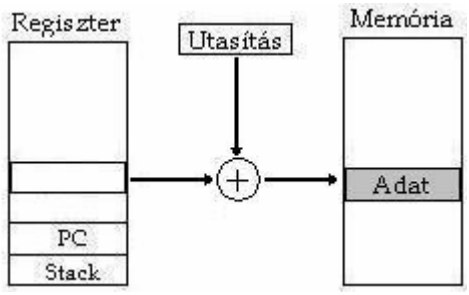
7. PC Relatív címzés: a memóriabeli adatot a regiszteren belüli PC értéke és az utasítás eltolási (offset) értéke azonosítja.
Effektív cím = Regiszteren belüli PC értéke + eltolás (offset) értéke



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



9. Indexelt címzési mód: A memóriában lévő operandus helyét legalább két érték összegéből kapjuk meg. Tehát a tényleges címet az indexelt bázisértékből és az általános célú regiszter értékéből kapjuk meg. Ezt az indexelt címzési módot használják adatstruktúrák indexelt tárolásánál. Pl: tömböknél.
Effektív cím= utasításfolyam bázis értéke + általános célú regiszter értéke



(A könyvben további példák találhatók részletesen!)

9. Vezérlő egységek:

A számítógép vezérlési funkcióit ellátó *szekvenciális* egység. Feladata: az operatív tárban lévő gépi kódú utasítások értelmezése, részműveletekre bontása, és a szekvenciális (sorrendi) hálózat egyes funkcionális részeinek vezérlése (vezérlőjel- és cím-generálás).

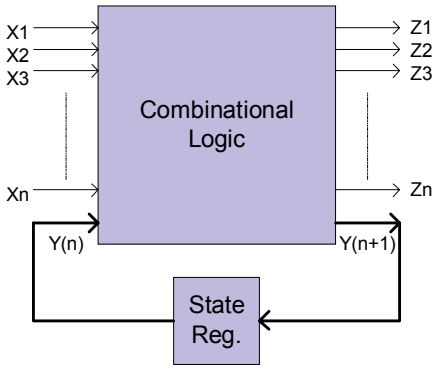
Vezérlő egységek főbb tervezési lépései:

- megfelelő technológia kiválasztása, rendszerkomponensek kiválasztása
- komponensek összekapcsolása a működési sorrendnek megfelelően
- RTL leírás alkalmazása az akciók ill. adatátvitel pontos leírására
- *adatút (data-path)* pontos megtervezése (legfontosabb tervezői feladat – példák a könyvben)
- kívánt vezérlő jelek azonosítása, meghatározása

Vezérlő egységek típusai: - **huzalozott**: 1. Mealy ill. 2. Moore automata modell,
- **mikroprogramozott**: vertikálisan, horizontálisan

a.) Mealy automata modell:

A sorrendi hálózatok egyik alapmodellje. (A KH. = kombinációs hálózatok esetén a kimenetet csak a bemenetek függvényeként kaptuk meg). Azonban a valóságban minden egyes eszköznek van bizonyos minimális késleltetése, és a kimeneten az eredmény véges időn belül jelenik meg, vagy a korábbi értékek visszacsatolódnak a bemenetre. A sorrendi hálózatokat megvalósító vezérlő egységeknél kimenetek nem csak a bemenetek pillanatnyi, hanem a korábbi állapotoktól is függenek. (N memóriaelem esetén 2^N lehetséges állapotot tud a rendszer elfogadni.)



Három halmaza van: X – a bemenetek, Z – a kimenetek, Y – az állapotok halmaza. Visszacsatolni az állapotregisztert a késleltetés miatt kell.

Két leképezési szabály van a halmazok között:

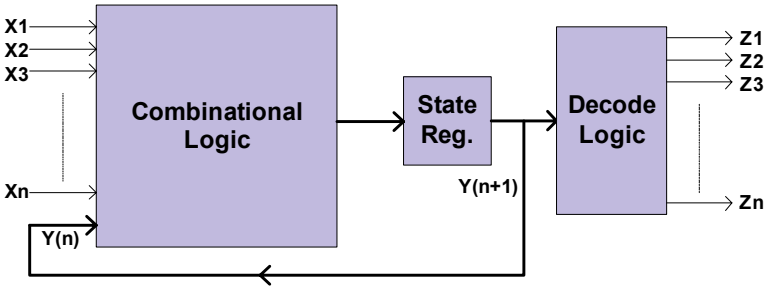
$\delta(X_n, Y_n) \rightarrow Y_{n+1}$: következő állapot meghatározása

$\mu(X_n, Y_n) \rightarrow Z_n$: kimenet meghatározása

Azonban problémák merülhetnek fel az állapotok és bemenetek közötti szinkronizáció hiánya miatt (változó hosszúságú kimenetet kaphatunk). Ezért alkalmazzuk legtöbb esetben a következő Moore féle automata modellt.

b) Moore automata modell:

Itt a kimenetek közvetlenül csak a pillanatnyi állapottól függenek (a bemenettől függetlenek). Tehát a kimenetet nem a bemenethez, hanem az állapotoknak megfelelően szinkronizáljuk. Ezért használunk állapotregisztert.



Három halmaza van: X – a bemenetek, Z – a kimenetek, Y – az állapotok halmaza. Visszacsatolni az állapotregisztert a késleltetés miatt kell.

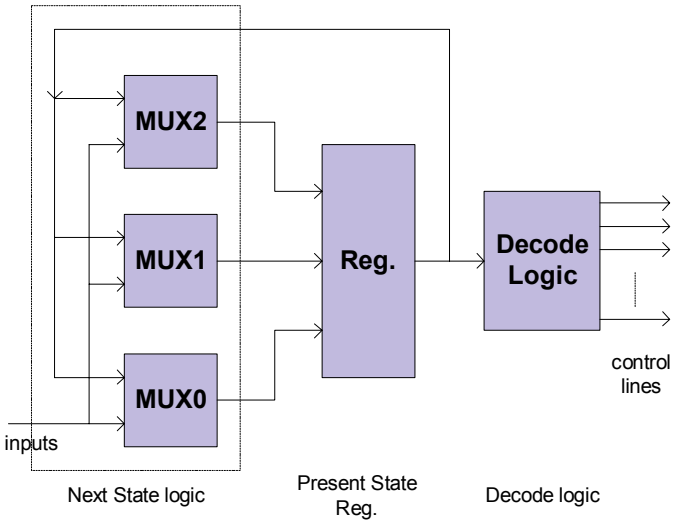
Halmazok közötti leképezési szabály:

$\mu(Y_n) \rightarrow Z$: kimenet meghatározása!

$\delta(X_n, Y_n) \rightarrow Y_{n+1}$: következő állapot meghatározása

/Input – Next State – Present State – Output/

Egyszerű **multiplexeres** és dekódoló logikából álló vezérlő egység: A Moore modell alapjaira épül. A *Present State regiszter* tárolja a rendszer aktuális állapotait. A bemeneti *Next State Logic* a Present State regiszterből visszacsatolt, ill. a bemenetről érkező jelekből választja ki a következő állapotot, és képzi kimenetét. A Next State Logic közvetlenül multiplexerekből épül fel. Végül a kimeneti *Output Logic* dekódolja a Present State regiszterből érkező jelet, és mint vezérlőjelet teszi a kimenetére, amely vezérlővonalakon keresztül jut a megfelelő komponensekhez. Egyazon időpillanatban jön létre a következő állapot, és a megfelelő vezérlő jelek

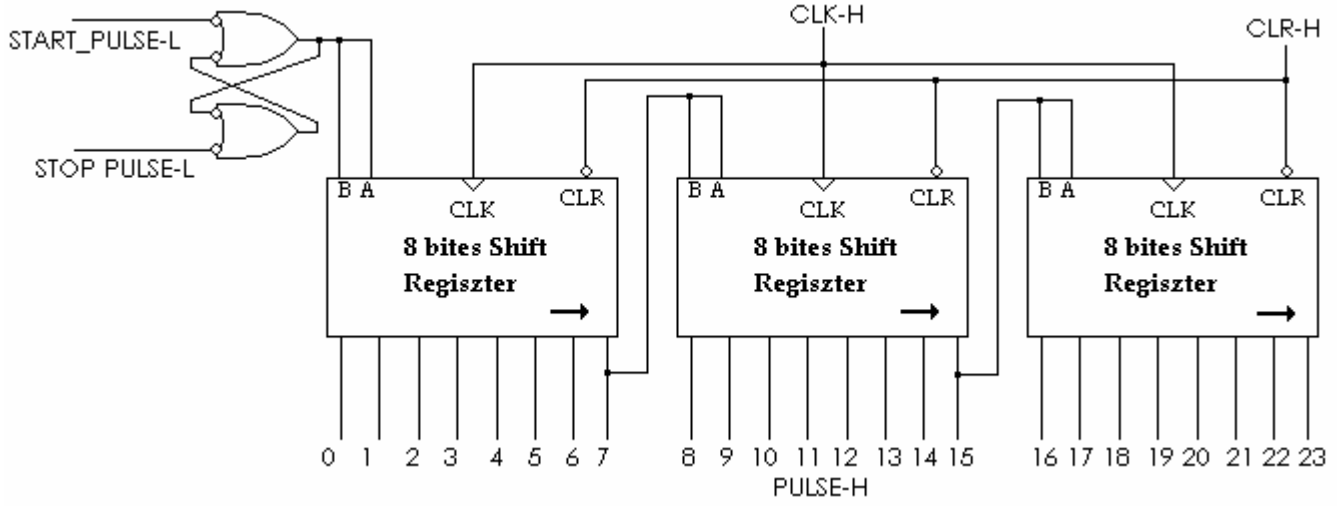


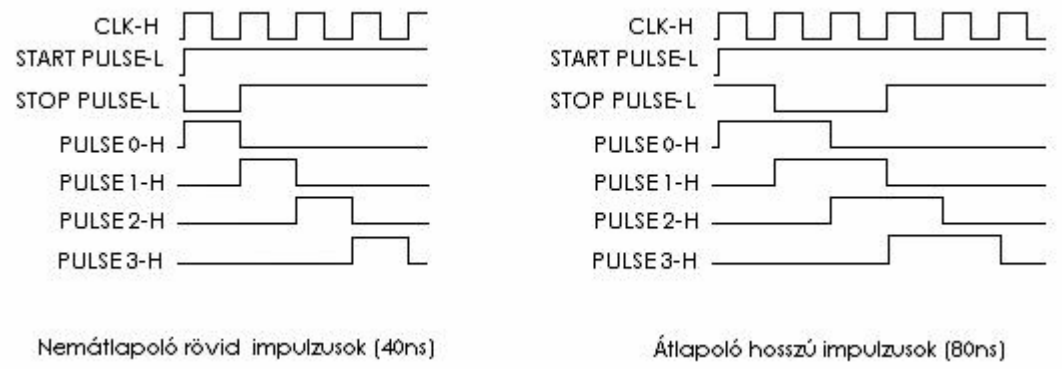
c.) Sorrendi vezérlő egységek megvalósítása Shift regiszterekkel, (az átlapoló impulzusok elve).

A shift regiszteres megvalósítás az egyedi késleltetéses módszerhez áll közel. Adatút diagrammot használunk a vezérlőjelek azonosítására, folyamat diagrammot a regiszter-transzferek (RTL) ábrázolására, míg idő diagrammot a vezérlőjelek kölcsönhatásának leírására. Kapcsolásunk 3db 8-bites shift-regiszterrel (74ALS164) állítja elő az impulzusokat (közvetve vezérlő jeleket is).

Inicializáláskor a kívánt impulzus előállítását a START_PULSE_L beállításával érhetjük el, amelyet a teljes folyamat végéig „L" alacsony aktív szinten tartunk. A következő órajelciklusban a PULSE_0-H jel állítódik be magas jelszintre *rövid 40 ns-os impulzus* ideig. A STOP_PULSE-L vezérlőjelet a PULSE_0-H jel negálásával kapjuk meg (abban az esetben, ha egy ciklus, azaz 40ns ideig tart). Az órajel minden egyes felfutó élére shiftelődik az impulzus. Ekkor nem lapolódnak át, mivel egymás utáni 40ns –os részekből állnak össze, és a megfelelő PULSE_XX kimeneti vezérlőjelek OR kapcsolatából képződnek.

Azonban bizonyos esetekben impulzushibák ún. tüskék keletkezhetnek (pl. ha az egyik jel alacsony szinten marad, a másik viszont magas jelszintre vált). Ezeket a nem megfelelő jelváltásokat vagy SET-RESET flip-flopok-kal küszöbölhetjük ki, vagy pedig alkalmazni kell az **<u>átlapoló impulzusok</u>** technikáját. Ezt az idődiagramot a jobboldali ábra mutatja. Két egység *hosszú impulzusokat* (80ns) egyszerűen létrehozhatunk a PULSE_1-H jel és az invertált STOP_PULSE-L jel OR kapcsolatával (a bal oldali ábra jeleiből !). Az így kapott impulzus mentes lesz a hibáktól, és kiküszöbölhetők a hazárdjelenségek.

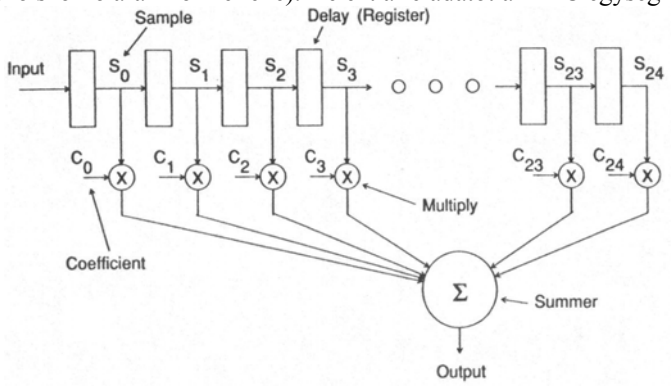




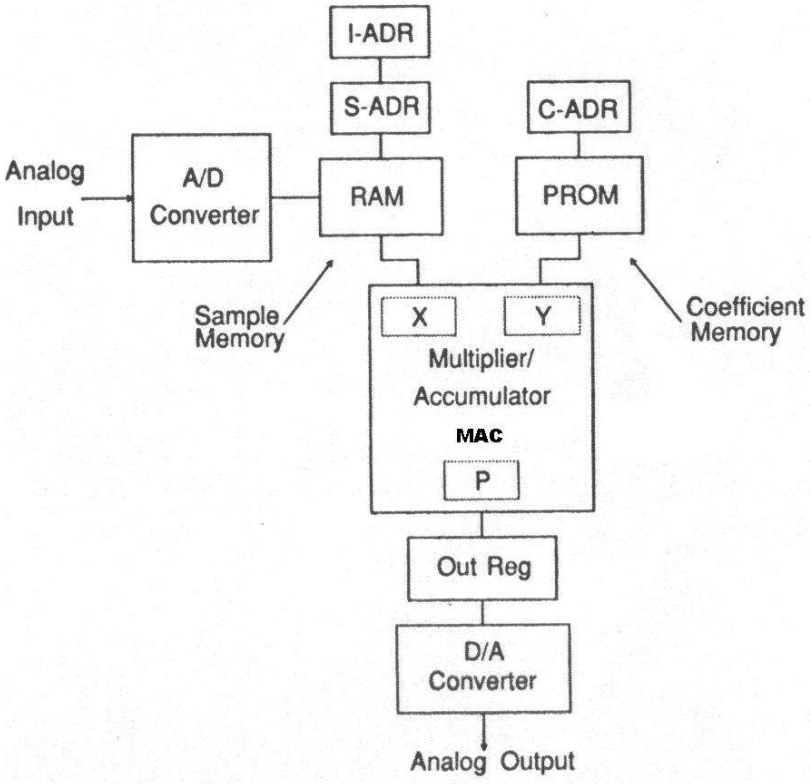
Adatút tervezési példa: FIR (Finite Impulse Response) szűrő esetén (részletes leírás a könyvben!)
FIR=Finite Impulse Response szűrővel egy digitális rendszerben könnyen kiszámíthatók pl. skaláris/ belső szorzat, mátrixműveletek stb. Alkalmazzák a digitális jelfeldolgozásban (DSP), áteresztő szűrőknél. Feladat: tervezzünk egy FIR

$$output = \sum_{i=0}^{24} S_i x C_i$$

szűrőt, amely 25 együtthatót képes kezelni, és a kimenetét képező eredményt a következő egyenlettel kapjuk meg:
ahol Si a bemeneti adatfolyam i. mintája, ill. Ci az i. konstans értéke. A/D konverterrel állítja elő a mintákat, ill. D/A konverterrel alakítja vissza analóg jellé, az eredmény szempontjából elfogadható formába. Minden egyes mintát meg kell szorozni a neki megfelelő együtthatóval (koefficiens), majd a szorzatokat össze kell adni. Egyenként 25 mintát veszünk késleltetve, (elvileg 25 különböző szorzó áramkör kellene). De ezt a feladatot a MAC egység látja el.



A FIR szűrő fontosabb komponensei a következők:
MAC: (Multiplier/Accelerator): elvégzi a szorzást, és az összeadást minden egyes órajel ciklusban. Három regiszterből áll: két bemeneti (X és Y) és egy kimeneti regiszterből (P).
Együttható- memória (Coefficient memory): *C_MEM* tárolja a Ci konstansok értékeit. Ez egy kisméretű PROM memória.
Együttható- memóriacímző regiszter: *(C_ADDR)* egy számláló, amely a következő együtthatót azonosítja. 0-ról indul minden egyes iterációkor és az együtthatók címei egyesével inkrementálódnak.
Minta- memória (S_MEM): az éppen aktuális és az azt megelőző 24 mintát tároljuk el. RAM, amely 32 értékből csak 25-t használ.
Minta- memóriacímző regiszter: (S_ADDR) Ez egy számláló, amely az aktuális minta címét inicializálja, és következő utasítás címére inkrementálódik.
Kezdő minta cím regiszter: (I_ADDR): azonosítja az algoritmus kezdetét.
A/D konverter: (ADIN) Minden iterációban ezen az egységen keresztül kapjuk az új adatot.
D/A konverter: (DAOUT) A kimeneti regiszterből kapja az adatot, és analóg jellé alakítja.
Kimeneti regiszter: (OUT) MAC-ből jövő értéket tárolja és a D/A konverternek továbbítja.



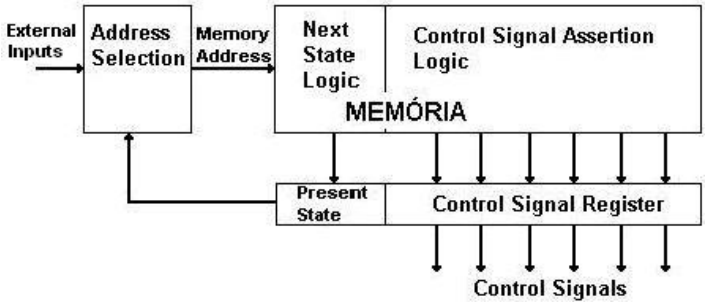
10. Mikrokódos vezérlők:

Általánosságban: a vezérlő egység feladata a memóriában lévő gépi kódú program utasításainak értelmezése, részműveletekre bontása, és ezek alapján az egyes funkcionális egységek vezérlése (a vezérlőjelek megfelelő sorrendben történő előállítása).
Rendszer tervezésekor, miután a feladat elvégzéséhez szükséges vezérlőjeleket definiáltuk, meg kell határozni a kiválasztásuk sorrendjét, és egyéb specifikus információkat (rendszer ismeret, tervezési technikák, viselkedési leírások,
A komplex vezérlési folyamatokat reguláris módszerrel lehet egyszerűsíteni: nevezetesen gyors memóriákat kell használni az utasítássorozatok tárolásánál. Állapotgépekkel (FSM) modellezik a vezérlő egység működését, és ezt a modellt transzformálják át **mikrokódot** használva.
Az ütemező-vezérlő végzi a vezérlő jelek sorrendi előállítását. Lehetnek:

-huzalozott: áramkörökkel fizikailag megvalósított (Mealy, Moore, MUX-os modellek esetén – lásd 9. tétel, PLD-k)

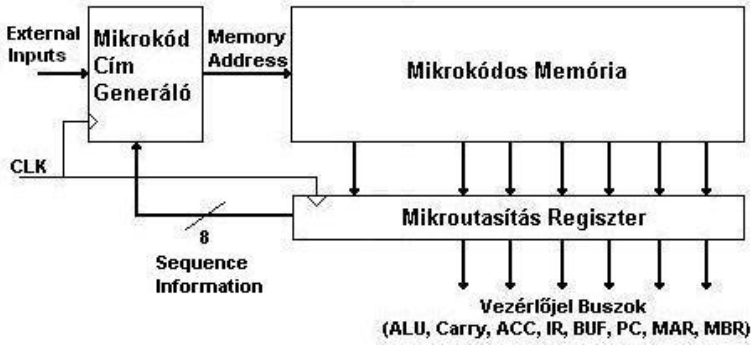
-mikroprogramozott: az adatútvonal vezérlési pontjait memóriából (ROM) kiolvasott vertikális- vagy horizontális-mikrokódú utasításokkal állítják be

Állapotgép megvalósítása memóriával:



Address Selection: (mint új elem) a következő utasítás (Next State), és beállítani kívánt vezérlőjel (control signal assertion) címére mutat a memóriában.
A *memória címet (memory address-t)* külső bemenő jelek és a present state határozzák meg együttesen. E cím segítségével megkapjuk az adott vezérlő információ pontos helyét a memóriában, ill. ez az információ, mint új állapot betöltődik a vezérlőjel regiszterbe (Control Signal Register).
Next State kiválasztásához szükséges log.memória méretét az aktuális állapotok száma, az állapotdiagram komplexitása, és a bemenetek száma határozza meg.
Control Signal generálásához szükséges log.memória méretét a bemenetek száma, a függvény (vezérlő jel) komplexitása, és a vezérlőjelek száma határozza meg.

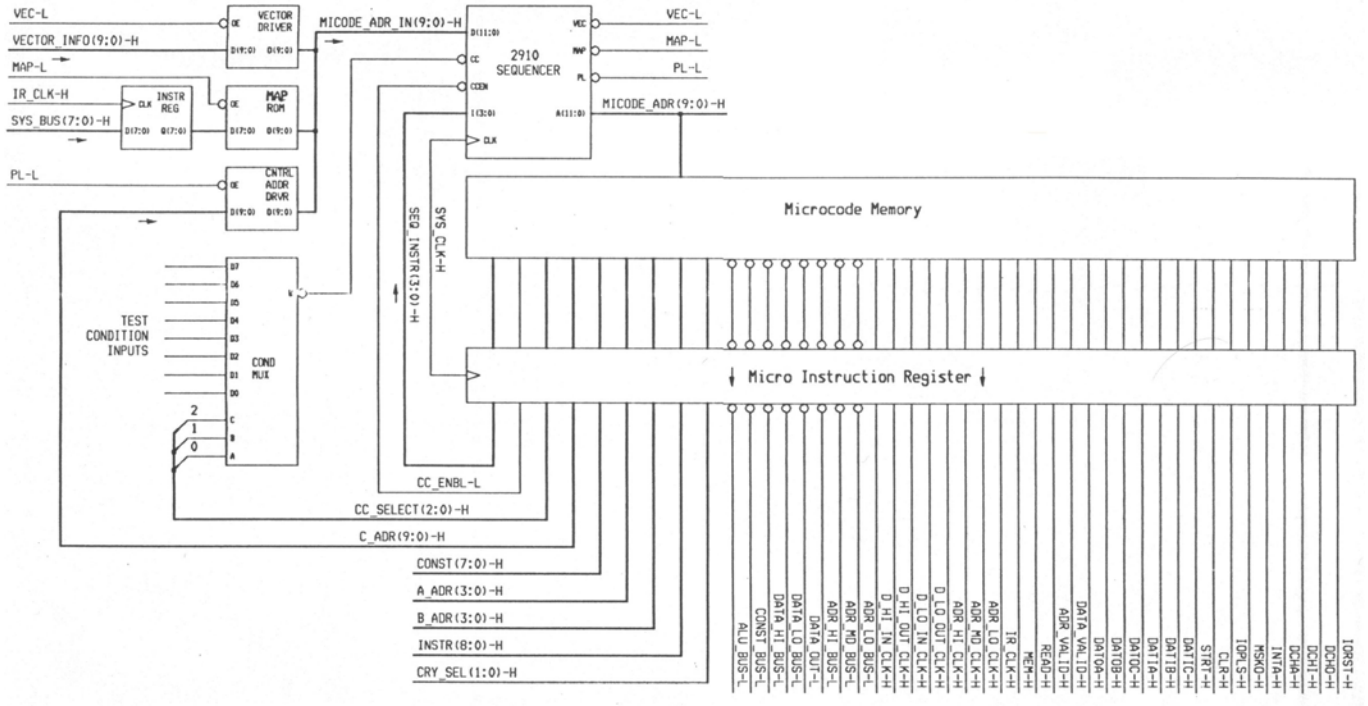
Mikrokódos vezérlő: (az előző ábrán látható állapotgép részeinek összevonásával kapjuk)
Micro Instruction Register: a „Present State" (aktuális állapot) regisztert + a Control Signal regisztert egybeolvasztja (az adatút vezérlővonalainak beállítása / kiválasztása). Mikroutasítások sorrendjében generálódik a vezérlőjel!
Microcode Memory: a Control Signal Assertion Logic vezérlőjel generálás/beállítás + „Next State" kiválasztása (mikroprogram eltárolása) összevonása
Microcode Address Generator: a vezérlő jelet az aktuális mikroutasítások lépéseiként sorban generájla, de címkiválasztási folyamat komplex. Sebesség a komplexitás rovására változhat! (komplexebb vezérlési funkciót alacsonyabb sebességgel képes csak generálni). A következő cím kiválasztása még az aktuálisan futó mikroutasítás végrehajtása alatt végbemegy! Számlálóként működik: egyik címről a másik címre inkrementálódik (mivel a mikroutasításokat tekintve szekvenciális rendszerről van szó). Kezdetben resetelni kell!



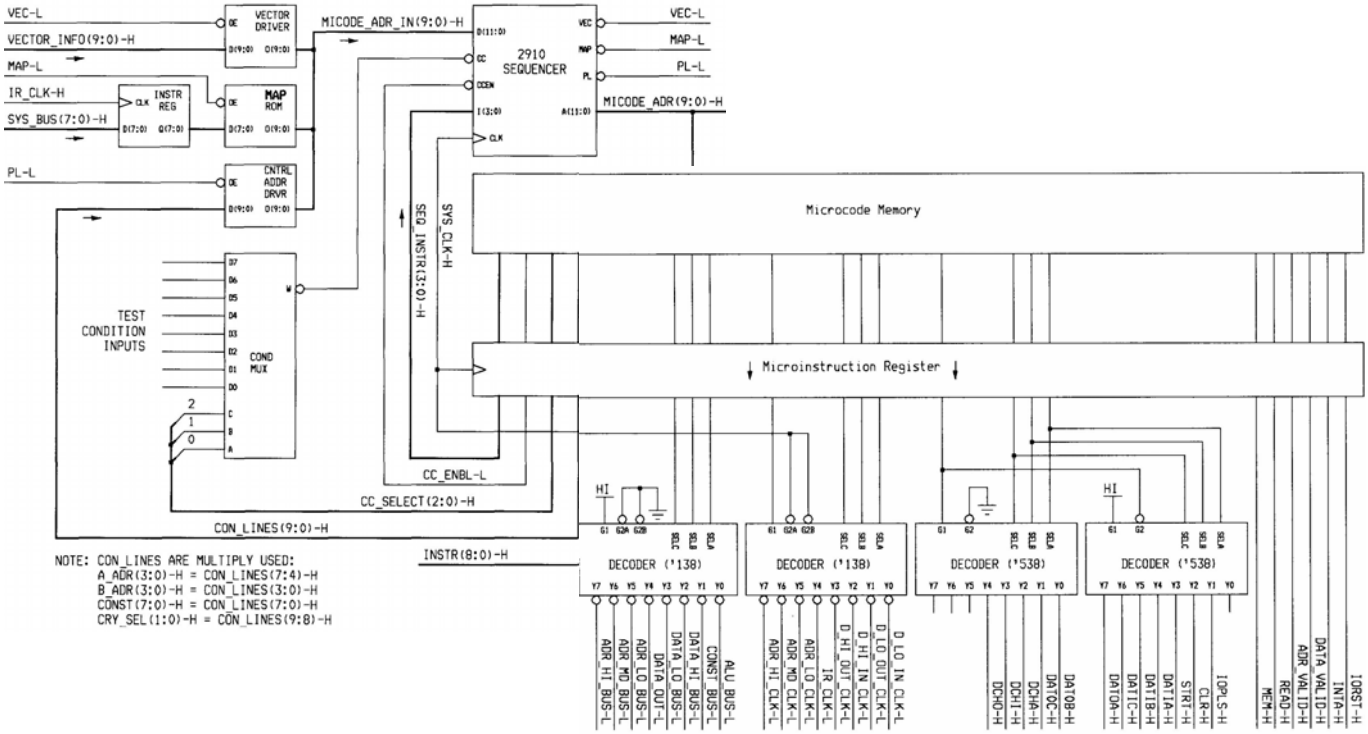
Egy gépi ciklus alatt egy mikroprogram fut le (amely mikroutasítások sorozatából áll). A műveleti kód a végrehajtandó mikroprogramot jelöli ki. A mikrokódú memória általában csak olvasható ROM, ha írható is, akkor dinamikus mikroprogramozásról beszélünk. Ha a mikroprogram utasításai szigorúan szekvenciálisan futnak le, akkor a címüket egy egyszerű számláló inkrementálásával megkaphatjuk. Memóriából érkező bitek egyik része a következő cím kiválasztását (Sequence Information), míg a fennmaradó bitek az adatáramlást biztosítják. A mikrokódos vezérlő egység címeket olvas ki a memóriából, majd a bejövő adatokon műveletet végez. Lassabb, mint a huzalozott vezérlő egységek, hiszen itt a memória elérési idővel is számolni kell (nem csak a visszacsatolt aktuális állapot késleltetésével)

Alapvetően két mikroprogramozási módot különböztetünk meg:

1. Horizontális mikrokód: minden egyes vezérlőjelhez saját vonalat rendelünk, ezáltal horizontálisan megnő a mikro-utasításregiszter kimeneteinek száma, (=horizontálisan megnő a mikrokód). Minél több funkciót valósítunk meg a vezérlőjelekkel, annál szélesebb lesz a mikrokód. Ennek köszönhetően ez a leggyorsabb mikrokódos technika, mivel minden bit független egymástól ill. egy mikrokóddal többszörös (konkurens) utasítás is megadható. Pl: a megfelelő regisztereket (memória, ACC) egyszerre tudjuk az órajellel aktiválni, ezáltal egy órajelciklus alatt az információ mindkét irányba átvihető. Növekszik a sebesség, mivel nincs szükség a vezérlőjelek dekódolását végző dekódoló logikára. Így minimálisra csökken a műveletek ciklusideje. Azonban nagyobb az erőforrás szükséglete, és fogyasztása.

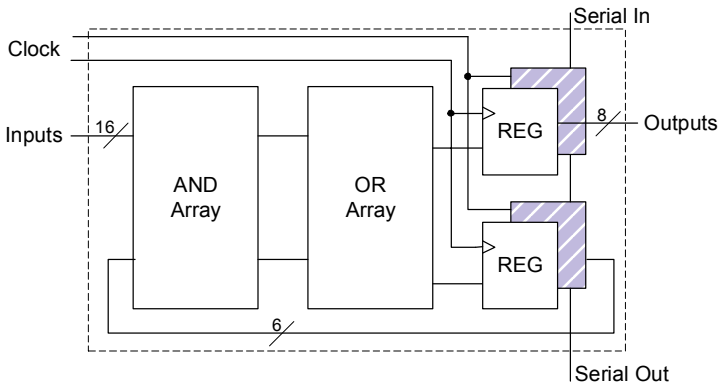


2. Vertikális mikrokód: nem a sebességen van a hangsúly, hanem hogy takarékoskodjon az erőforrásokkal (fogyasztás, mikrokódban a bitek számával), ezért is lassabb. Egy időben csak a szükséges (korlátozott számú) biteket kezeljük, egymástól nem teljesen függetlenül, mivel közülük egyszerre csak az egyiket állítjuk be. A jeleket ezután dekódolni kell (a dekódolás több időt vesz igénybe). A kiválasztott biteket megpróbáljuk minimális számú vonalon keresztül továbbítani.
A műveletek párhuzamos (konkurens) végrehajtása korlátozott. Dekódolás: N bites buszt, $\log_2(N)$ számú bittel próbálunk kódolni. Több mikroutasítás szükségeltetik -> így a mikrokódú memóriát „vertikális" irányban kell megnöveli.



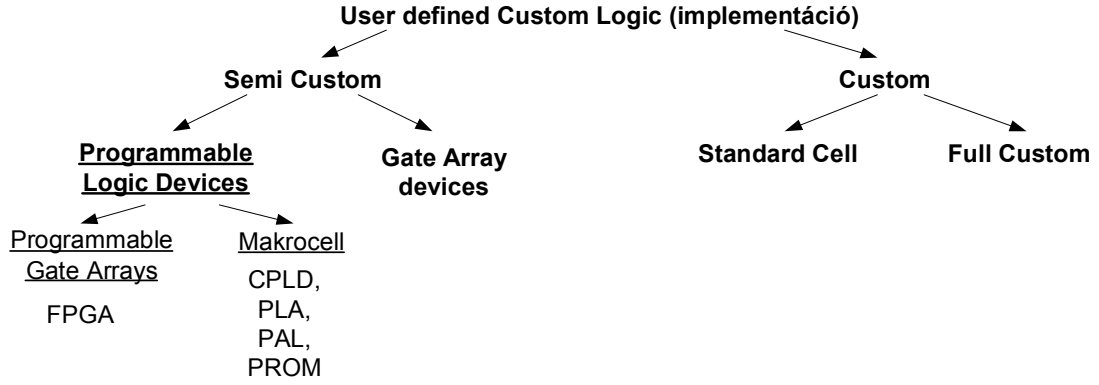
11. Vezérlő egységek programozható alkatrészekkel:

Egy vezérlő egység Next State logikai blokkjának megvalósítása a tervező, gyártó feladata. Általában változó logikát használnak a függvények megvalósításánál. A felhasználó által programozható logikai sorrendvezérlő (**Field Programmable Logic Sequencer**) programozható alkatrészekből építhető fel.



A logikai sorrendvezérlőnek 16 külső bemeneti vonala van, 1 RESET és 1 kimenet-engedélyező vonala, ill. 8 kimeneti vonala. A regiszterek egy-egy állapotot tárolnak, amelyek az órajel hatására a kimenetre íródnak. Továbbá 6 belső visszacsatoló vonala van. A Next State ill. a kimeneti szintek meghatározásánál programozható AND/OR tömböket használnak.
Működése: - normál (D-FF)
- debug (Shift Reg) - állapotok

Felhasználó által definiált logikai implementációk a következők:

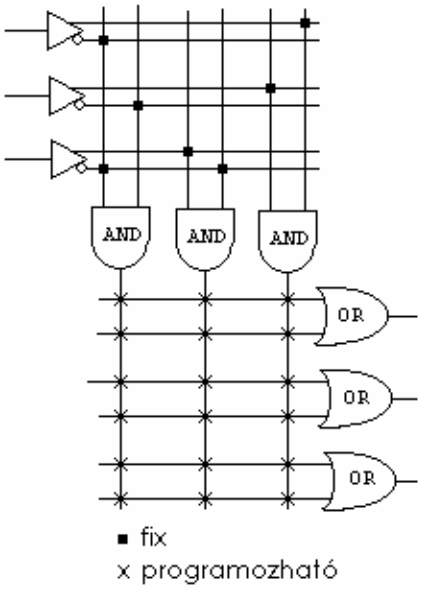


Programozható logikai kapcsolások két fő típusa:
1.) Makrocellák / PLD-k: (Programmable Logic Devices):
- PLA
- PAL
- PROM
- EPLD, CPLD

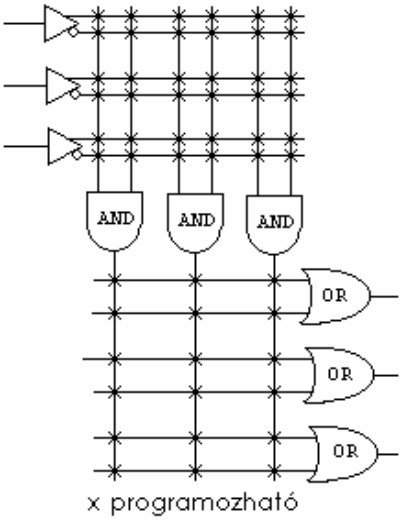
2.) FPGA (Field Programmable Gate Array): Programozható Gate Array áramkörök
- XILINX, Altera, Actel (főleg űrkutatásban) sorozatok

1.) Makrocellás típusok:

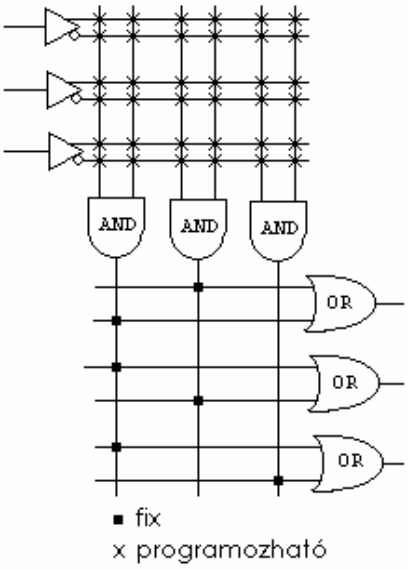
a.) PROM: (Programmeble Read-Only Memory): Speciális feladatok ellátására készítenek ilyeneket a gyártók. A benne lévő információt írás után már nem lehet megváltoztatni (kivétel: EPROM, EEPROM), ezért tápfesz nélkül is megőrzik tartalmukat. Egy cella felprogramozása a felhasználó által történik, de csak egyszer egy biztosíték átégetésével. A cellák négyzetes mátrixban helyezkednek el, címzésük a megfelelő sor-oszlop vonalra „1”-et adva történik. ÉS kapuval aktivizáljuk a kívánt cellát. A sor-oszlop címeket (vagyis a címvektort, amelyek 2-4-8… bitesek, 2 valamely hatványa) dekódolni kell. Az összes lehetséges minterm előáll az OR kapcsolatukként (n bemenetből 2^n kimeneti kombinációval): tehát az AND=fix, és az OR=programozható. (Pl: EPROM/ EEPROM). OR kapuk kimeneténél mindenhol tároló regiszterek (pl. D flip-flopok) helyezkednek el, amik visszacsatolódhatnak a bemenetekre!



b.) PLA: (Programmable Logic Array): alapja a diszjunktív normálforma=mintermek vagy kapcsolata. Tartalmaz 3-állapotú invertereket, AND, OR kapukat. A felső mátrixban a ponált bementi változók, és negáltjaik között teremtünk ÉS kapcsolatot a kereszteződő vezetékek segítségével. Az alsó mátrixban az így kapott ÉS kapuk kimeneteit felhasználva, kialakíthatók a VAGY függvények (kimeneti változóként csak 1-1 VAGY kapu szükséges.). Tehát itt mind az AND, mind pedig az OR kapuk programozhatóak! Az OR kapuk kimeneteinél mindenhol D-tárolók vannak, amik visszacsatolódhatnak a bemenetekre!

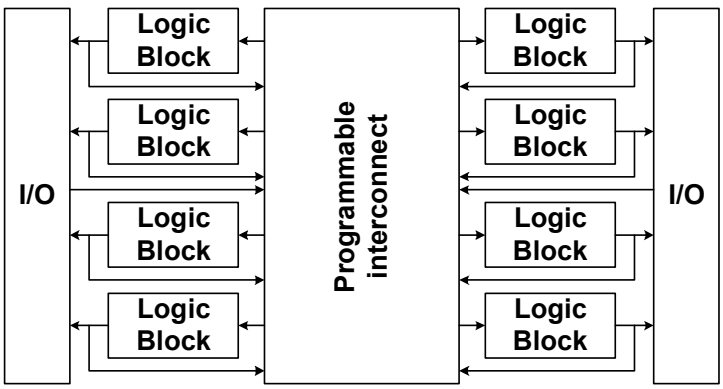


c.) PAL: (Programmable Array Logic): Hasonlóan viselkedik, mint a PLA, de itt csak az AND kapuk programozhatóak, az OR kapuk fixek.
Gyorsabb, mint a PLA, fix összeköttetések miatt, kevesebb kapcsoló kell.
Kimenetein D-tárolók helyezkednek el, amik visszacsatolódhatnak a bemenetekre!

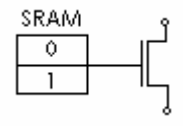


d.) CPLD (Complex Programmable Logic Device):
Logikai Blokk-ban található elemek:
- PAL / PLA
- Regiszterek

Interconnect (összeköttetés) lehet:
- Teljes, vagy
- Részleges összeköttetés hálózat

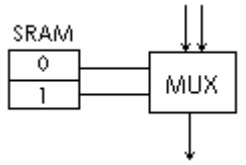


Hogyan programozhatók a VLSI alkatrészek? Programozási technikák a következők:

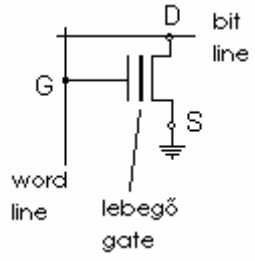


a.) SRAM-os: az SRAM cellára egy áteresztő tranzisztor van csatolva. A tranzisztor vagy kinyit (vezet), vagy lezár. Az SRAM értéke, ami egy bitet tárol (’0’ vagy ’1’) letölthető. Összeköttetéseket, vagy MUX-ok állását is eltárolja.
Tulajdonságai:

- végtelen sokszor újraprogramozható (statikus RAM)
- táp kikapcsolása után az SRAM elveszti tartalmát
- bekapcsoláskor (inicializáláskor) a programot be kell tölteni, fel kell programozni
- az SRAM cellára egy áteresztő tranzisztor van csatolva. A tranzisztor vagy kinyit (vezet), vagy lezár. Az SRAM értéke, ami egy bitet tárol (’0’ vagy ’1’) letölthető. Összeköttetéseket, vagy MUX-ok állását is eltárolja.
- 1 bit tárolása az SRAM-ban (min. 6 tranzisztorból áll)
- sok tranzisztor (standard CMOS), nagy méret, nagy disszipáció
- nem kell frissíteni az SRAM-ot
- nagy 0.5-2 kΩ átmeneti ellenállás
- nagy 10-20 fentoF parazita kapacitás

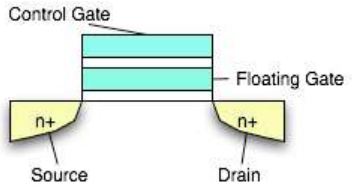


b.) Multiplexeres: az SRAM-ban tárolt ’0’ vagy ’1’ bitet használunk a Multiplexer bemeneti vonalának kiválasztásához. (Működése hasonló az SRAM celláéhoz.)



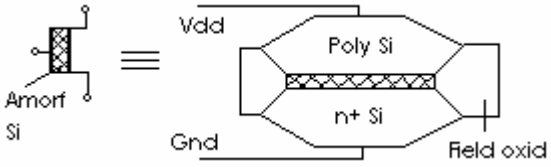
c.) Floating Gate (lebegő gate): Két-gates tranzisztort használunk, amelynek középső gate-je a lebegő gate. A másik gate fix. INTEL alkalmazza. Programozható összeköttetéseknél, csomópontokban használatos.
Tul:

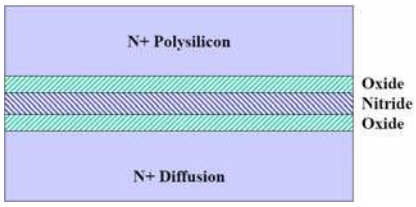
- programozása: töltéseket viszünk fel a word-line felől a lebegő gate-re, kinyit a tranzisztor (a control gate befolyásolja a működését)
- többször törölhető (kis ablakon keresztül UV fénnyel)
- írás: nagy feszültség hatására töltést viszünk fel a lebegő gate-re
- kikapcsoláskor is megőrzi tartalmát, töltések nem sülnek ki (akár 99 évig)
- nagy 2-4 kΩ átmeneti ellenállás
- nagy 10-20 fentoF parazita kapacitás



d.) Antifuse:
A tranzisztor Gate-jét amorf kristályos Si alkotja, amelyet relatíve nagy feszültség (kb 20-30V) hatására átkristályosítunk, így vezetővé válik végérvényesen. Texas Instruments alkalmazza.
Tul:

- az „átégetés” irreverzibilis folyamat, nem lehet újraprogramozni
- csak egyetlen egyszer programozható
- kis méreten megvalósítható, kis disszipáció
- kis átmeneti ellenállás 300 Ω
- kis parazita kapacitás 1.1-1.3 fentoF
- előállításához sok maszkréteg szükséges, drága technológiát igényel
- Típusai: Amorf Si vagy ONO (Oxid-Nitrid-Oxid)





e.) EPROM/EEPROM/FLASH-os: (Lattice)

- **A floating-gates technológiát alkalmazza**!
- Megőrzi tartalmát (non-volatile)
- elektromosan írható/törölhető végtelen sokszor (EEPROM, FLASH)
- UV-fénnyel törölhető (EPROM)
- nagy átmeneti ellenállás 2-4 kΩ
- nagy parazita kapacitása
- További gyártástechnológia (sok maszk-réteg alkalmazása szükséges) - drága

FPGA (Field Programmable Gate Array) architektúrák

(További információk részletesen: http://www.xilinx.com)

Az előadásokon részletesebben a XILINX által gyártott – felhasználó által „gyorsan” újraprogramozható, konfigurálható – Gate-Array áramkörökkel ismerkedtünk meg.

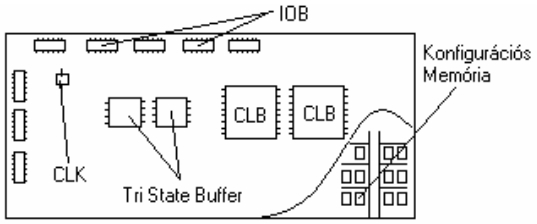
A Xilinx FPGA áramköröknek három alapvető építőeleme van:

CLB: *konfigurálható logikai blokk*: szükséges logikai kapcsolatok megvalósítása egy logikai tömbben. Tartalmaz 2db D Flip-Flop-ot és 2db független 4-bemenetű LUT-t tetszőleges (F és G): logikai függvényeket lehet velük megvalósítani.

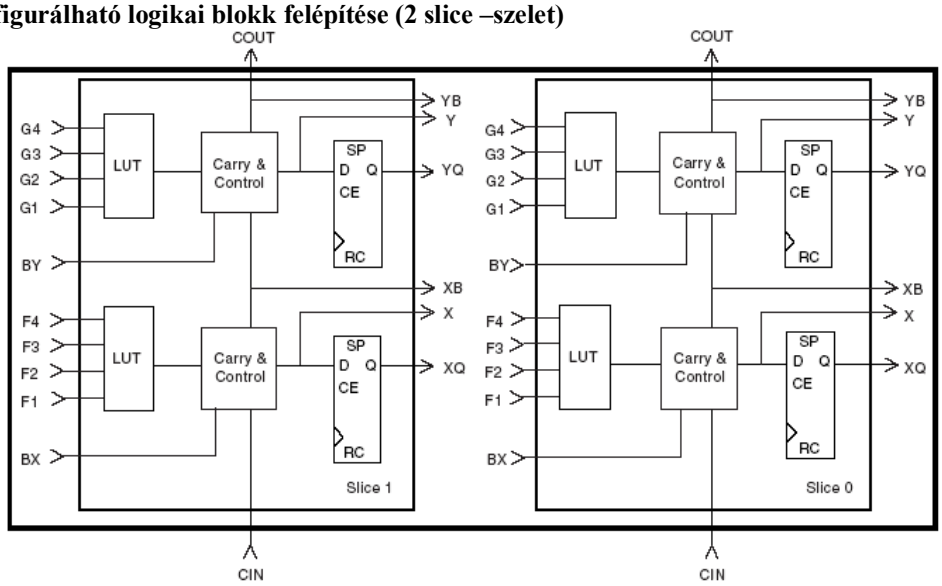
IOB: *programozható be/kimeneti blokk*: kapcsolat a belső vonalak és a tokozás lábai (pad) között (lehet bementi /kimeneti / kétirányú (tristate) kapcsolatként definiálni). Szintén D-FF-kel rendelkezik, belső puffere (buffere) van, nem kell az adatokat a külső lábon (pad) tartani. A kimenet lehet negált, vagy ponált.

PI: Programozható összeköttetések: az IOB-k és CLB-k kivezetéseinek (lehetnek TTL v. CMOS) összehuzalozása. Konfigurációs memóriában vannak tárolva az IOB-k, CLB-k és PI-k állapotai. Az összes belső összeköttetés programozható kapcsolókkal valósítható meg. Három típusa: egyszeres /dupla /hosszú összeköttetés.

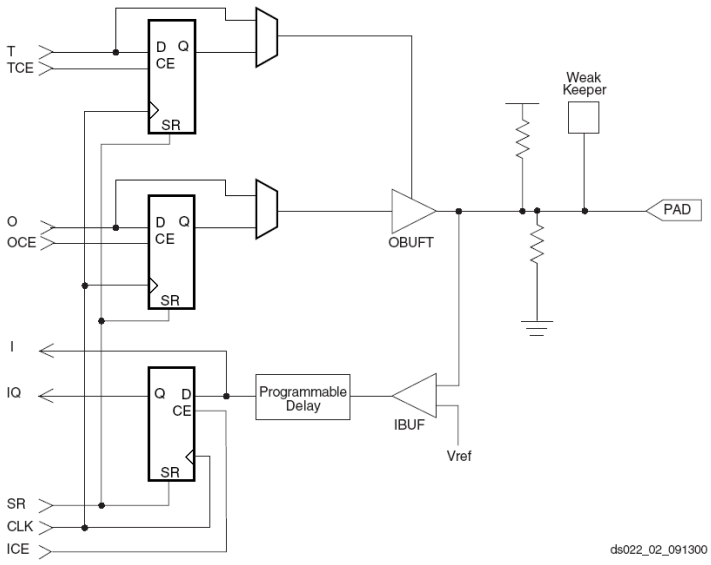
Ezek mellett a Xilinx tartalmaz tri-state buffereket, CLK órajel-meghajtó áramköröket, és konfigurációs memóriákat is. Minden Xilinx processzorral nagyszámú kapu programozható, 50-150MHz-es tartományban működtethetők, tehát gyorsak. (ezek az adatok a korábbi XC3000, XC4000 családok esetén álltak fenn). Egy Xilinx FPGA általános felépítése a következő:



Virtex CLB: Konfigurálható logikai blokk felépítése (2 slice –szelet)

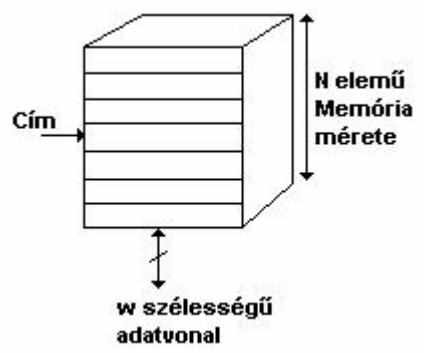


Virtex IOB - I/O Blokk felépítése:



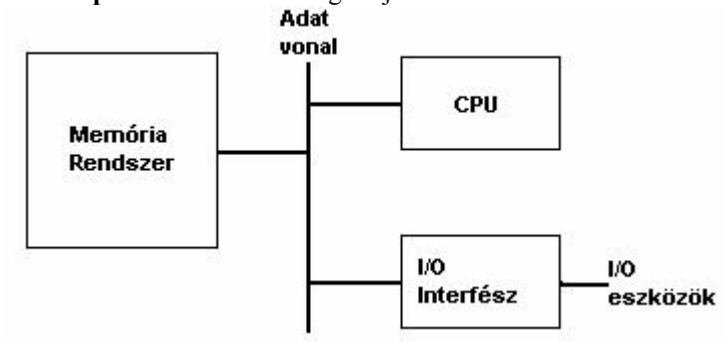
12. Memóriák:

MEMÓRIÁK: a számítógép adatait, utasításait tárolja. Memóriát meghatározó alapvető tényezők a következők:

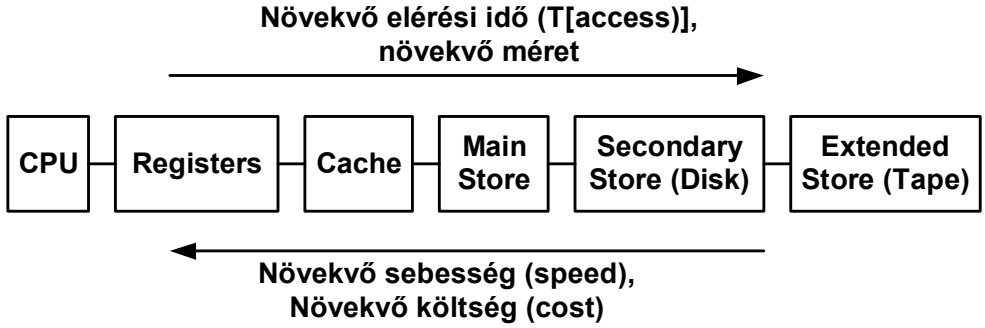


Paraméterek:
- memória mérete (N elemű, rekeszű)
- címvezetékek száma N függvényében: $\log_2(N)$
- w: adatvonal szélessége
- memóriaszervezési módszer
- sebesség
  - T_a: minimális (access time) elérési idő: az a legrövidebb idő, ami a cím kiadásától az adat megjelenéséig tart.
  - T_{ct}: memória (cycle time) ciklus idő: minimális idő két elérés között (burst módban)

Memória - CPU és I/O **interface-k kapcsolatának** blokkdiagramja:



Memória hierarchiák a sebességviszonyok és méret megkülönböztetése szempontjából lehetnek:

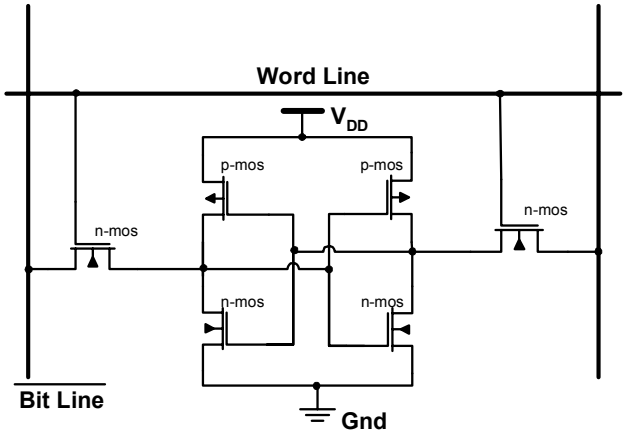


- Regiszterek: a gyorsabbak, mivel fizikailag a legközelebb vannak a processzorhoz: kis számú, nagyon gyors regiszterbankot használunk. De nagyon drágák.
- Cache memória: a leggyakrabban előforduló információt tárolja, növeli a műveleti (végrehajtási) sebességet (L1, L2, L3)
- Main Store/Memória: programok és az adatok itt tárolódnak (Neumann v. Harvard architektúra szerint!). Elegendő mennyiség szükséges az OS zavartalan működéséhez. (ns-os elérési idő)
- Secondary Store/Disk. háttértárolók, Másodlagos-merevlemezes tároló elemek, nagy kapacitásúak. Fileok, könyvtárak, swap-terület. Hosszabb távon tárolunk. (ms-os elérési idő)
- Extended Storage/Tape: szalagos, lemezes meghajtók, mágneslemezek. Nagyon lassúak.

Memóriafajták elérésük/hozzáférésük szerint:
- SAM Volatile M: Serial Access Memory. Soros elérési idejű, tápfesz. kimaradásakor felejtő ún. „elpárolgó" memória
- RAM: Random Access Memory: Véletlen hozzáférésű memória, írható és olvasható
- ROM: Read Only Memory: Csak olvasható memória, véletlen elérésű (RAM is egyben)
- PROM: programmable ROM: programozható vezetékek közötti ellenállások/biztosítékok átégetésével ill. átkötésével.
- EPROM: elektromosan programozható, UV-fénnyel (ablakon keresztül) törölhető.
- EEPROM: elektromosan programozható és törölhető
- SRAM: tápfesz. alatt megőrzi tartalmát, gyors, nem kell frissíteni, nagy hegyet foglal
- DRAM: tápfesz. alatt is elvesztené a tartalmát, frissíteni kell, ezért lassú
- RWM: Read/Write Memory: írható /olvasható memória
- CAM: Content Address Memory: tartalom szerint címezhető memória (asszociatív memória) Pl :CACHE memória.
- FLASH: memóriakártyák

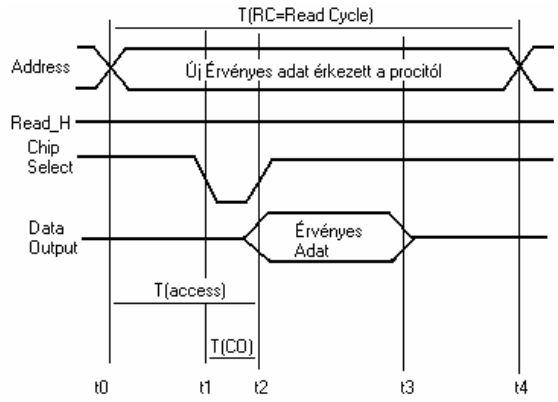
Az információtárolás két alapvető formája félvezető memóriák esetén a következő:
- SRAM: Statikus memória
- DRAM: Dinamikus memória

SRAM cella: 6 db CMOS tranzisztorból építhető fel. n-mos és p-mos (n és p csatornás tranzisztorokból épül fel) 2-2 db, + 2 db áteresztő tranzisztor. Word line: írás / olvasás művelet, Bit line: érték 0 / 1
(Ezen kívül épülhet bipoláris tranzisztorból ill. tisztán nMOS tranzisztorból) -

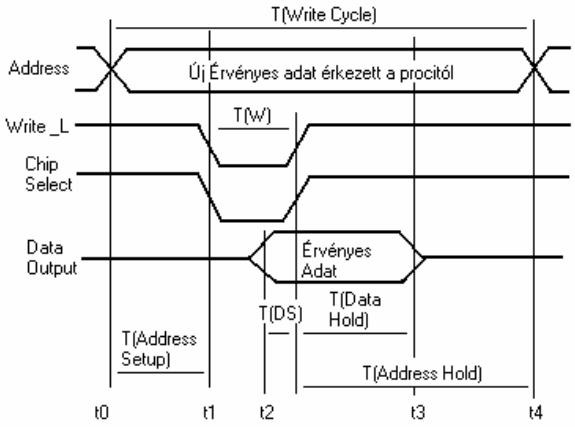


Statikus memória írási/olvasási folyamatai: (idődiagram)

Olvasási ciklus:

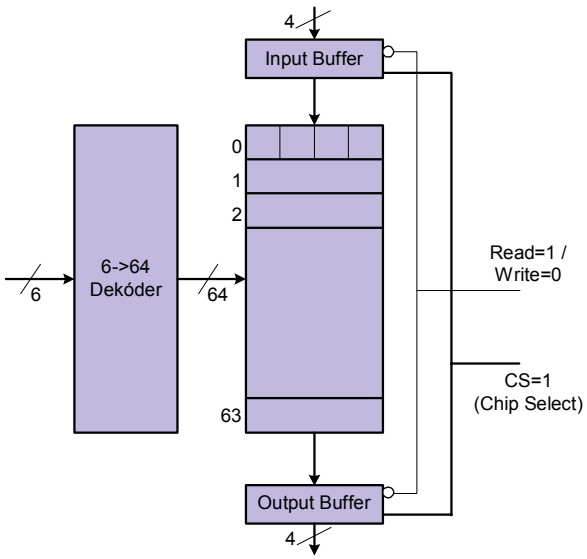


Írási Ciklus:

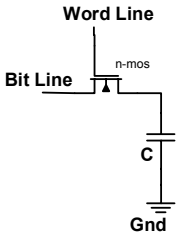


SRAM tulajdonságai:
Az információ a tápfesz alatt is megmarad, nem kell frissíteni. SRAM esetén T(R/W Cycle)≈T(Access Time) közel azonos. Megvalósítható bipoláris (0,1) SRAM cellával, nMOS tranzisztorokkal, vagy CMOS tranzisztorokból (6 tranzisztor). Kisebb kapacitású memória építhető fel SRAM-ból, de gyorsabb a DRAM-nál, mivel tápfeszültség alatt sem kell frissíteni, viszont nagy a fogyasztása. Integritási sűrűsége 4x rosszabb a DRAM-nál. Tápfeszültség kikapcsolásával elveszti a tartalmát.
Felhasználása: cache memóriákban, digitális oszcillátorokban, logikai analizátorokban, operatív tárakban (memória).

SRAM 64x4bites felépítése:

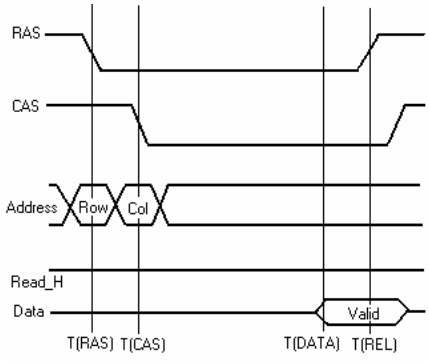


DRAM cella: Tárolás: egy kisméretű kondenzátor (C) töltése és kisülése.

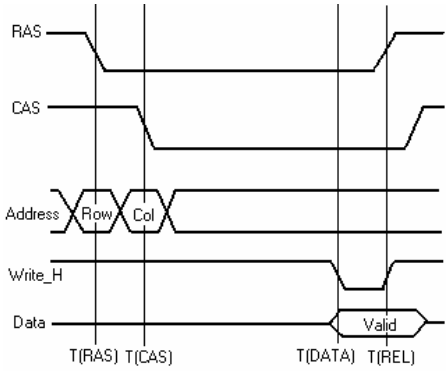


Dinamikus memória írási/olvasási folyamata: (idődiagram)

Olvasási ciklus:



Írási ciklus:



Olvasási ciklus:

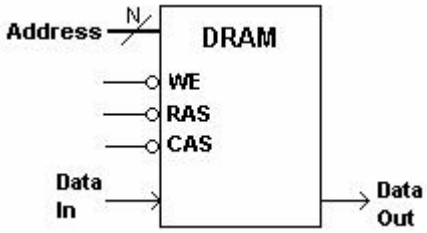
A DRAM-oknak több címvonalra van szüksége, mivel a címek két részre, egy sor- és egy oszlop-azonosítóra (RAS-CAS) vannak felosztva. Miután a RAS jelet alacsony szintre állítja be (lefutó élre vezérelt) a sorcím megjelenik a címvonalon. Ezt kitartja rövid ideig, majd a CAS jelet állítja be alacsony szintre, és az oszlopcím jelenik meg. Ezután válik elérhetővé a memória (setup time). Majd a memória elérési idejétől (T access time) függően kis idő múlva az adat érvényessé válik (Valid). Amikor az RAS felszabadul (T (REL)) a kimeneten az adat visszatér a háromállapotú (tri-state) feltételhez. Read_H végig magas aktív, hiszen olvasási ciklust hajt végre.

Írási ciklus:

Az olvasástól csak annyiban különbözik, a WE (write enable), Write_L jel engedélyezi az írást, (az írás lefutó élre történik).

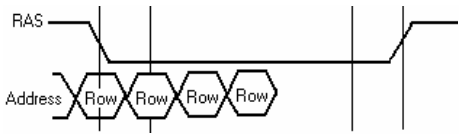
DRAM-ok tulajdonságai:

MOS technológiával készülnek. A tápfeszültség alatt is frissíteni kell *2-10 ms*-ként, mivel idővel elvesztik tartalmukat. Nagyobb kapacitású, mint az SRAM, de méretben is nagyobb, bonyolultabb és lassabb is. A hozzáférési idő kétszer nagyobb a memória R/W ciklusidejénél: 2*T(R/W Cycle)=T(Access Time). Integritási sűrűsége 4x jobb. (IRAM: az időzítő elektronika a DRAM-ra van integrálva, úgy látszik, mintha SRAM lenne, a frissítést tekintve). A CS=Chip Select jelet két részre osztottuk fel: RAS=sorkijelölő, és CAS=oszlopkijelölő komponensekre.



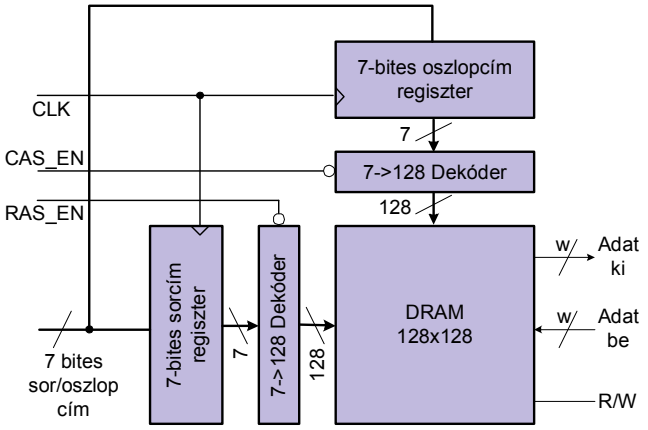
Dinamikus memória frissítési folyamata! (idődiagram)

Frissíteni kell, mivel kondenzátoron tároljuk az információt, melynek feszültsége idővel exponenciálisan csökken, tehát kisül.



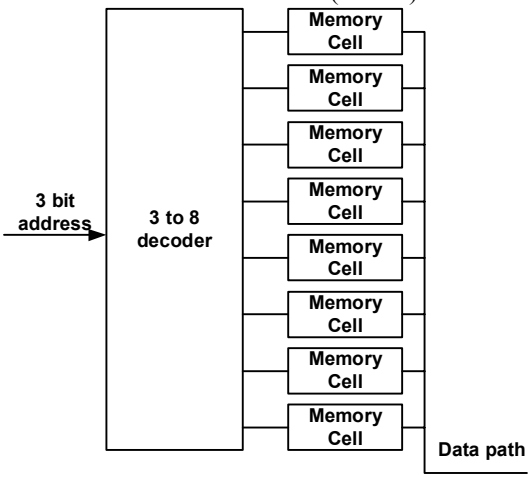
A frissítéskor először például egyetlen CAS oszlopcímet adunk ki, majd a RAS-al az összes sorcímet, hogy az összes cella frissítésre kerüljön. Frissítés történhet a kiolvasási ciklus végeztével is (még a következő beírás előtt). Átlagosan: 2-10ms.

128x128-as DRAM felépítése:

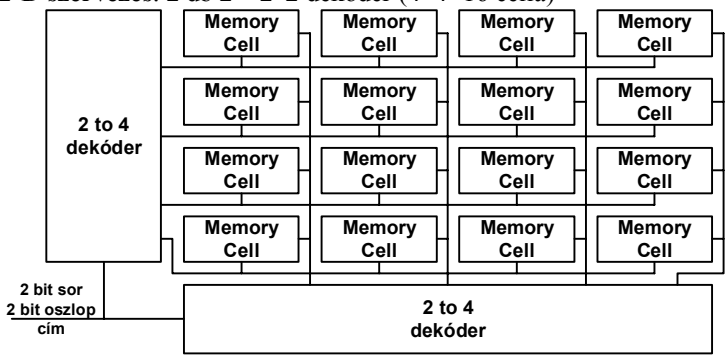


RAM-ok alapcellái: elrendezésük szerint – 1 Dimenziós ill. – 2 Dimenziós RAM struktúrákat ismerünk. A dekódoló egység megválasztása a tömbben lévő elemek számától függ. 1 Dimenziós esetben N bit esetén N→2^N típusú dekódoló szükséges. 2-D-s esetben pedig N bit esetén felosztjuk sorokra és oszlopokra: N=X+Y. Ezek a címvonalak X→2^X ill. Y→2^Y típusú dekódolók szükségesek az egyes sorokban, ill. oszlopokban lévő elemek azonosításához.

1-D szervezés: 3→2^3 dekóder (8 cella)



2-D szervezés: 2 db 2→2^2 dekóder (4*4=16 cella)

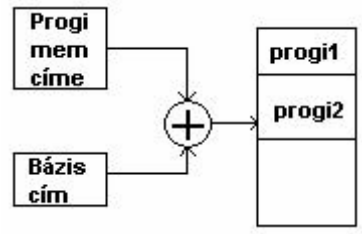


LAPOZÁSI TECHNIKÁK: (lásd 13. tétel)

13. Virtuális memóriakezelés:

Ha a rendelkezésre álló fizikai memóriaterület kevesebb, mint amit a programok igényelnek, akkor virtuális memóriára van szükségünk. Virtuális memóriakezelés: kapcsolat a rendelkezésre álló fizikai, ill. a program által igényelt logikai memória megfelelő szervezése között. Feladat: a program által igényelt virtuális címet, a valós memóriacímre kell konvertálni. Ezt nevezzük címfordításnak.

- Az OS elrejti a program számára a fizikai memóriakorlátot, azzal hogy virtuális memóriát használ.
- Multi-programming: egyszerre több program fut, minden programnak saját logikai/bázis címtartománya van, amellyel hivatkozni lehet rá.
- Fizikai cím = program memóriacíme + program báziscíme



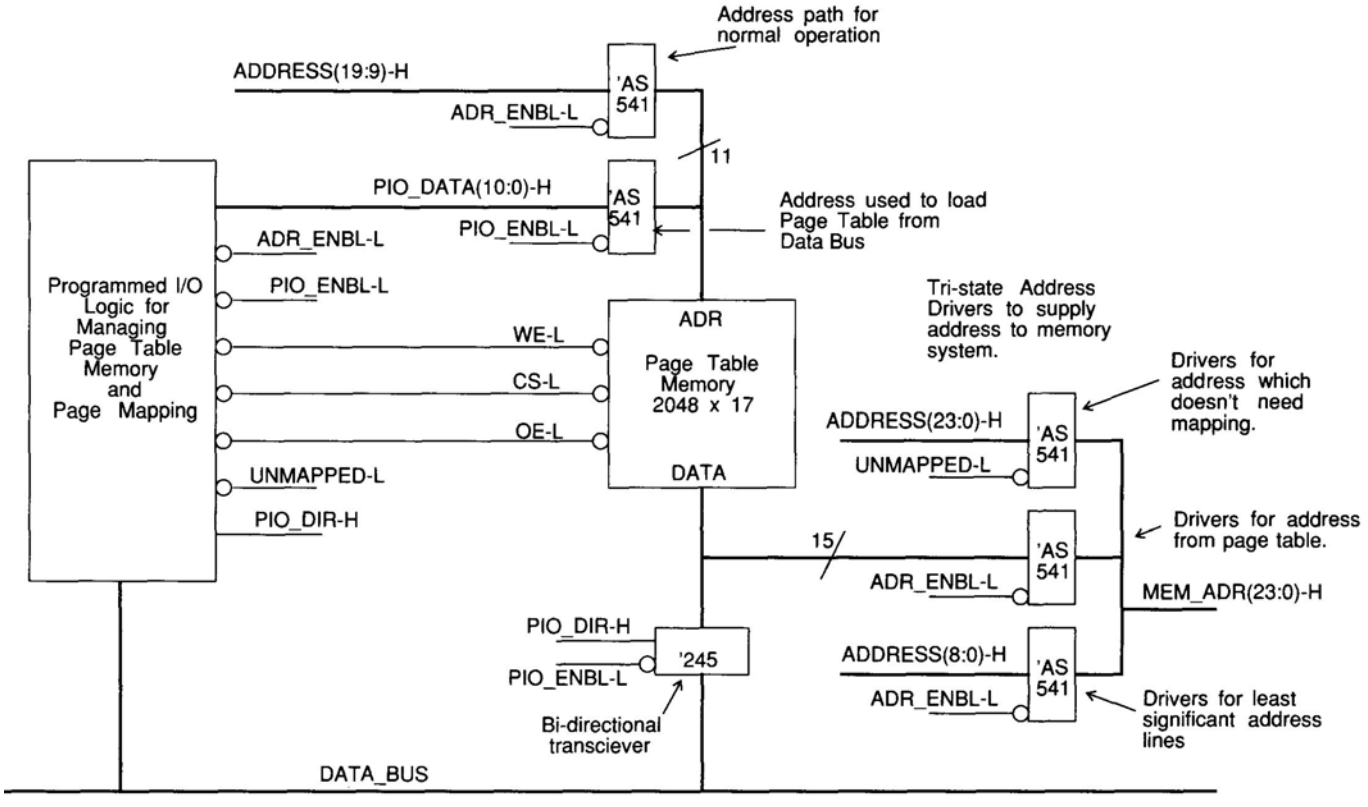
A virtuális memóriakezelés két típusa:
- lapozás
- szegmentálás

LAPOZÁS:
Lap: a program és a memória is fizikailag egyenlő méretű darabokra van osztva (fizikai határok). A főprogram, szubrutinok, globális adatok, lokális adatok, verem mind-mind egy-egy azonos méretű lapnak felelnek meg a memóriában. A lapozás gazdaságtalan, mert a kis méretű adatot is egy nagy méretű lapba kell tölteni, nem használja ki a rendelkezésre álló lapméretet.
Fizikai Cím=Lap Báziscíme + Lap eltolás (offset) címe. A „+ „itt *konkatenációt / összefűzést*, nem pedig összeadást jelent. Ezáltal sokkal gyorsabb lesz a fizikai cím leképezése, mivel nem kell összeadót használni, a konkatenációt egyszerű huzalozással megoldhatjuk. A hexadecimális címek egy-egy számjegyét 4 bites bináris számmá kódoljuk, és összefűzzük.
Pl: Logikai cím a virtuális memóriában: 2,344= 2. lap száma és a lap 344. rekeszcíme a virtuális memóriában. Legyen a 2. lap címe 2C00. Ezt kell összefűzni a 344-el.

2	C	0	0	+	3	4	4	= 2	F	4	4	(Fizikai Cím)
0010	1100	0000	0000	+	0011	0100	0100	= 0010	1111	0100	0100	

Az OS minden futó processhez hozzárendel egy laptáblát. A laptáblából ki a lap számát (hexa érték) követően a hozzá tartozó lapcímet (hexa), továbbá az egyes lapok elérési információit: R/W/X: olvasható /írható /végrehajtható. A fizikai címet a CPU határozza meg. A lapok kis egységek, nagy laptábla kell, célszerű őket a memóriában tartani.
Itt a lapcímeken az összeadás helyett konkatenációt használunk: időt nyerünk (a fenti lapcímben a legkisebb helyiértékű bitek csupa nullák, azokat nem kell hozzáadni; amikben pedig nem csupa nulla szerepel, azokat egyszerűen egymás után fűzzük). A lapok mérete általában mindig kisebb, mint a szegmensek mérete, ezért a kisebb méretű lapok száma több kell, hogy legyen (pl 1024)

Lapozásos címfordító áramkör:



A lapok mérete 512 byte-os, a laptábla 2.048 bejegyzést (lapot) tartalmaz.
A megcímezhető max memória 512x2048= 2^20. (1Mbyte) 20 bit a virtuális címtartomány.
A cím lapcímből és lapon belüli eltolási címből tevődik össze. Egy lap 512 (2^9) byteos = 9 bitet használunk (ADDRESS (0:8)) a lapon belüli hely azonosítására. További „fennmaradó" (2048 = 2^11) 11 bitet a megfelelő lap címének azonosítására (ADDRESS (9:19)). A laptábla tehát 2.048x17 nagyságú, 17 bittel címezhető (17 = 15 bit a címzésre + 2 státuszbit). A címfordítás csak 15 bites. Az egyik státuszbit jelzi, hogy a lap a főmemóriában található-e, ill. a másik jelzi, hogy a betöltés óta módosult-e a lap.
2 mód:
- laptábla adatokkal feltöltése: a vezérlőjelek: CS:Chip Select, WE: Write Enable hatására PIO_DATA-n keresztül adatokkal töltjük fel a laptáblát (11 bites)
- normál viselkedés: A 9 legkisebb helyiértékű címvonalat (ADDR:8:0) kapjuk közvetlenül a címbuszról. Mivel összesen 24 fizikai címvonalunk van a maradék 15 vonal a laptáblából jön. 11 bites címvezetéken keresztül azonosítjuk a megfelelő lap helyét a laptáblában. A laptábla kiválasztja a kívánt lap báziscímét és az OE: Output Enable vezetékre rakja. Konkatenációval megkapjuk a fizikai címet. (A 11 magasabb helyiértékűt a 9 alacsonyabb helyiértékűvel összefűzve)

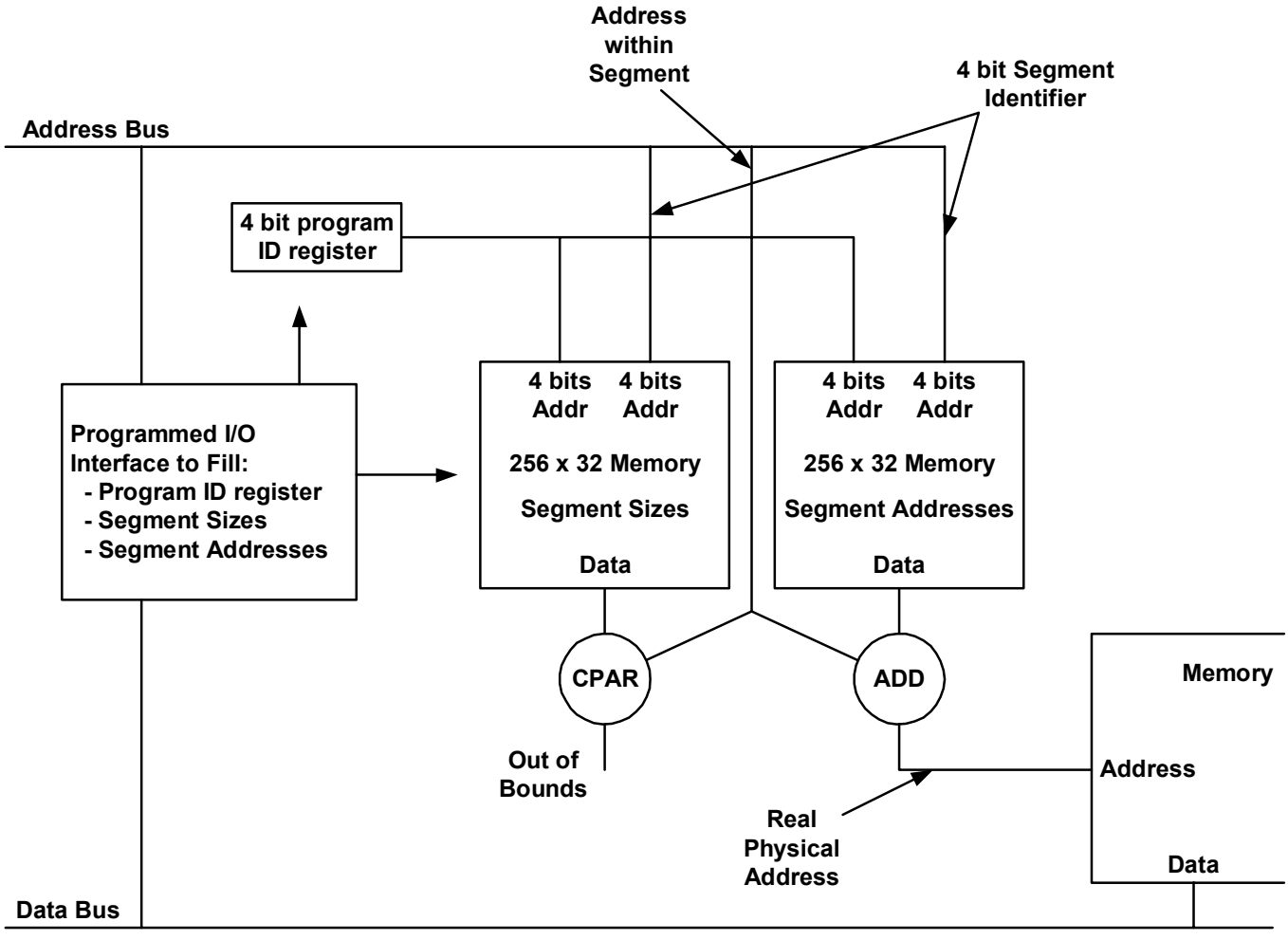
SZEGMENTÁLÁS:

Szegmens: a program változó méretű logikai egységekre van feldarabolva. A program négy fő szegmense funkciójuk szerint:

- 0. szegmens: főprogramok (olvasható/végrehajtható),
- 1. szegmens: szubrutinok,
- 2. szegmens: csak olvasható adatok,
- 3. szegmens: olvasható / írható adatok.

A program a memóriát közvetlenül nem érheti el, csak a szegmensen keresztül.
FIZIKAI CÍM= Szegmens bázis címe + Szegmensen belüli eltolás (offset) címe
A „+" itt ténylegesen *összeadást* jelent! HW-es megvalósítás kell az összeadó miatt, és kell egy összehasonlítás, azért hogy nem címeztünk-e ki a szegmens címtartományán kívülre! Hiba esetén megszakítást (interrupt) küld.
Szegmenstábla: A szegmenstáblából olvassuk ki a szegmens számát (hexa érték) követően a hozzá tartozó szegmens hosszát, szegmens címet (hexa), továbbá az egyes szegmensek elérési információit: R/W/X: olvasható /írható /végrehajtható.
Pl: Logikai cím a virtuális memóriában: 0,154 = 0 szegmens (főprogram) száma, és a szegmens 154 offsetcíme a virtuális memóriában. Legyen a 0.szegmens hexa címe 1310. Ezt kell összeadni! a hexa 154-el.
1 3 1 0 + 1 5 4 = 1 4 6 4 (Fizikai Cím)

Szegmentálásos címfordító áramkör:



Az áramkör 16 programot tud kezelni, amelyek egyenként 16 szegmensből állnak (16x16=256 location). Az áramkör a címgeneráló és a memória között helyezkedik el. 32 bites címvonalat használ. A szegmenscímet párban generálja: áll egyrészt egy szegmensszámból és egy szegmensen belüli eltolásból (offset). Program futása előtt a CPU betölti a megfelelő szegmenset a memóriába, majd beállítja a *szegmens hosszát és címét* a két 256x32-es memóriába (az egyik a szegmens méretét, a másik a szegmens címét tárolja). Ezután az OS beállítja a megfelelő mintát a ProramID regiszterben, és inicializálja/elindítja a programot. A ProgramID egy 4 bites regiszter - a 16 különböző program egyidejű tárolásakor - bizonyos minták alapján kapcsol a programok között. A memóriában a szegmenset a „báziscímének" és a „szegmensen belüli eltolás címének" *összeadásával* kapjuk meg, mindig összehasonlítjuk a szegmensen belüli címet a maximális címmel. Ha túl nagy címet kapunk, akkor egy „out of bounds" (határon kívül vagyunk) megszakítást küld.

ADDR(REAL)= ADDR(BASE)+ADDR(OFFSET) /Ez a formula általánosan igaz a lapozásra és szegmentálásra is !!!/
Szegmentálás hátránya: ha sokszor cserélgetjük a szegmenseket, akkor a változó blokkmérete miatt üres helyek, hézagok keletkeznek, ami a memória rossz kihasználásához vezet, mivel a kisméretű üres területekre már nem lehet behívni újabb szegmenset. Ezt nevezzük *külső töredezettség*nek. A szegmenstábla kezelése az OS feladata, és kell HW-es támogatás is, az összeadó miatt.

CACHE memória:

CACHE: kisméretű, de nagy sebességű memória, amely a processzor és a főmemória között helyezkedik el. Célja a műveletek nagysebességű végrehajtása. Működése hasonlatos a virtuális memóriáéhoz, csak az aktuálisan használt adatokat (LRU technika) tárolja cache-ben. A cache a program (felhasználó) számára rejtett, a program nem tudja, hogy használja-e, csak azt érzékeli, hogy gyorsabb a végrehajtás.

A cache tároló legfontosabb jellemzői a következők
- *a cache-tár mérete*, a blokk mérete (az adatcsere a főtár és a cache között mindig blokkos formában történik),
- egy blokk kikeresésének *eljárása* a cache tárban,
- *aktualizálási eljárás*, amely szerint a processzor által módosított adatot a cache-tárba és a főtárba írjuk,
- a megfelelő *helyettesítési* stratégia (replacement policy), amivel eldöntjük, hogy a cache-ben melyik blokkot lehet felülírni, ha új blokkot kell bemásolni,
- a főtár és a cache-tár *adategyezőségének* biztosítása.
- **Cache hit:** Ha processzor olyan adatot igényel, mely a cache-ben megtalálható, akkor találatról vagy cache hit-ről beszélünk. A találatot a cachevezérlő azzal állapítja meg, hogy a bemásolt blokkokhoz tartozó címrészek (toldalék vagy „tag") alapján valamelyik cache blokkban benne van-e a processzor által igényelt adat főtárbeli címe.
- **Cache miss:** Ha a processzor által igényelt adat nincs meg a cache-ben, ezt tévesztésnek vagy cache miss-nek nevezzük.
Blokk (def) a központi tár rekeszeinek egymás utáni tömbje, érvényesül a „lokalitás" elve.

További fontos paraméterek (képletek):
- T_{CA} cache memória elérési idő
- T_{MS} főtár (main store) elérési ideje
- T_{EFF} effektív elérési idő
- h: cache memória „hit rate"-je

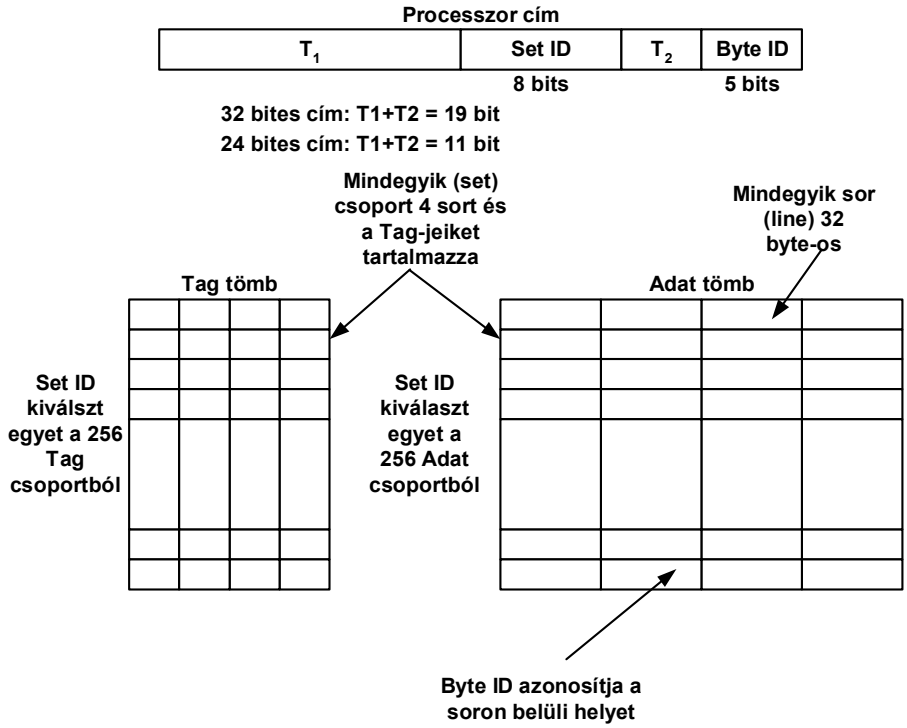
$$T_{EFF} = h \times T_{CA} + (1-h) \times (T_{CA} + T_{MS}) \rightarrow T_{EFF} = T_{CA} + (1-h) \times T_{MS}$$

$$S = \frac{T_{MS}}{T_{EFF}} \text{ speed up.}$$

Cache tartalom visszaírási stratégiák:
- Write through: Adat megváltozásakor mind a cache, mind pedig a főmemória aktualizálódik, frissül
- Write back: Az adat csak igény esetén aktualizálódik, frissül a főmemóriában (dirty bit használata)

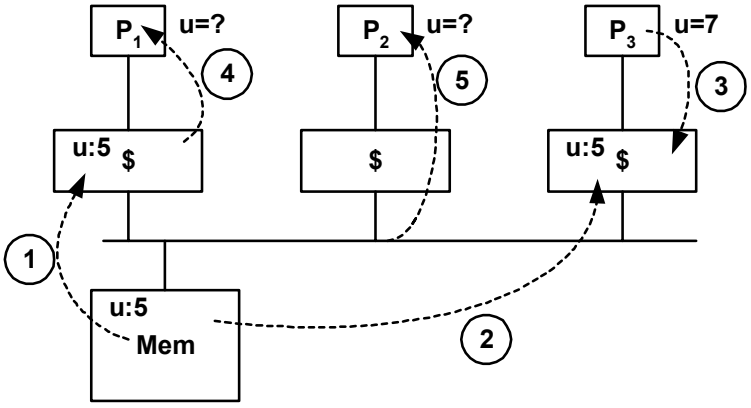
Asszociatív Cache memóriák fajtái:
- *közvetlen leképezésű:* a központi memória minden egyes blokkja csak egy sorban (rögzített / direct sorindex) szerepelhet a cache-ben. Alacsony hit rate
- *teljesen asszociatív:* a központi tár bármelyik blokkja a cache-be bárhova betölthető, a blokk címe pedig bekerül a cache toldalék részébe. Gyors és Drága hw-t igényel
- *n-utas csoport asszociatív:* Olyan cachetároló, amely több, "n" sorból álló csoportokra van osztva, és az egy csoporthoz tartozó cachetárolórész önmagában teljesen asszociatív tárolóként működik. Annak megállapítása, hogy egy cachebe írandó blokk melyik csoporthoz tartozik, a közvetlen leképezésű cachehez hasonlóan, a memóriacímből képzett indexszel kerül meghatározásra. (előző két módszer ötvözete)

Pl. 4-utas csoport asszociatív cache:

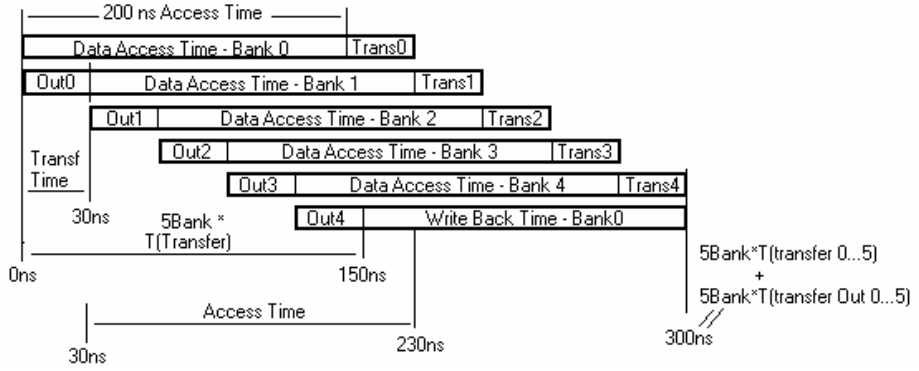


Cache Toldalék (TAG): A cache sorának cím- és vezérlési adatokat tartalmazó része.
Részei:
- címrész: főtárból bemásolt blokkra vonatkozó címinformációk;
- vezérlőrész: a cacheblokk adataira vonatkozó érvényességi információk bitenként kódolva.

Cache koherencia probléma:
Osztott cache használat esetén léphet fel, amikor a P processzorok saját $ cache memóriával rendelkeznek. Probléma abból adódhat, amikor ugyanazon osztott memória blokk (u location) tartalma megjelenik egy vagy több processzor saját cache memóriájában, és például történik egy írási tranzakció (egyik processzor módosítja a főmemória tartalmát), a többi processzor pedig még a régi cache-beli tartalommal (másolat) dolgozik, tehát nem aktualizálódnak a cache memóriák értékei.
Példa:



CACHE szervezése memóriabankokkal:
Legyen T(translation) = átviteli idő 30ns, T(access)= adat-hozzáférés (írás v. olvasás) 200ns.
Változó memória elérés, a cachet alkotó memóriabankok külön-külön címezhetőek és hozzáférhetőek egy közös buszon keresztül. A buszrendszer alkalmazása felgyorsítja a folyamatot, és a memóriabankok használatával nő a sávszélesség. A bankok tartalmának címzése időben elkülönül egymástól (interleaving technika). A memóriakérést az összes bank megkapja, majd a megfelelő bank a válaszát egy adott időrésben belül küldi vissza. Egy buszvezérlő a bankokhoz kapcsolódó interface-n keresztül kezeli a bankokat, amely segítségével biztosítja az információátvitelt.
300ns alatt az 5. Memóriabank (Bank_4) is elérhetővé válik a buszon keresztül, és a cache-be történő átvitel is befejeződik.



Egyszerűsített Példa:
T(access)=200ns egy-egy memóriabank elérési ideje (írás / olvasás)
T(translation) = 30ns átviteli idő
Mennyi időt vesz igénybe az 5. memóriabank írásának/olvasásának folyamata?
Megoldás: 5*T(access)+1*T(translation) =5*200+1*30 =1030 ns

(**Cache memóriákról részletesen:** cachecoherence.pdf illetve a fóliákon!)

14. Input / Output egységek, buszrendszerek:

A számítógép a külvilággal, perifériákkal az I/O egységeken keresztül tartja a kapcsolatot. Az információ továbbítását az egységek között buszok végzik, amelyek interfészekkel kapcsolódnak egymáshoz. **Interface**: azon szabályok összessége, amelyek mind a fizikai megjelenést, kapcsolatot, mind pedig a kommunikációs folyamatokat leírják. Egy busz általában 3 fő kommunikációs vonalból áll: *vezérlőbusz, adatbusz, és címbusz*.
Kommunikáció során megkülönböztetünk egy eszközpárt: a *Master*-t és *Slave*-t. A Master (pl CPU) általában, mint kezdeményező, birtokolja és ellenőrzi a buszt, és átadja (írás / olvasás) az adatokat a Slave-nek (pl. Memória). A kommunikációhoz előredefiniált **protokollokra** (szabályok és konvenciók gyűjteménye) van szükség, amelyek meghatározzák az események sorrendjét és időzítését. A kommunikáció feltétele a másik egység állapotának pontos ismerete. Lehetséges több M-S modul (pl multimaster-rendszer több kezdeményezővel) is egy rendszerben. **Két alapvető protokollt különböztetünk meg:**

- **A.) Aszinkron kommunikációs**
- **B.) Szinkron kommunikációs**

A.) ASZINKRON PROTOKOLLOK:

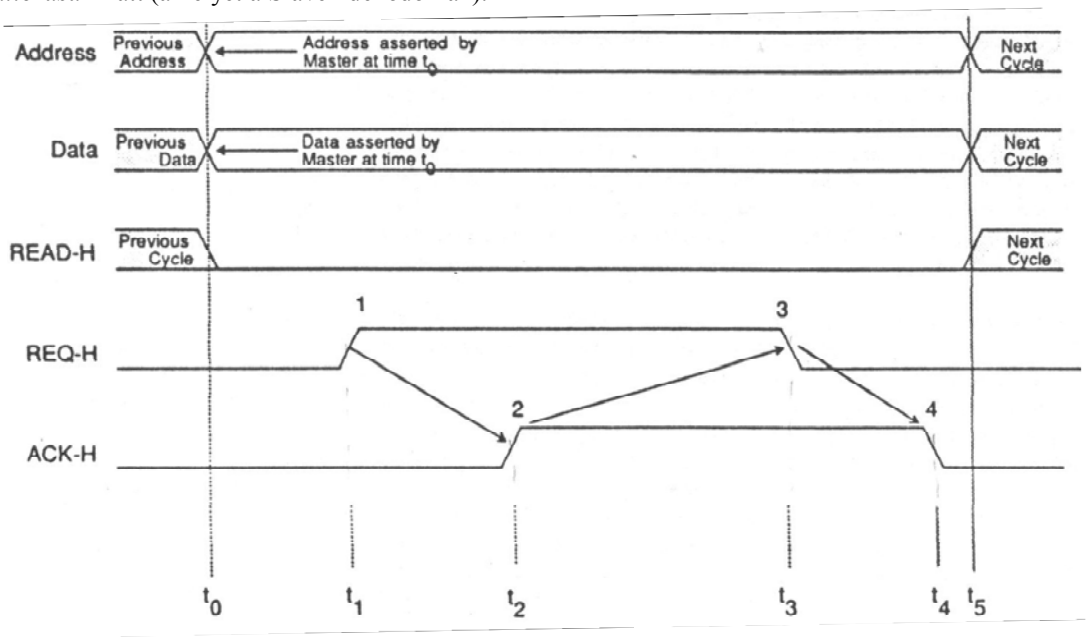
Ebben az esetben a Master-Slave modulok nem használnak közös órajelet (CLK-t): ha az egyik egység végez, elindít egy másik tranzakciót. A Master, mint kezdeményező, aktiválja a megfelelő vezetékeket. A Slave válaszol. A Slave címének azonosítására egy külön egység szolgál. Vezérlőjelekkel zajlik a kommunikáció: írási / olvasási folyamatokat különböztetünk meg. Ilyen vezérlőjelek lehetnek a READ, REQUEST, ACKNOWLEDGE (ha READ-H = Master olvas a Slave-ről, ha READ-L= Master ír a Slave-re)
+ egyszerűen felépíthető, nem kell (közös) órajel, gyors átvitelt biztosít (modulok sebességétől függ)
- beépített késleltetések, csak korlátozott hosszúságú buszok /vezetékek lehetnek.

Handshaking protokoll: (kétszeres „kézfogás" – 4 fázisú kommunikáció)

Aszinkron protokoll: a buszrendszer moduljai nem közös órajellel működnek, hanem vezérlő jelek segítségével. Háromféle buszvonalat ismerünk: *cím-, adat- és vezérlő-* buszt. A master által *címvonal*ra rakott cím (kezdeményezés) egyértelműen meghatározza a tranzakció célállomását. Meghatározott idő áll rendelkezésre, hogy a Slave modulok összehasonlítsák saját címükkel a célcímet. Ha megegyezik, akkor válaszol a Master-nek, és a tranzakciót a *vezérlővonalak* megfelelő beállításaival szabályozza. A vezérlővonalakat tehát a Master-Slave közötti kommunikáció szinkronizálására használjuk. Ezt nevezzük **handshaking** protokollnak. Címvonalak csoportjából (Address), adatvonalak csoportjából (Data), és három vezérlőjelből (READ-H, REQ-H, ACK-H) áll. Ha READ-H magas, akkor a Master olvas a Slave-ről, ha alacsony, akkor ír a Slave-re. A REQ (kérés) és ACK (nyugtázás) vezérlőjelek határozzák meg az események időzítését és pontos sorrendjét.

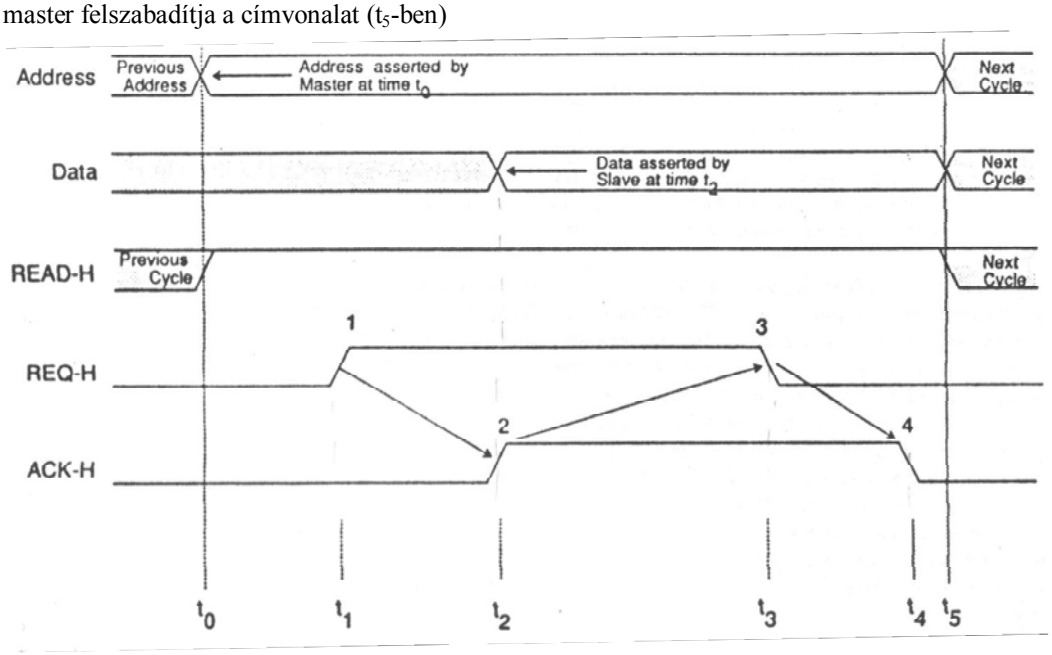
WRITE ciklus: „a Master modul ír a Slave-nek"

- t_o-ban a master (amely az arbitráció után már megkapta a busz vezérlését) megadja a kívánt slave címét.
- véges idő kell hogy a jel a slave modulokhoz érjen, azok dekódolják a címet, ezért a master vár bizonyos ideig mielőtt beállítja a request vonalat t_1-ben. (*skew time*= a slavekhez elsőként ill. utolsóként érkező címek közötti időkülönbség)
- ezt a request jelet minden slave veszi ugyan, de csak az fog válaszolni, akinek a címe megegyezett a master által kért címmel.
- amikor a slave megkapta az adatokat a mastertől, nyugtázza t_2-ben.
- a master megkapja a nyugtát, így tudja, hogy az átvitel megtörtént, ezért felszabadítja (alacsony állapotba helyezi) a request vonalat t_3-ban.
- ezt (a request felszabadítását) érzékeli a slave, és felszabadítja a nyugtázó vonalat t_4-ben
- a master a címvonalat tartja még egy bizonyos ideig (t_5-ig) a request vonal felengedése után is, a *cím esetleges megváltozása* miatt (amelyet a Slavek dekódolnak).



READ ciklus: „a Master modul olvas a Slave-ről”. Nagyon hasonlít az írási folyamathoz, de abban különbözik, hogy a READ-H jel magas szinten van, és az adatvonalat a slave állítja be. Ez jelenti az olvasást.

- t_o hasonlóan, master (amely az arbitráció után már megkapta a busz vezérlését) megadja a kívánt slave címét
- t_1- ben a master a request vonal beállításával a megcímzett slave-től adatot kér, olvasni szeretne
- t_2-ben veszi a kérést a slave, nyugtázza és beállítja magas szintre a nyugtázó vonalat.
- t_3-ban a master megkapja az adatot a slave-től, felszabadítja a request vonalat.
- t_4-ben érzékeli a slave, hogy a master felszabadította a requestet, ezért így ő is felszabadítja a nyugtázó vonalat.
- végül a master felszabadítja a címvonalat (t_5-ben)

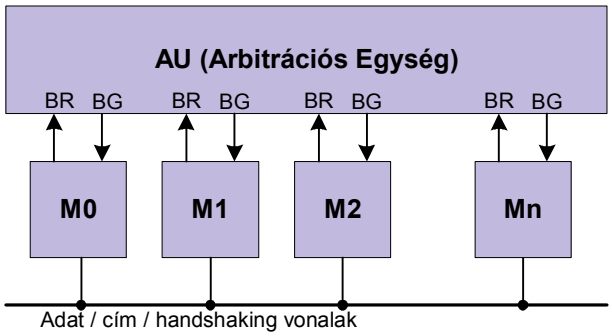


Arbitráció:

Egy tetszőleges I/O művelet esetén (aszinkron v. szinkron buszos átvitel) több Master (commander) egység is meg akarja szerezni egyszerre a busz irányítását. Különböző előredefiniált algoritmusok segítségével egyértelműen azonosítható, hogy a „versenyhelyzetben” a következő átvitelt (a következő ciklusban) melyik Master fogja megvalósítani. Az *arbitrációs eljárás* egy döntési folyamat (mechanizmus), amely az aktuális adatátvitellel párhuzamosan zajlik le, és még az adatátvitel befejezése előtt eldől, hogy melyik következő master adhat. A Mastereket M_1...M_n-el jelöljük, a BR: Bus Request (busz kérése, igénylése) a Master által, míg a BG: Bus Grant (igénylés elfogadása, engedélyezés) jelet a központi AU: Arbitration Unit (arbitrációs egység) bocsátja ki.

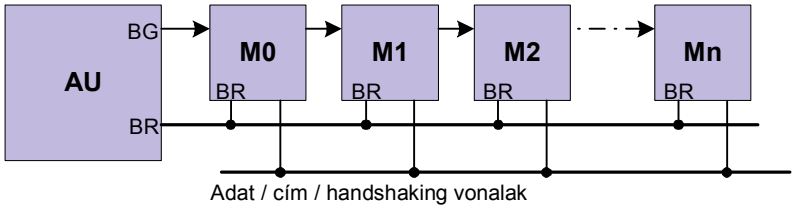
Az arbitrációnak 3 típusa van: párhuzamos, soros (daisy chain), lekérdezéses (polling).

a.) *Párhuzamos:*



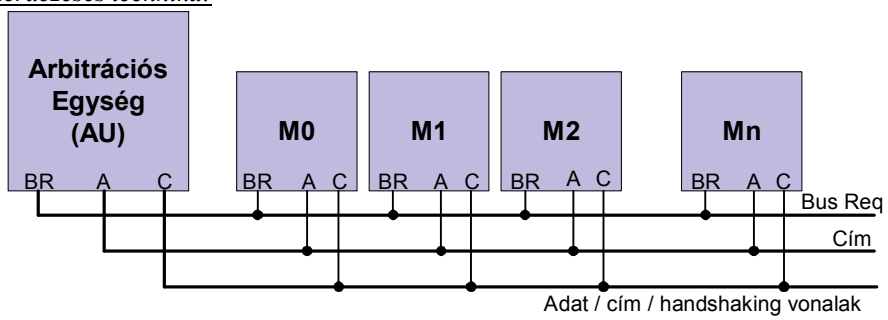
Ez a leggyorsabb módszer, minden M-nek kitüntetett, egyenrangú kapcsolata van az AU-val (két vonalon: BG és BR-en). Az arbitráció lehet *(i) first asserted/first served (FIFO),* vagy (ii) *round robin*, vagy (iii) *prioritásos alapú.* Ezek a módszerek többfajta lehetőséget biztosítanak mind egyszerű, mind pedig komplex esetben. Az AU vezérli a közös buszon az adatátvitelt. Az adat-cím-handshake vonalat a kijelölt master kezeli, az adás befejeztével pedig kiadja ismét a BR-jelet. Hátránya, hogy drágább, mint a többi módszer (a párhuzamos ágak miatt minden M-hez 2 vonal kell), és az M-ek száma korlátozott.

b.) *Soros „Daisy Chain”:*



A BG vonal sorosan van kötve, míg a BR vonal mindegyik M-hez csatlakozik. Az AU nem ismeri, hogy pontosan melyik M kívánja elérni a buszt, ezáltal az átvitel leegyszerűsödik: csupán azt tudja, hogy bizonyos ciklusonként BG-jelet kell kibocsátania. A soros csatlakozás miatt a legelső Master (M_o) rendelkezik a legnagyobb prioritással (*fizikai prioritás*), így ha ő igényelt, minden esetben megkapja a buszt. Bármennyi eszközt is sorba köthetünk, nincs felső korlátja. Hátránya, hogy az arbitrációs idő (ha a legutolsó M igényel és az előtte lévők nem) egyenes arányban van a sorba kapcsolt M-ek számával.

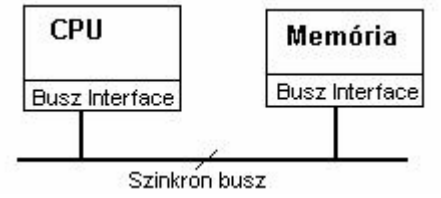
c.) *Polling / Lekérdezéses technika:*



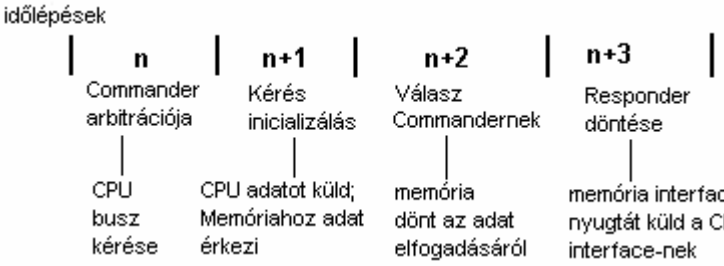
Mindegyik Master egy közös BR buszon igényelhet, és az AU eldönti, hogy melyik Master-től kapta a kérést. Annak a címét az *address* vonalra rakja. Minden ciklusban megnézi az igényléseket, és a legmagasabb prioritással rendelkezőt fogadja el. Itt is többféle prioritásos módszer megvalósítható: pl. (FIFO, round robin). Hátránya, hogy nagyobb az időszükséglete a párhuzamosnál, így ritkábban alkalmazzák (pl: I/O kérések arbitrációjánál), segítségével a processzor egy program futtatásakor lekérheti az I/O eszközöket, megszakítást engedélyezhet.

B.) SZINKRON PROTOKOLLOK:

+ Közös órajellel működnek a buszra csatlakozó egységek. Tehát a műveleti időt az órajelciklus határozza meg. Mivel nincs szükség párbeszédre (pl. handshaking), gyorsabb az aszinkron működésénél. Jel: Master=Commander, a Slave=Responder.
- Az órajelet mindig a leglassabb (legtávolabb lévő) egységhez kell igazítani.



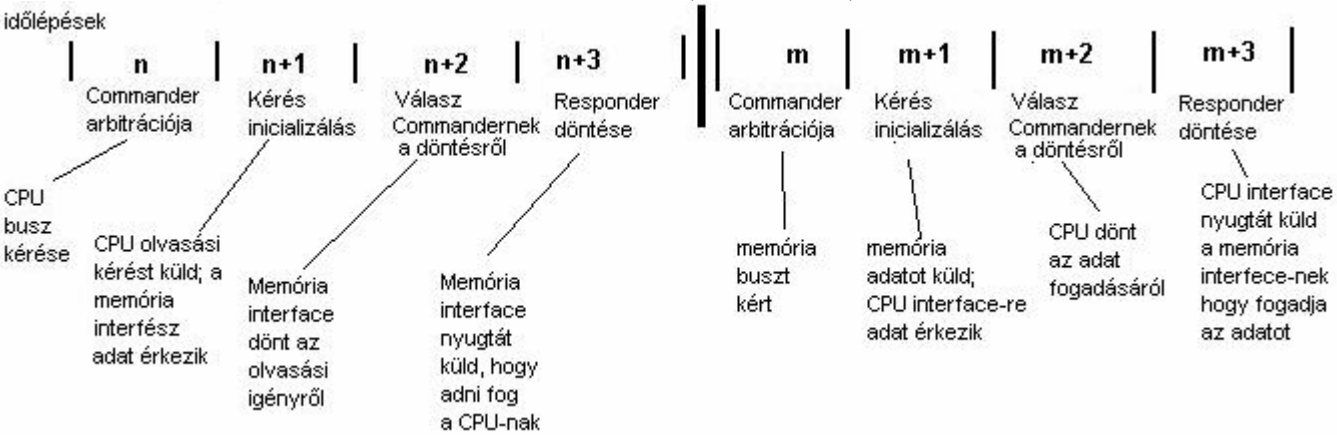
Írási ciklus: (CPU ⇒ MEM)



Szinkron adatátvitel 4 fő lépése:

1. Commander arbitrációja, kiválasztása
2. átvitel megkezdése (de még nem fogad)
3. válasz a kérésre, döntéshozás
4. nyugta, Responder veszi az adatot

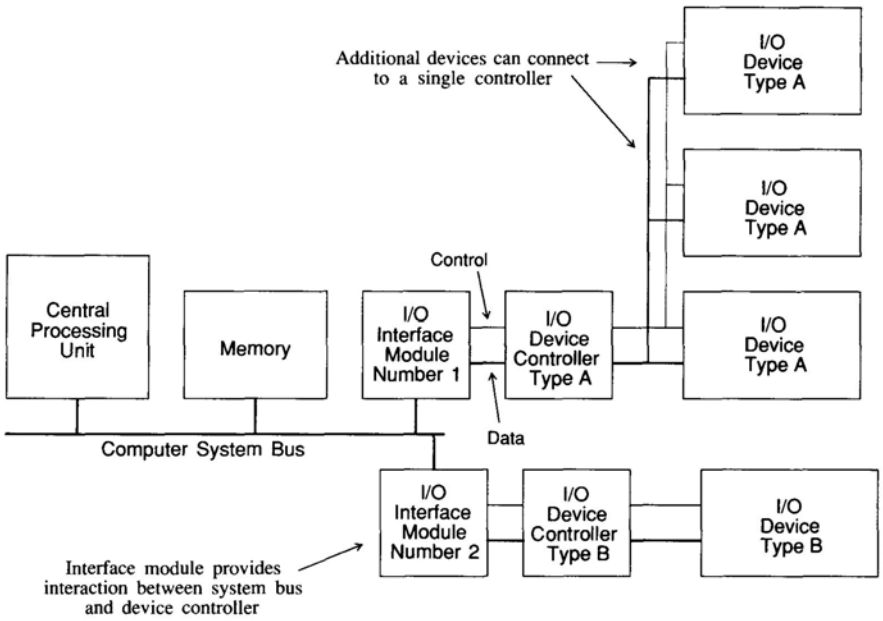
Olvasási ciklus: (CPU ← MEM)



I/O interfacek fejlődése:

1. CPU által közvetlenül vezérelt
2. I/O kontrollerrel vezérelt modul: a CPU programozott I/O utasításokat használ, (még megszakítások nélkül)
3. Interruptos kontroller: megszakítás vezetékekkel egészül ki
4. DMA- közvetlen memória elérésű, saját cpu-val rendelkezik, tehermentesíti a CPU-t, perifériát közvetlenül eléri
5. I/O csatornák: az eszköz saját utasításkészlettel rendelkezik, I/O program a memóriában van
6. I/O processzorok: saját elkülönített RAM-al rendelkeznek (nem a központi memóriát használja)

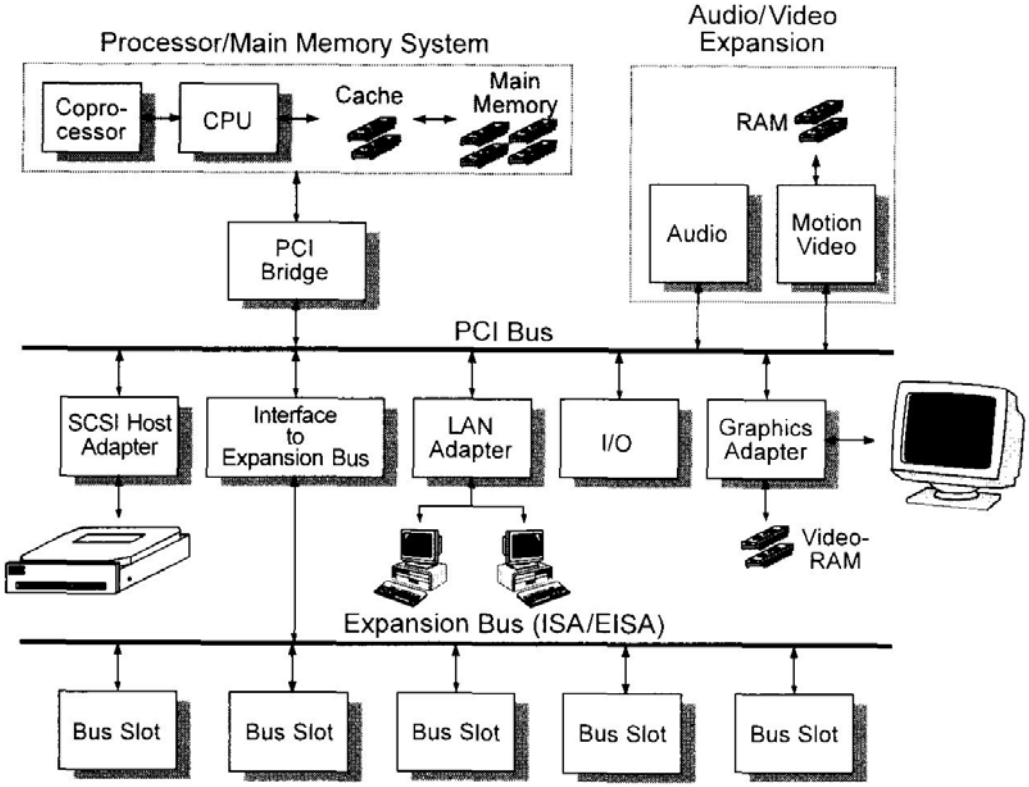
Programozott I/O interfész blokkdiagramja:



Két fő részre osztható: 1. rendszerbusz (CPU, memória) 2. I/O eszközvezérlők a különböző típusú I/O eszközökkel tartják a kapcsolatot. A rendszerbuszt és az I/O eszközvezérlőket az I/O interfacek kapcsolják (illesztik) össze. Az eszközök gépi kódú utasításokkal (assembly) vezérelhetők. Egy új I/O eszközt (A v. B típusú) a megfelelő típusú eszközvezérlőbe kell bekötni.

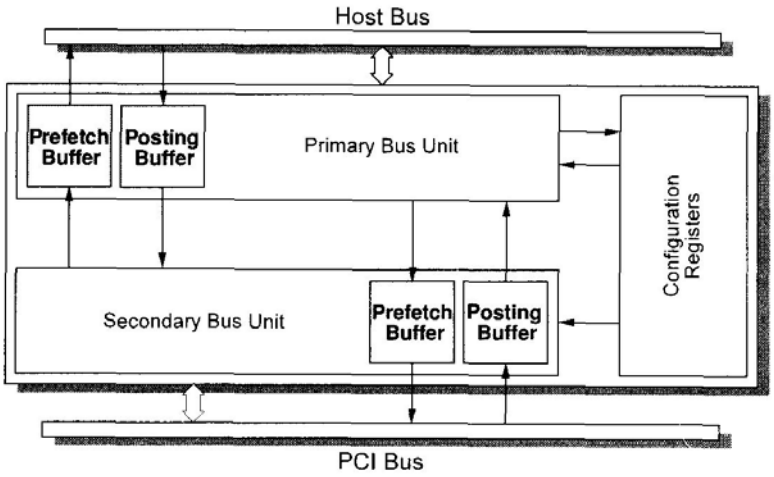
BUSZRENDSZEREK: ISA, MCA, EISA, VESA LB, PCI, SCSI, AGP…

PCI busz: PCI= Peripherial Component Interconnect. Teljesen új, a korábbiaktól önálló szabvány (Ver 2.1): 33MHz-es órajel támogatás. Nem függ a processzortól, újabb rendszerekben is felhasználható. Kapcsolatot a processzor és a PCI sin között egy PCI HOST BRIDGE biztosítja. Támogatja a többprocesszoros rendszereket, kompatibilis az ISA, EISA, MCA rendszerekkel. A PCI csatlakozóhelyekre speciális intelligens kártyák helyezhetők, amelyek képesek önálló adatátvitelt végrehajtani a processzor tehermentesítése céljából, így gyorsabb működés érhető el. (3.3 – 5V szabvány). /Csatlakoztatható SCSI-, hálózati-, hang-, videó-kártya./ Konfigurálása szoftveres úton, a BIOS-on keresztül történik. / Arbitrációs mechanizmust használ.



A PCI 32 bites multiplexált adat/cím jeleket alkalmaz (64 bites változata is létezik, főleg szerverekben). Régebbi alaplapokon az ISA ill. AGP bővítőhelyek mellett általában 3-4 PCI slot volt található. Maximális átviteli sebessége (4-es **burst**onként / löketszerűen) 132Mbyte/sec (4 byte*33MHz=132Mbyte/s). A külső zaj, zavarok elkerülése végett, a PCI elemeket rövid úton kell összekötni, így a PCI jelek egy oldalán vannak kivezetve (PCI Speedway), sok földelést használva. Saját POST (Power On Self Test) önellenőrző kóddal is rendelkezik, a hibák felderítése végett, ami a számítógép bekapcsolásakor inicializálódik.

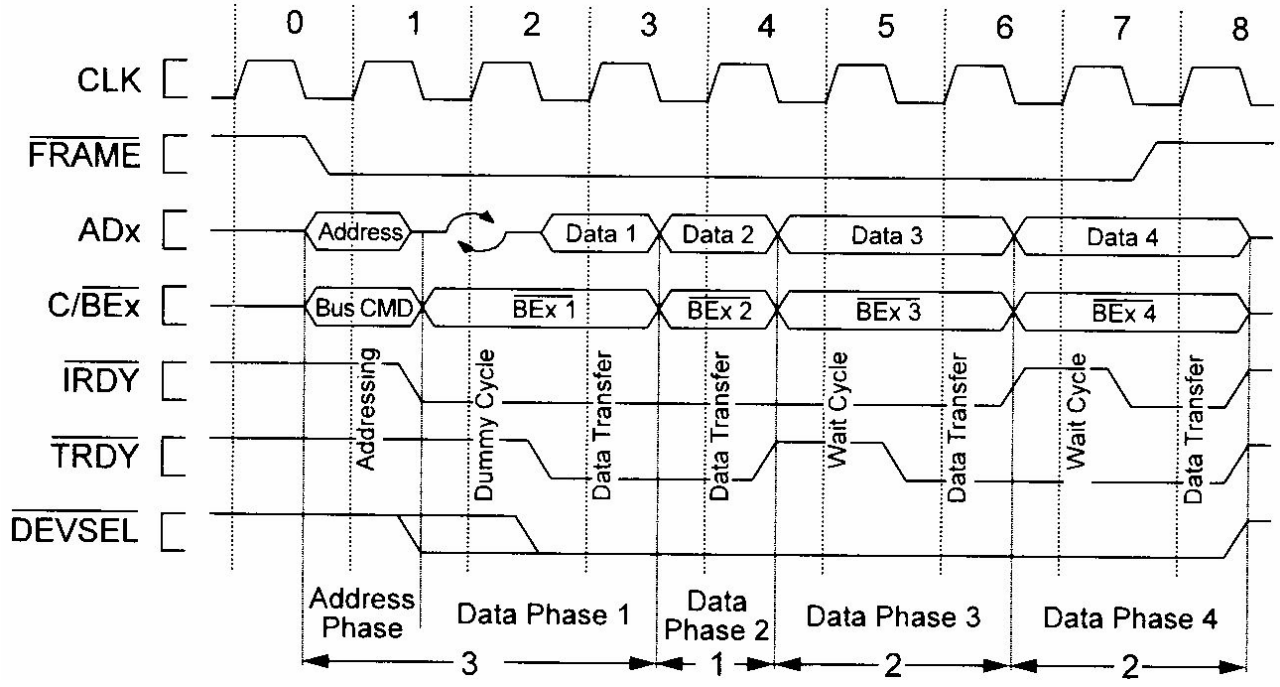
PCI Bridge:
A PCI BRIDGE a PCI buszt köti össze a rendszerbusszal (CPU LOCAL BUS –al és MEMORY BUS –al egyben), vagy másnéven a HOST busszal. A konfigurációs regiszter a PCI BRIDGE részét képezi. A Host busz kapcsolódik az elsődleges busz egységhez, amelyben Prefetch és Posting bufferek találhatóak. Ez az egység közvetlenül is tud kommunikálni a másodlagos buszegységgel, amely a PCI buszhoz kapcsolódik. De a két egység megcímezhető a konfigurációs regiszteren keresztül is.



PCI busz jelei (jelcsoportok a PCI csatlakozón):
A csatlakozó 188 lábú, oldalanként 94-94 lábbal. Sok GND található a zavarások elkerülése végett. A tápról jön 12V, 5V, 3.3V-os jel is.
CLK: (Clock) PCI busz órajele (0-20-33MHz)
AD0-AD31: (AD: Address/Data) multiplexált cím/adat vezetékek 32 bites üzemmódban (ahol AD00-AD07 byte-címzés esetén az LSB, míg AD24-AD31 byte-címzés esetén az MSB-t jelöli).
AD32-AD63: 64 bites üzemmódban
IRDY: (Initiator Ready) adatok olvasásakor (negált jel)
TRDY: (Target Ready): adat írásakor (negált jel)
DEVSEL: (Device Select) ez egy nyugtajel, a target ezzel nyugtázza, hogy a címet dekódolta
IDSEL: a Chip Selectnek felel meg, adatírás vagy konfiguráció során lehet hozzáférni a chip-hez
STOP: az adatforgalom megszakítását jelzi a target-nek
FRAME: adatátviteli ciklus jelzése (negált jel)
PAR: (Parity) adatok és címek paritás ellenőrzése
C/BE3 – C/BE0: (Command / Byte Enable) egy-egy jel egy byte-ot foglal magába. Megmutatja, hogy melyik byte tartalmaz érvényes adatot, ill. írunk vagy olvasunk-e.
PERR – SERR: (Parity error ill System error) hibajelek
INTA – INTD: megszakításjelek (felfutó élre vezéreltek)
REQ: (Request) sínhozzárendelés a kéréshez
GNT: (Grant) sínhozzárendelés engedélyezéshez
TCK, TDI, TDO, TRST: (Test Clock, Test Data in, Test Data Out, Test Reset) PCI sín tesztelésének jelei
RST: (Reset) regiszterek tartalmának törlése, és a PCI jeleinek kiinduló helyzetbe állítása

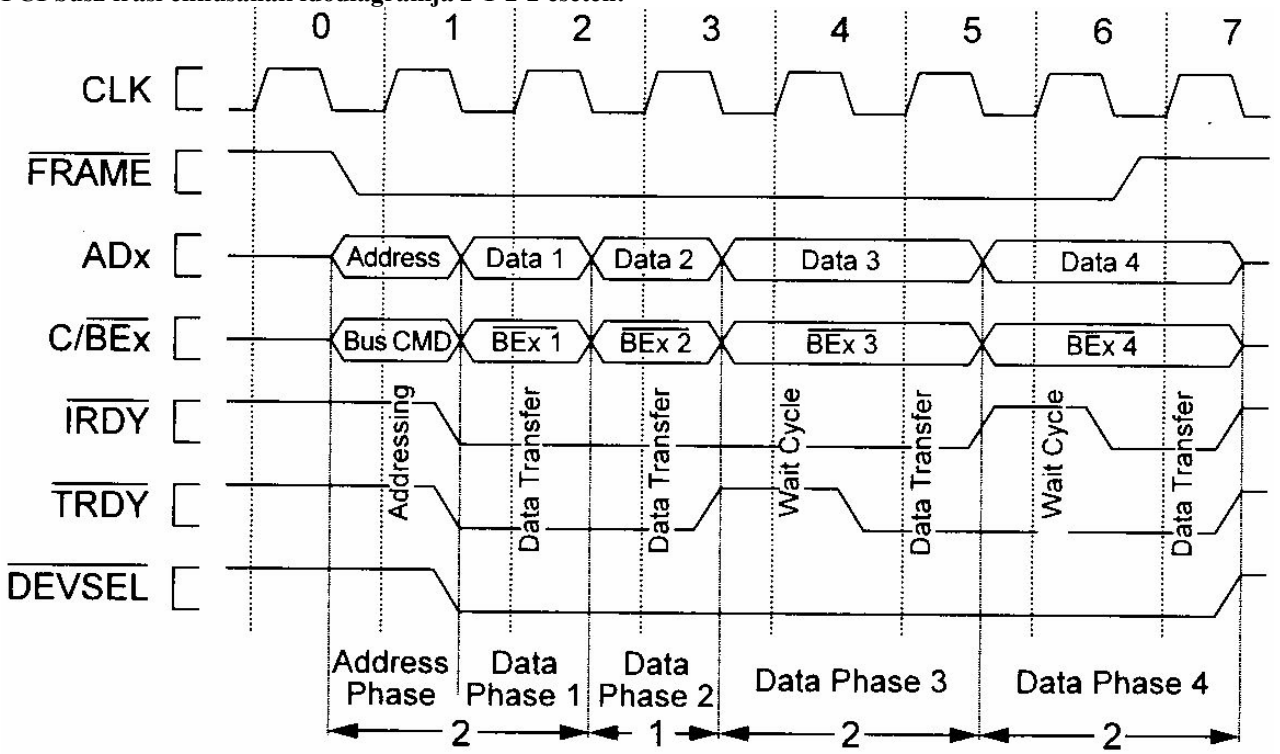
PCI busz (burst=löketszerű) Olvasási ciklusa:
Burst: löketszerűen egyszerre több adatot szeretnénk kiolvasni. Az alábbi ábrákon látható módon 1 cím kiadása után 4 adat jön. A címzési fázis utáni első órajelciklusban az átvitel irányát módosítani kell, a közös multiplexált ADDR/DATA busz miatt. Ezért van szükség egy üres ciklusra (Dummy cycle-ra). Tehát különböző ciklusokkal adható meg, hogy mi fog történni: Pl optimális 3-1-1-1, vagy ***3-1-2-2***. Miután az arbitráció során az egység sikeresen megkapta a vezérlést, a FRAME jelet alacsony szintre állítja, ezzel inicializáljuk az adatátviteli folyamatot. Ezután a címzés fázisa következik, mely egy ciklus idejű, majd egy üres ciklus, végül az első adat megérkezéséhez szükséges ciklus (Esetünkben az első fázis 3 ciklus idejű: címzés + üres ciklus(ok)+TRDY adat). Az IRDY jel alacsony jelszintre váltása után, a TRDY jel is alacsony szintre vált, megkezdődhet adatok továbbítása löketszerűen, egymás után 1-2-2 ciklusonként.
Tehát 3-1-2-2= 3 órajelciklust vár az első adatig, utána 1-2-2 ciklusonként jönnek az adatok (közben wait ciklusok vannak)
(Írásnál nincs üres ciklus, a cím kiadása után egyből mennek az adatok 2-1-2-2 ciklusonként, az optimum 2-1-1-1)

PCI busz olvasási ciklusának idődiagramja 3-1-2-2 esetén:



(PCI busz írási ciklusa esetén a C/BEx jel Command része negáltan, míg Byte-Enable része ponáltan szerepel.)

PCI busz írási ciklusának idődiagramja 2-1-2-2 esetén:



VESA LB: Video Engineering Standard Association – Local Bus
Helyi sin már az AT386-osoknál létezett, 32 bites cím/adatsínnel, és 33MHz-es órajelfrekvenciával. A perifériákkal való kapcsolat gyorsítására született, egy egységes felületet teremtett a kártyák (max. 3 féle kártyát kezel: videó-, merevlemez vezérlő-, háló-kártya) használatához. A VLB a gyorsító-kártyát kötötte össze a processzorral és a memóriával. A VLB nem más, mint a processzor belső buszának „meghosszabbítása”.

PCI-Express v1.0 / 2.0 (fóliákon illetve Addison.Wesley.PCI.Express.System.Architecture.eBook-LiB.chm**)**

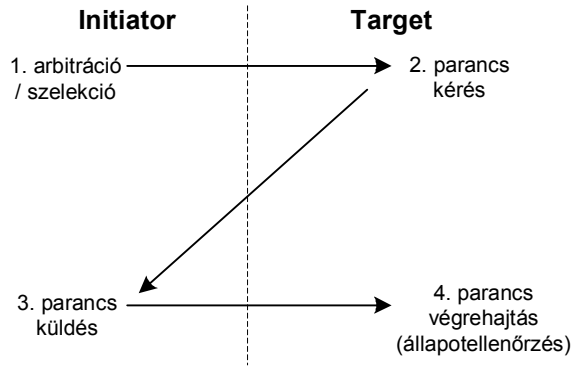
15. SCSI Interface:

SCSI= Small Computer Standard Interface: komplex, intelligens, sínorientált eszköz (merev-, hajlékonylemez, CD-ROM, szalagos egység, scanner...) interface. Különféle perifériák illesztésére, a processzor tehermentesítésére fejlesztették ki, az operációs rendszertől független felülettel rendelkezik. Sokoldalú eszköz, mivel nemcsak PC-s környezetbe, hanem UNIX munkaállomásokba és Apple-Mac gépekbe is beépíthető.
Tulajdonságai: A SCSI a buszok között általánosan 8 eszközt definiál (manapság max. 15 ilyen eszköz csatlakoztatható) legnagyobb sebessége 160Mbyte/s, max buszhossz 12m (UW-160). Csatlakoztatható több belső és 1 külső eszköz is. Az eszközöket egyetlen vezérlő a ***hostadapter*** kezeli, amely a számítógépes rendszer eszközeinek kapcsolatát építi fel a SCSI rendszerrel. Nagysebességű, párhuzamos blokkos átvitelt biztosít a processzor és perifériák között. A szabványos SCSI csatolónak 50 (v. 68) lába van, amelyből 9 vezérlő-, 9 adat-vezeték.

A saját processzorral és memóriával rendelkező intelligens SCSI egységek kezdeményezőként **(*initiator*)** és fogadóként **(*target*)** is működhetnek! Maximálisan 8 initiator, és 8 target működhet egyszerre. A kezdeményező adja ki az utasításokat, a cél pedig feldolgozza, és végrehajtja azokat. Minden egységnek saját különböző címe van (0-7), amelyeket jumperekkel kell beállítani, hogy az ütközéseket elkerüljük. Az egyes SCSI szabványoknak megfelelően hostadapter/vezérlő címe mindig 0 (LSB), nem változik, a szalagos egységé pedig a 7-es (MSB), azonosítójuk SCSI-ID=0 ill. 7.

A SCSI_ID mellett létezik a LUN = Logikai Egység Azonosítószám is: minden targethez hozzárendelhető további 8 logikai egység, amiket az SCSI parancsok esetén saját LUN-nal azonosíthatunk. A kommunikáció 8 bites adatbuszon 1 bites paritás ellenőrzés mellett zajlik. Ha a target lassú, érdemesebb magasabb prioritású szintet állítani neki. A busz elején a vezérlő hostadapter van, a másik végére pedig a lezáró ellenállást mindig az utolsó eszközre kell tenni. Az egységeket egymás után felfűzve, egyforma (50 eres) szalagkábellel kell csatlakoztatni.

A SCSI négy fő kommunikációs fázisa a következő (handshaking):
1. command, 2. data, 3. message, 4. status



A teljes SCSI busz fázisok a következők:
- busz szabad?
- arbitráció / szelekció (ki adjon?)
- üzenetküldés
- parancs elmegy
- adatkapcsolat
- állapotellenőrzés (status)
- üzenetzárás, kapcsolatbontás

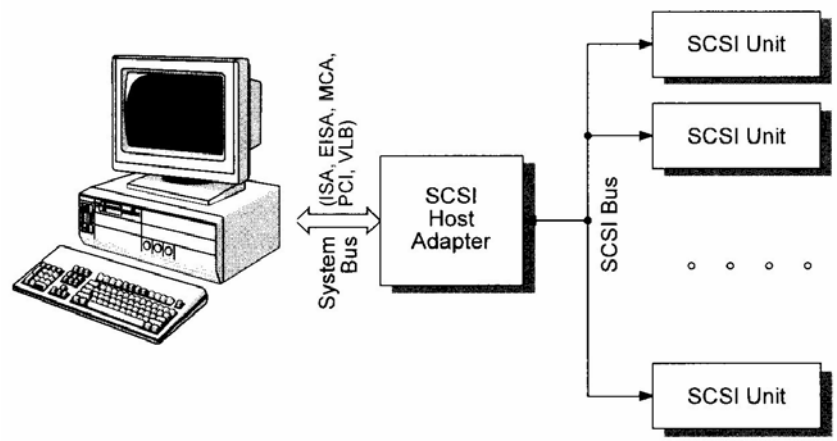
SCSI Vezérlőjelek:
- REQUEST: handshaking parancskérés, target által kezelt
- ACKNOWLEDGE: üzenet nyugtázása az initiator által
- BUSY: buszon a target foglaltságának jelzése (egy eszköz szabad, ha BUSY=0)
- SELECTION: initiator kiválasztja a target-et (SELECTION=0, nincs kiválasztva), (SEL=0 esetén a target újra felépíti a kapcsolatot az initiatorral a busz ideiglenes felszabadítása után)
- C/D: Control /Data: a target által kezelt jel, amely jelzi, hogy vezérlőinformáció / adat információ van a buszon.
- I/O : Input/Output: szintén a target által kezelt vezérlőjel, amely az adatbusz adatforgalmának irányát mutatja
- MSG= Message: az üzenetküldés fázisának jelzésére szolgál, a target által kezelt
- RESET: az összes csatlakoztatott SCSI eszköz „reset-elése”, inicializálása

SCSI szabványok:

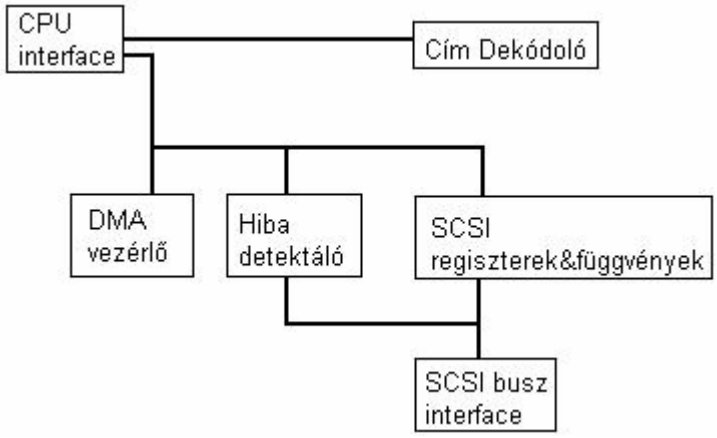
SCSI I: 8 bites busz; átviteli seb. *aszinkron* módban 2.5Mbyte/s, szinkron módban 5Mbyte/s; 5Mhz busz-frekvencia; max 7 egység csatlakoztatható, 50 pólusú csatoló
SCSI II (Fast SCSI): első valódi SCSI szabvány, 8 bites busz; 10Mbyte/s; 10 Mhz buszfrekvencia; max 7 egység; 50 pólusú
Wide-SCSI: 16 v. 32 bites szinkron busz; 20Mbyte/s; 10Mhz; 15 egység; 68 pólusú csatlakozó
Ultra Wide SCSI: 16 v 32 bites; 40Mbyte/s; 20 Mhz; max 15 egység; 68 pólusú csat.
Ultra2 Wide SCSI: 16 v 32 bites; 80Mbyte/s; 40Mhz; max 15 egység; 68 pólusú; max kábelhossz. 12m
Ultra-160 SCSI: 16 v 32 bites; 160Mbyte/s; 80Mhz; max 15 egység; 68 pólusú; 12m (differenciális jellel működik)
Ultra-320 SCSI: 16 v 32 bites; 320Mbyte/s; max 15 egység; 68 pólusú; 12m (differenciális jellel működik)
Soros
Optikai

SCSI sínrendszer további jelei:
/DATA BUS: adatmozgás az egységek között
/DATA BUS PARITY: paritásbit ellenőrzés
/ACKNOWLEDGE és /REQUEST: aszinkron módú átvitel (handshaking) jelzésére
/CONTROL-DATA: vezérlőadatok, parancsok, állapotinformációk jelzése
/INPUT-OUTPUT: adatáramlás iránya
/SELECT: a kezdeményező kiválasztja a megfelelő egységet
/ATTENTION: vezérlő figyelmezteti a célt
/BUSY: sín foglalt állapota
/RESET: a sín alaphelyzetbe állítása

SCSI rendszer és a PC rendszerbusz kapcsolata:



SCSI vezérlő általános felépítése:



(**További SCSI részek**: fóliákon illetve SCSI.pdf)

16. RISC és CISC számítógép architektúrák:

1.) RISC processzorok jellemzői: (177.oldal) Pl: Motorola 88000 RISC rendszere vagy Berkeley Egyetem RISC I rendszere.
RISC: Reduced Instruction Set Computer (Csökkentett utasításkészletű számítógép):

- Csak a kívánt alkalmazásra jellemző utasítástípusokat tartalmaz, az utasításkészlet összetettségének csökkentése végett kihagytak olyan utasításokat, amelyeket a program amúgy sem használ, ezáltal nő a sebesség.
- *Minimális utasításkészletet és címzési módot* (csak amit gyakran használ), gyors HW elemeket, optimalizált SW használ.
- Azonban, hogy a programozási nyelvek komplex függvényei leírhatók legyenek (ahogyan az a CISC-nél működik) szubrutinokra, és hosszabb utasítássorozatokra (sok egyszerű utasítás) van szükség.
- Hogyan tudjuk a rendszer erőforrásait hatékonyan kihasználni? Gyorsabb működés érhető el (MIPS), egyszerűbb architektúra megvalósítására kell törekedni.
- *Azonos hosszúságú utasításformátum* (korlátozott utasításformátum miatt a tárolt programú gépeknél az *F-D-E* folyamatban a dekódolás minimális idejű lesz (nullának feltételezzük), amely során azonosítani kell a végrehajtandó utasítást)
- *Huzalozott (hardwired) utasításdekódolás* (a hardveres dekódolás megvalósításához kombinációs logikát használ, azonban a mai memóriaalapú mikro-kódú gépeknél ez lassabb).
- *Egyszeres ciklusvégrehajtás*: (minden egyes ciklusban egy utasítást hajt végre, ha ezt sikerülne elérni, akkor lenne optimális az erőforrás kihasználás - VLSI technológiafüggő. Egy lebegőpontos művelet rendkívül kis idő alatt végrehajtható. Hátránya, hogy vannak bizonyos műveletek, amelyeket egy ciklus alatt nem kapunk meg: pl. a memóriában lévő érték inkrementálásakor az értéket előbb ki kell venni, frissíteni, majd visszaírni a memóriába).
- *LOAD/STORE memóriaszervezés*: 2 művelet – tölt és tárol (regiszter <-> memória). Regiszterre azért van szükség, mivel a betöltött adatot sokkal gyorsabban tudjuk kiolvasni, mint a memóriából. Az aritmetikai/logikai utasítások a regiszterekben tárolódnak. A regiszterek gyorsabbak, mint a memória-intenzív műveletek.
- További architektúra technikák: pipeline (párhuzamos utasítás feldolgozás), többszörös adatvonalak, nagyszámú gyors regiszterek alkalmazásával.

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

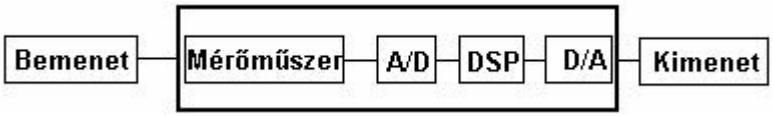
Regiszterablakozási technika: (RISC-I) Az eljáráshívás és az abból való visszatérés során a szubrutinok nagy processzoridőt használnak, mivel memóriában tárolnák az adatokat. A felhasznált processzoridő minimalizálására találták ki a regiszterablakos technikát.
A nagyszámú regiszterhalmazból egyidőben csak adott, korlátozott számú regisztert használ a rendszer. Ezt a korlátozott számú regisztert egy ablakkal azonosítják, és elkülönítik a többi regisztertől. Az ablak megváltozásakor egy pointer azonosítja, hogy az aktív regiszterek helyett új regiszterkészletet kell egy ablakba fogni. Amikor az ablak átlapolódik a szubrutinok, alprogramok között, akkor a paraméterek átadását *globális regisztereken* keresztül oldja meg, amelyek az összes szubrutin által elérhetőek. Az utasításkészletben 5 bittel azonosíthatjuk az utasításnál használni kívánt regisztert, ezáltal 32 különböző regiszter (R0-R31) címezhető meg (5 bit segítségével, mivel 2^5=32) Tehát vannak:
- *közös (globális) regiszterek*: R0-R9, minden szubrutin által elérhetők
- *rutin specifikus regiszterek*: R10-R31 mely további három részből áll (a regiszterek között történhet átlapolódás!)
  a.) Alacsonyszintű regiszterek: R10-R15
  b.) Lokális regiszterek: R16-R25
  c.) Magasszintű regiszterek: R26-R31

Ez az eljárás mindaddig jól működik, ameddig a paraméterek száma kisebb a regiszterek méreténél, mivel nem igényel memória-intenzív stack műveletet.

2.) CISC processzorok jellemzői: (177.oldal)
CISC: Complex Instruction Set Computer (Komplex utasításkészletű számítógép):

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

17. DSP processzorok:



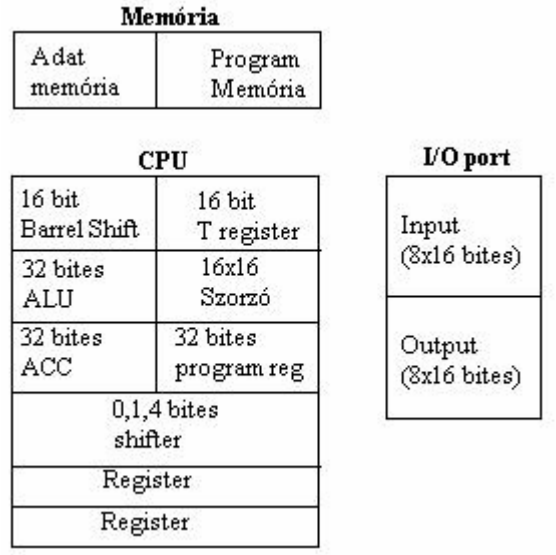
DSP tulajdonságai:
- olyan utasítások szerepelnek, amelyek a jelprocesszálást segítik
- nagy sebesség, bizonyos számításigényes feladatoknál: FFT, konvolúció, korreláció, szűrők stb.
- cél: algoritmusok real-time futtatása miatt sűrű mintavételezés szükséges a gyorsan változó folyamatok nyomon követésére
- megbízható, digitális architektúra
- olyan programozási, rendszerfejlesztési HW ill. SW kell, amelyekkel optimális (gyors) programot tudunk készíteni
- FDE: (Fetch – Decode – Execute) utasítások
- Pipelining technika: regisztereket használ az egyes aritmetikai (pl. összeadó, szorzó) áramkörök összekapcsolásánál, amely gyorsabb számítási sebességet eredményeznek, mivel nem kell megvárni az egyes ágakon érkező adatokat, hanem a korábbiakat a regiszterbe rakva további műveleteket végezhetünk. /4 szintű pipeline. Fetch-Decode-Read-Execute/

DSP előnyei:

- egyszerűen integrálható adott rendszerekbe (gyorsan piacra kerülhet, így egy chip előállítási költsége gyorsan megtérül)
- hatékony architektúra
- SW fejlesztői környezet, fejlesztői támogatás
- költség-hatékony
- a Texas TMS320cX család visszafelé kompatibilis a megírt program lefordítása/végrehajtása szempontjából

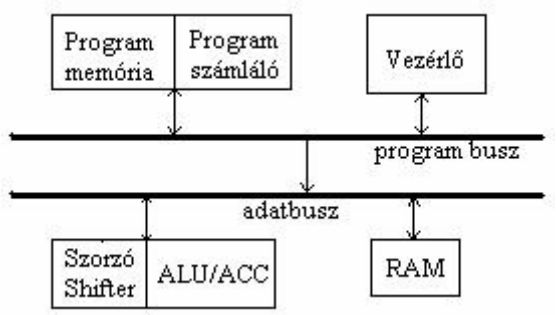
Számítási teljesítmény mérőszámai:
MFLOPS= Millió lebegőpontos utasítás/ másodperc
MIPS= Millió fixpontos (integer) utasítás / másodperc

DSP TMS320C10 processzor:

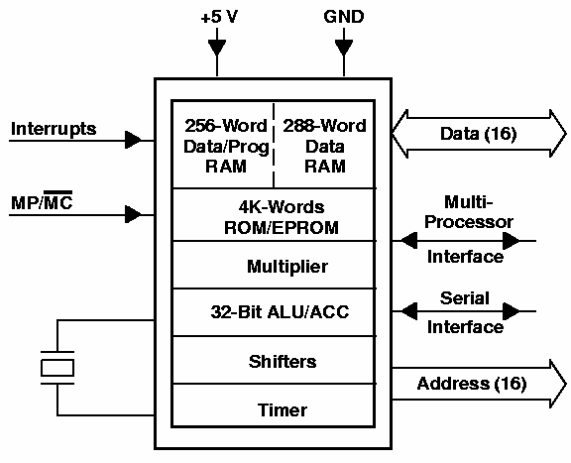


Ez a processzor a skalár-szorzat gyorsabb kiszámítására lett kifejlesztve. Egy utasítási ciklus alatt két számot összeszoroz és egy akkumulátorban lévő számhoz hozzáadja (x=a*b+c)
Egy utasítás végrehajtási ideje 160-280ns
Harward architektúrát használ: szét van választva a memóriában az adatterület és a programterület (nem Neumann elvű)

TMS320C1X (Harward architektúra): program memória + adat memória (RAM)



TMS320c25 DSP: jó tulajdonságai a robosztusság, zajérzéketlenség;



- Cél: a feldolgozandó CNN rács konvolválása a template-tel, mint konvolúciós ablakkal.
- 40MHz frekvencia; *fixpontos 16 bites* - 68 lábú jelfeldolgozó processzor. CMOS technológiával készül -> kis teljesítménydisszipáció (500mW); 80-100ns-os utasításvégrehajtási ciklusidő; 1 ciklus alatt szorzás/tárolás; 5V tápellátás;
- módosított Harward architektúra: adat és programmemóriája külön helyezkedik el, közöttük lehetséges az adatátvitel! Két adatmemóriája, amelyből az egyik variálható (programmemória lehet).: 4Kbyte-os On-Chip PROM és (256+288)x16 bites szóhosszúságú RAM; ill. 128Kbyte (2x64K) szóhosszúságú Off-Chip adattárhely; (nem Neumann).
- 32-Bites ALU egység: aritmetikai / logikai műveletek végrehajtása egy órajelciklus alatt. Elágazó utasítások kezelése. ALU egyik bemenete az akkumulátorból (ACC), míg a másik a szorzó vagy skálázható/állítható léptető Szorzat Regiszterből (PR) jön, miután a RAM-ból érkező adattal feltöltötték. Az eredmény az ACC-ben tárolódik. Az állítható 16 bites léptető bemenetét az adatbuszról kapja, 32 bites kimenete az ALU-hoz csatlakozik. Biztosítja a 16 bites bemenő adat megfelelő balra léptetését helyiérték helyesen 0-16 pozícióig.
- 16x16 Bites szorzóegység -> előjeles v. előjel nélküli 32bites szorzat eredmény kiszámítása egy gépi ciklus alatt. Két részre osztható: 16bites Temporary Regiszter (TR) mely ideiglenesen tárolja az egyik operandust, és 32 bites Szorzat Regiszter (PR).
- Blokkos adatátvitel; On-Chip időzítő-vezérlő műveletek; PGA-PLCC tokozások; soros port, D0-D15: 16 bites adatbuszok (LSB-MSB) A0-A15: 16 bites címbuszok (LSB-MSB)
- MP/MC: Microprocessor/Microcomputer: lehet egyprocesszoros rdsz.; párhuzamosan működő multiprocesszorok egyike; hierarchikus (slave/host) processzorok egyike globális memóriaterülettel; vagy perifériaprocesszor
- Rugalmas, nagysebességű (12.5 MIPS), processzortömbbe szervezhető chip. Párhuzamos architektúra, hatékony utasításkészlet (133 utasítással), HW-esen implementált funkciók; AR0-AR7: 8 külső/kiegészítő tároló.
- Magas szintű C nyelven programozható

TMS320c40 paralel DSP: *32 bites lebegőpontos* jelprocesszor;
Cél: nagysebességű belső/párhuzamos működés, tartós teljesítmény.
325 lábú kerámia tokozású (CGA), dupla metál CMOS technológia; 50ns utasítási ciklusidő;
CPU:
- -40/32 bites lebegőpontos/egész szorzó;
- -40/32 bites lebegőpontos/egész ALU műveletvégzés
- - 12db 40 bites lebegőpontos regiszter, 8 külső regiszter; 14 Control Regiszter; 5Vos tápellátás;
- - Két azonos külső Adat és Cím-busz: osztott memória hozzáférés; nagy 120Mbyte/s átviteli sebesség;
- - Különálló belső Adat-, DMA társprocesszor buszok
- -On-Chip Program Cache: 512Byte
- -Program/Adat RAM: 8Kbyte dual access/1ciklus
- -ROM alapú bootolás: 8-32Bites memóriával
- -1/x ill. √x kiszámítása egy ciklus alatt (Barrel Shifter: 1 ciklus alatt 32 lépés)

DMA társprocesszor: konkurens feldolgozás, CPU tehermentesítése
- -6 db DMA csatorna inicializálása a CPU beavatkozása nélkül
- -Párhuzamos DMA átvitel
- -memória-memória adatátvitel

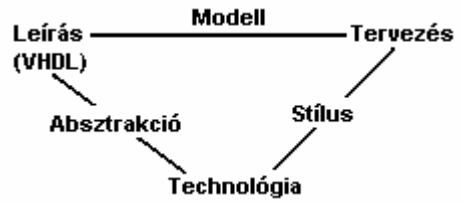
Kommunikációs portok:
- - külső HW vagy SW kommunikáció
- - 6db kommunikációs port a procik közötti közvetlen kommunikációhoz (DMA processzorok)
- - 20Mbyte/s kétirányú átvitel
- -FIFO tárak a procik közötti kommunikációhoz
- -arbitráció és handshaking támogatás

Lehetséges DSP alkalmazási területek:
- grafika: 3D, digitális látás, animáció, képfeldolgozás, alakzat felismerés
- hang/ beszéd: beszédfelismerés, hangfeldolgozás, szöveg-beszéd konverzió
- irányítás: robot-, motorirányítás
- katonai: titkos kommunikáció, radar feldolgozás, navigáció
- telekommunikáció: visszhangszűrés, modemek, adattitkosítás, videokonferencia
- autóipari: motorirányítás, rezgés analízis, csúszás- kipörgés-gátló,
- orvosi: hallókészülékek, diagnosztikai eszközök

18. A magas-szintű szintézis alapjai:

HLS: High Level Synthesis: Magas-Szintű Szintézissel történő egységbe foglalás
A HLS szinkron digitális rendszerek logikai szint feletti automatikus tervezését teszi lehetővé. Bemenete magas szintű hardver leíró nyelv (VHDL), amelynek segítségével a rendelkezésre álló adatbázisok alapján elektromos áramköröket építhetünk.
1.) In –core modell szintézis: a bemeneti magas-szintű nyelvből gráf-reprezentációt állít elő, amely a vezérlést és az adatokat külön-külön kezeli. (Lásd CFG, DFG).
2.) Adatfolyam analízis. Ez a módszer határozza meg a változók értékének élettartamát, amelyeket sokszor használunk a magas-szintű fordítónál, továbbá meghatározza rendeltetésszerű viselkedést (unholding változóknál). Allokáció és scheduling során is használjuk.
3.) A lehető leggyorsabb scheduling/ütemezés: a gyors végrehajtás eléréséhez minimalizálja a lehetséges útvonalak számát (azaz minimalizálja az utasításokban szereplő ciklusok számát). Optimalizálja az utak hosszát: a hossz a vezérlési lépések számát jelenti. Az FSM (véges állapotú gép) készítésekor a CFG-t körmentessé teszi a visszacsatolások elhagyásával.
Lehetnek feltételes elágazások. Az utak száma a csomópontok számának növelésével exponenciálisan nő.
4.) Modul hozzárendelés: A modul hozzárendelési probléma akkor lép fel, amikor a műveleteket többfajta funkcionális egységgel (FU: functional unit) kell megvalósítani, mivel ez a hozzárendelés választja ki az a megfelelő FU-t az adott utasítás elvégzéséhez.
5.) Belső (internal based) allokáció: a teljes kezdő adatút generálása és optimalizálása
6.) Késleltetés és méret becslése: logikai minimalizálás (vezérlőjelek számának csökkentése, ha lehetséges), ill. kimenetek generálása

HLS esetében mindig egy adott technológia szerint tervezünk, modellek segítségével.
A *leírás – tervezés – technológia* hármas a következő:



A HLS a tervezési-ütemezési (scheduling), a feladat felosztási- (allocation) és a kötött (binding)-algoritmusokra helyezi a fő hangsúlyt. Olyan eszközök lettek kifejlesztve, amelyek elfogadják, szimulálják és egyesítik/szintetizálják a terveket egy magasabb absztrakciós szinten: az emberi gondolkodásához közelebb állnak, segítségükkel komplexebb rendszerek építhetők fel egyszerűen. Egy magasabb szintű szintézisnél már nem elég csupán a tervezési- és feladat felosztási algoritmus, mivel az egyes hardware komponenseket (pl. DMA, buszvezérlők, interface részeket) szoftveres rendszerszintű specifikációval és leírással kívánjuk programozni, és ezeket a komponenseket egyesítjük.

A magas szintű szintézis elméleti alaptétele:
Turing Church tézis: (emlékeztető)
A szimbolikus számítások 3 fő ekvivalens reprezentánsai a következők:
- Turing gép (TM – hardver)
- μ-rekurzív függvény (algoritmusok, program)
- CFG- környezetfüggetlen nyelvtan (OS)

Egy nyelv, amely leírja a teljes tervezési folyamatot, az egyes tervezési lépések jóságát méri:
Nyelv <=> HW

A szintézis alapvető problémái/lépései: a szintézis magasabb szintű kiterjesztésével növekvő komplexitású problémát kellene megoldani (a leíró nyelvek, tervek, technológia magasabb absztrakciós szintjének bonyolultsági foka megnő).
- *Layout-terv szintű szintézisnél* a következő objektumról beszélünk: tranzisztorok, vezetékek ill. buszok (pl. metál rétegek), kontaktusok. Ennek a szintnek egy jó formális alapja a gráfelmélet, amely az objektumok elrendezésével és összekötésével foglalkozik. Ez a legalsó szint.
- *Logikai szintézis v. chip optimalizáció*: a Boole algebrai egyszerűsítéseken és a következő komponenseken alapul (pl: kapukon, flip-flopokon, kombinációs – szekvenciális hálózati elemeken stb.)
- *Chip szintézis*: a szintézist magasabb absztrakciós szintekre kiterjeszteni nehézkes az elméleti formalizmusok hiánya miatt (pl: Boole algebra, gráfelmélet), nem adható egyértelmű tervezési leírás.
- *Rendszerszintű szintézis*: SOC=System On a Chip technológia.

1. A tervezési lépések, koncepciók: probléma, hogy a tervezés ideje alatt a tervezési leírás akár többször is megváltozhat, nehézkes a specifikáció, különböző szemlélete lehet a tervezésbe bevont csapat tagjainak. Tehát nehéz egy egységes nyelvet adni általános célokra. Ehelyett különböző tervezési szemléletmódot (grafikus / szöveges) alakítunk ki az optimalizálást és ellenőrzést tekintve. Támogatnia kell továbbá a kézi, tervezői beavatkozást, ill. rendkívül fontos a nem-automatizálható lépések miatti többszörös ellenőrzések bevezetése!
2. Adatbázis: egy *központi adatbázis* tárolja el kezdetben a terv összes nézetét (aspektusát), amelyből aztán a kiválasztottak később megjeleníthetők, ill. a szintézis végrehajtása után bővíthetők. A formátum sémában tárolódnak az egyes nézetek típusai és formátumai. Továbbá rendelkezésre egy *gyorskereső adatbázis* is, amely megadja az egyes komplex összetevők paramétereit (terület, késleltetés, teljesítmény, dimenziók, alak). Ezek a paraméterértékek megváltoztathatók és új elemek generálhatók. Pl. ez adatbázis egy adott komponenshez tartozhat kapcsolási rajz, layout rajzolat, ill. generált VHDL leírás is.
3. Technológia függetlenség: ezt a legnehezebb biztosítani. Hogy ezt biztosítani tudjuk „layout szinten”, egy virtuális rácson dimenzió nélküli objektumokat kell megszerkeszteni, és a valós méretre kiterjeszteni (technológiai file-ban okoz csak változást). Logikai szinten a technológiai függetlenség általános kapuk használatával érhető el, amelyeket áttranszformálhatunk a könyvtári komponensek közé. Fel kell ismerni a hiányzó komponenseket és implementálni kell őket.
4. Tervezés tanulása: régebbi technológia átültetésével lehetséges (adaptation). Meg kell tanulni a layout-, logikai- és rendszer-szintű optimalizációt. Tehát *optimalizációs szabályok* és *ellenőrzési stratégiák* elsajátítása szükséges.
5. Szintézis komplexitásának tervezése: a szintézist egy magasabb absztrakciós szintre kiterjesztve több tervezési szint, több eszköz, és nyelv jelenik meg. Azonban lehetőség van rá, hogy a teljes szintézist két szintre tudjuk csökkenteni. Az egyik a *rendszerszintű szintézis:* regiszter-transzfer komponensekre fordítjuk le a kommunikációs folyamatokat, míg a másik a *komponens-szintézis*, amely az előbb kapott komponensek leírását layout szintre fordítja le Pl. viselkedésszintű leírások. Az egyes funkciókhoz lépések sorozatával struktúrát rendelünk: alacsonyabb szintű építőelemekkel és összeköttetéseikkel. Állapotváltozók táblázatának segítségével a múltbeli viselkedéseket az idő függvényében vizsgálhatjuk. Az egyes állapotok közti operációkat az ALU végzi, az eredményeket memóriában, regiszterekben tároljuk.

Alapvetően kétfajta leírás generálható:

Viselkedési (behavioral):	Adatfolyam	Vezérlésfolyam
Strukturális (architectural):	Adatút diagram (DFG)	Vezérlésfolyam diagram (CFG)

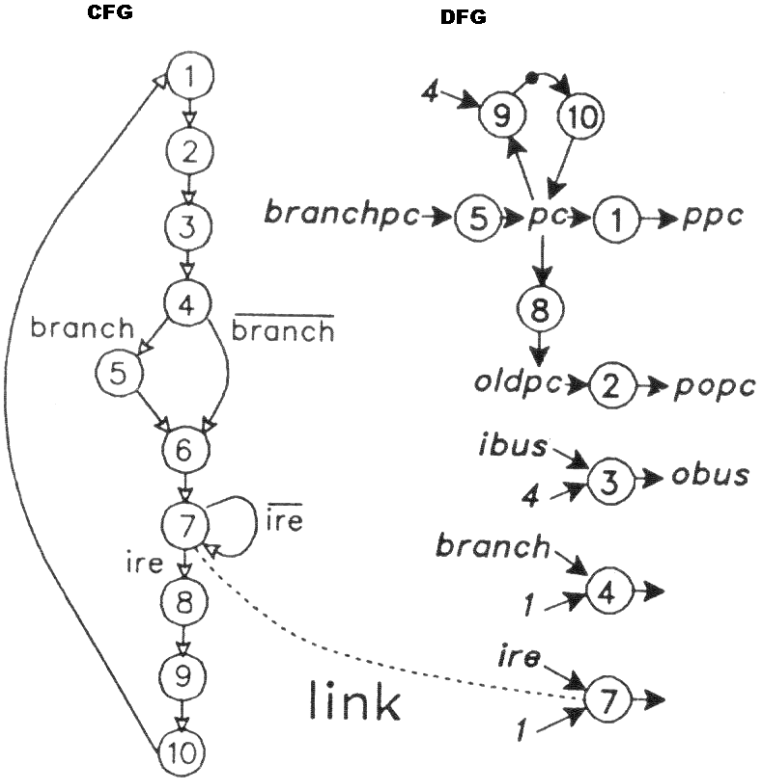
A (HLS) Magas Szintű Szintézisnél a tervezési specifikáció magas-szintű nyelven van definiálva, amelyben szereplő belső összetevők (pl. az egyes állapotok és a hozzájuk tartozó kapcsolatok) lefordíthatók (ábrázolhatók) **CFG (Control Flow Graph**: vezérlésfolyam gráf), vagy **DFG (DataFlow Graph**: adatfolyam gráf) segítségével. Az SSIM (Sequential Synthesis In-Core Model: Soros-Szintézisű Belső Modell) egy tervezési ábrázolás, amely a folyamat egy viselkedési leírása, ahol vezérlést és adatot teljesen függetlenül kezeljük. Az SSIM-et mindig a HLS előtt kell alkalmazni.

A **CFG** egy irányított gráf, és CFG = (N,P), ahol N a csúcsok halmaza (Pl: hozzárendelés, összeadás, logikai műveletek), míg a P az élek halmaza (Pl: precedencia relációk). A CFG segítségével ábrázolhatók az alapvető vezérlő szerkezetek soros leíró nyelven (viselkedési leírás).

- *Sorozat, szekvencia*. Egy él $(n1,n2) \in P$ jelenti, hogy n2 következik n1 végrehajtása után. Erre példa az ábrán látható (8,9) vagy (1,2) él.
- *Feltételes végrehajtás*. Egy művelet után pontosan egy újabb soron következő művelet hajtódhat végre, melynek kijelölése függ a feltételektől (mely élekhez kapcsolódik). A feltétel egy Boolean kifejezés, melynek értéke 1, ha a következő művelet végrehajtódik, egyébként 0. Az ábrán is látható, hogy a 4. és a 7. művelet elágazik különböző irányba (feltételes elágazás ill. ebben az esetben a ciklus is feltételtől függ). A feltételeket az élek mutatják.
- *Iteráció*. A CFG-k tartalmazhatnak ciklusokat, amelyek a folyamat iteratív viselkedését jelzik (hurkok). A 2.ábrán a 7. ill. 10. művelet jelöli az iterációt.

A **DFG** szintén irányított gráf, amelyet DFG=(N∪V,D) –vel definiálunk. Az N csúcsok az utasítások, míg V a változók halmazát jelöli. A D élek halmaza határozza meg az adatkapcsolatokat, (az ábrán az 1. utasításnál *pc* a bemenet, míg *ppc* a kimenet. Fontos, hogy a DFG-nél az egyes műveletek lehetnek ábrázolva úgy is, hogy nincs közöttük kapcsolódási pont (nem húzódik él), tehát részekre lehet bontani. A műveletek mind DFG, mind pedig CFG-ben azonos módon definiálhatók.

A következő ábrán látható a VHDL kódrészlet és a hozzá tartozó megfelelő CFG ill. DFG gráf:




```
Entity prefetch is
      Port (  branchpc, ibus in bit32;
              branch, ire: in bit;
              ppc, popc, obus: out bit32);
end prefetch;

architecture behavior of prefetch is
begin
      process
      variable pc, oldpc: bit32:=0;
        begin
              ppc <= pc;                  --1
              popc <= oldpc;              --2
              obus <= ibus + 4;           --3
              if (branch = '1') then      --4
                    pc:= branchpc;        --5
              end if;                     --6
            wait until (ire='1');         --7
            oldpc:=pc;                    --8
            pc:=pc+4;                     --9, 10
        end process;
end behavior;
```

A két gráf-reprezentáció között az egyes állapotokban lehetnek csatolások, *linkek* is.

A magas-szintű szintézis adja meg az SSIM tervezési struktúráját. Egy szintézissel előállított véges állapotú automata (FSM) az *állapotátmeneti gráffa*l jellemezhető, vagy egy szintetizált adatút az *adatút diagrammal*. Az SSIM hierarchikus felépítésű, azért, hogy bizonyos modulok (folyamatok, alprogramok) egyidőben és egymástól függetlenül kezelhetőek legyenek. A *modul hierarchiai gráf* a különböző modulok közötti kapcsolatokat tünteti fel. Mivel teljes SSIM-et a memóriában kell tárolni, amíg a magas-szintű szintézist el nem végezzük, biztosítani kell a tervezéshez szükséges memóriaterületet, hogy a sebesség megfelelő legyen.