Digitális Technika

Egyetemi Jegyzet

1. Kiadás

Dr. Sütő József

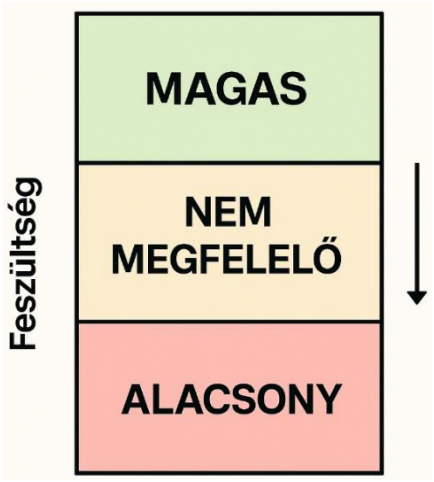
2025

Tartalomjegyzék

1. Bevezetés

Az elektronikus áramkörök két nagy kategóriába sorolhatók: **digitális** és **analóg**. A digitális elektronika diszkrét értékeket, míg az analóg elektronika folytonos értékeket használ. A diszkrét érték olyan értékeket jelent, amit le tudunk tárolni a számítógépes rendszerek memóriájában. A természetben, a legtöbb mérhető változó analóg. Gondoljunk például az eltelt időre vagy a hőmérsékletre. A hőmérséklet nem az egyik pillanatról a másikra változik meg 20-ról 21 fokra, hanem ez a változás folytonos. Ezeket az analóg mennyiségeket **mintavételezéssel** tudjuk **diszkretizálni**. Ez azt jelenti, hogy az analóg változót szabályos időközönként megmérjük és a mért értéket bináris kódként (erről később lesz szó) tároljuk le a számítógépes rendszerben.

Az egyik legfontosabb dolog, amit meg kell jegyezni, hogy a digitális elektronika olyan áramköröket foglal magába, amelyekben csak két lehetséges állapot létezik. Ezeket az állapotokat két különböző feszültségszint jelöl: a **magas (HIGH)** és az **alacsony (LOW)**. Ezt a két feszültség szintet digitalizáljuk. Általában a magas feszültség szintet egy 1-es értékkel reprezentálhatjuk, míg az alacsony feszültség szintet egy 0-ás értékekkel. Ennek szemléltetéséhet vegyük alapul az 1. Ábrát, ahol egy digitális rendszer feszültség szintjeit látjuk.



1. Ábra. Digitális rendszer feszültség szintjei.

Amennyiben a rendszer feszültség szintje a magas tartományba esik azt 1-es értéknek tekintjük, ha az alacsony tartományba, akkor azt 0-nak. Ha a feszültségszint a nem megengedett tartományban van, akkor arról nem tudjuk eldönteni, hogy az 0 vagy 1

értéket jelent, és ez a rendszer hibás működését fogja eredményezni. Példaként tegyük fel, hogy a rendszerünk a 0-3.3V feszültség tartományban működik. Ebben az esetben a magas feszültség szint 2V-3.3V között van, míg az alacsony feszültség szint 0V-0.8V között.

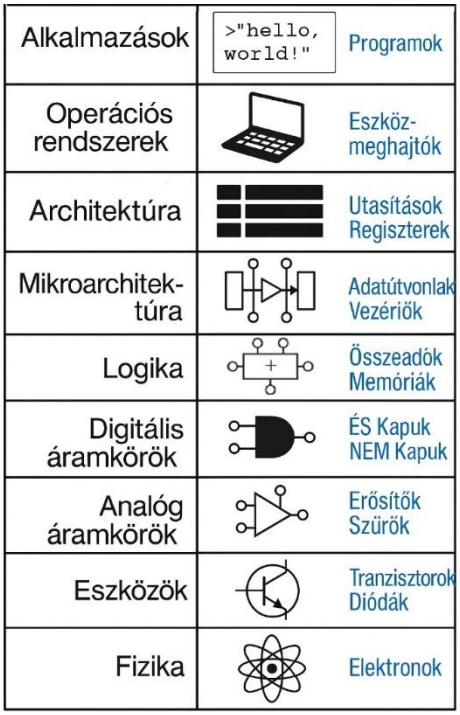
A bináris számrendszerben egy számjegy ezt a két értéket veheti fel. Ide vezethető vissza a bináris számrendszer fontossága a digitális rendszerekben, mivel az információ típusától függetlenül, mindent információt bináris számok segítségével tárolunk. Egy bináris számjegyet **bitnek** nevezünk, ami az információ tárolás alapegysége. A számok, szimbólumok, karakterek vagy más típusú információk ábrázolásához (tárolásához) több bitre van szükségünk és a rajtuk tárolt értéket **kódnak** nevezzük. Ebből az is következik, hogy a digitális technikában az **adat** kifejezés biteknek egy csoportját jelenti, amin hasznos információt tárolunk.

A későbbiekben látni fogod, hogy a digitális logikai kapuk is 0 és 1 értékekkel dolgoznak. Ez azt jelenti, hogy 0 és 1 értékeket kapnak a bemeneteiken és az általuk elvégzett logikai művelet alapján 0 vagy 1 értéket adnak kimenetként. Ez a viszonylag egyszerű működési elv nem igényelnek bonyolult matematikai számításokat vagy mélyreható fizikai ismereteket. Ehelyett a digitális rendszerek tervezőinek más kihívásokkal kell szembenéznie. Az egyik fontos kérdés az, hogy az egyszerű építőelemeket hogyan kapcsoljuk össze ahhoz, hogy a megtervezett rendszer működőképes legyen?

Miután megismerkedtünk a logikai kapukkal, azt fogjuk megvizsgálni, hogy hogyan lehet a logikai kapukat összekapcsolni összetettebb digitális áramkörökké, mint például összeadókká vagy tárolókká (memóriává). Ennek a jegyzetnek a szerkezeti felépítése strukturáltan fog haladni a számrendszerektől kiindulva egészen az olyan komplex digitális áramkörökig, mint például egy hétszegmense kijelző vagy jelzőlámpa vezérlő, amik a gyakorlatban is fontos szerepet töltenek be. Ezen az úton haladva megpróbálok rávilágítani arra, hogy a digitális rendszerek hasonlóan épülnek fel, mint a „LEGO játékok". Itt is vannak elemi „építőköveink" (tranzisztorok), amiket összekapcsolva összetettebb „építőelemeket" (logikai kapuk) hozhatunk létre. Ezeket is össze tudjuk kapcsolni annak érdekében, hogy összetettebb áramköröket alkossunk belőlük (**kombinációs** és **szekvenciális** áramkörök). Végül olyan bonyolult rendszer létrehozására is képesek lehetünk, mint például egy mikroprocesszor. Annak érdekében, hogy a bonyolult rendszereket létre tudjuk hozni egyszerűbb komponensekből, mindig szem előtt kell

tartani a **modularitást**, ami azt jelenti, hogy a rendszer minden komponensének jól definiált funkcióval és interfésszel kell rendelkeznie.

Ahhoz, hogy teljes képet kapjunk a digitális rendszerek működéséről, sőt azt is mondhatjuk, hogy a teljes számítógépes rendszer működéséről, a **hierarchikus** gondolkodási mód elengedhetetlen. Ez segítséget nyújt számunkra a bonyolult rendszerek, mint például egy számítógépes rendszer kezelésében. Erre azért van szükség, mert a modern digitális rendszerek akár több milliárd tranzisztort tartalmazhatnak. Egyetlen ember sem lenne képes megérteni ezeknek a működését, ha a vizsgálatot teljesan az alapoktól kezdné el, mint mondjuk az elektronok mozgása minden egyes tranzisztorban [1]. A strukturált gondolkodásban segítségünkre lesz az **absztrakció**. Leegyszerűsítve, ez azt jelenti, hogy elfedjük aztokat a részleteket, amelyek kevésbé fontosok a számunkra. Példaként vegyük a 2. Ábrát, ahol a számítógépe rendszerek absztrakciós szintjei látható. Ennek a jegyzetnek a keretein belül mi főként a logika kapuk és a logikai áramkörök szintjeire fogunk fókuszálni. Viszont ahhoz, hogy átfogó képet kapj a digitális rendszerek felépítéséről, néha át kell nyúlni más szintekre is, de ezekben az esetekben is segíteni fog az absztrakció.



2. Ábra. A számítógépes rendszerek absztrakciós szintjei.

2. Információtárolás digitális rendszerekben

Ahogy korábban már említésre került, a legtöbb fizikai változó, mint például a feszültség vagy az idő folytonos. Ezzel szemben a digitális rendszerek diszkrét változókkal reprezentálják az információt, amelyek véges számú értéket tudnak felvenni. Ha már információnál tartunk, célszerű bevezetni a **bit** fogalmát (bináris számjegy), ami az információtárolás alapegysége a digitális rendszerekben. Egyetlen biten két különböző értéket tudunk letárolni: 0-át vagy 1-et. Első pillantásra, ez a két érték önmagában nem tűnik túl hasznosnak, de ahogy korábban említettem a digitális rendszerek nagy előnye, hogy az építőelemeik 0-ás és 1-es értékekkel dolgoznak.

Tehát 1 biten vagy egy 0-át vagy egy 1-es értéket tudunk letárolni. Ezzel még nem megyünk túl sokra. Szerencsére a biteket össze lehet kapcsolni, ami lehetőséget nyújt több különböző érték tárolására. Egy diszkrét változónak, ami N különböző értéket vehet fel, a lehetséges információ mennyisége (D) bitben kifejezve a következő módon határozható meg:

$$D = log_2 N \tag{1}$$

Valójában a 0 és 1 értékek felfoghatóak igaz (1-es) és hamis (0-ás) értékeknek vagy úgy is szoktak hivatkozni rájuk, mint magas feszültség szint (1) és alacsony feszültség szint, mivel a digitális rendszerekben tipikusan a logikai 1-es magas míg a 0 alacsony feszültség szintekkel vannak reprezentálva. Ez egy fontos kapocs az analóg és a digitális világ között, hiszen a digitális rendszerekben a folytonos feszültség értékeket diszkrét értékké alakítjuk és azzal dolgozunk.

3. Számrendszerek

A mindennapi életünk során már megszoktuk, hogy decimális számokkal dolgozunk. Decimális számokkal vannak megadva a termékek árai, a várakozási idő a piros lámpánál, vagy a sorszám az orvosi rendelőben. Ezen felül azt is érdemes megjegyezni, hogy valójában végtelen sok számrendszer létezik, de szerencsére ezek közül nekünk csak egy néhány lesz fontos a digitális technikában. Annak ellenére, hogy a decimális számrendszert már mindenki jól ismeri, mi mégis ezzel fogjuk kezdeni a számrendszerek áttekintését annak érdekében, hogy rávilágítsak a számrendszerek általános működési elvére.

3.1 Tízes számrendszer

A tízes (decimális) számrendszer már mindenki számára jól ismert, mert ezt használjuk minden nap. Mivel a legtöbb embernek 10 ujja van, ez lehetőséget biztosít arra, hogy a tízes számrendszer egy számjegyének minden értékét meg tudjuk jeleníteni velük 0-tól 9-ig. Egyetlen decimális számjegyen összesen ezt a 10 különböző számot tudjuk reprezentálni. A számjegyeket össze is kapcsolhatjuk, hogy ezzel nagyobb értékű decimális számokat is képesek legyünk ábrázolni. A több számjegyből álló számoknál, minden egyes számjegy pozíciónak megvan a saját **helyiértéke**. A helyiérték nagysága a számjegyeken jobbról balra féle haladva növekszik, mégpedig a tízszeresére. Egy egész decimális számban az első számjegy helyiértéke 1, a következőé 100, az azt követőé 1000 és így tovább. Egy fontos kérdés az, hogy hogyan lehet meghatározni az egyes számjegyek helyiértékeit. Ehhez tudnunk kell a **számrendszer alapját**, ami a decimális számrendszerben 10. Ha megvan az alap, akkor ezt kell a megfelelő hatványkitevőre emelni. Egy egész decimális számban, a legkisebb helyiértékű számjegynél a hatványkitevő értéke 0-tól indul és ahogy haladunk a számjegyeken balra felé, ez az érték egyesével növekszik. Ennek szemléltetésre tekintsük a 3.1 Példát, ahol az is látható, hogy egy decimális számot hogyan bonthatunk szét a számjegyek helyiértékeinek felhasználásával. Az ilyen jellegű átírás a későbbiekben a segítségünkre lesz a számrendszerek közötti váltáskor.

3.1 Példa: Írd át a 8672-őt a számjegyek helyiértékeinek felhasználásával.

$$8672_{10} = 8*10^3 + 6*10^2 + 7*10^1 + 2*10^0$$

A szám egész része mellett arra is van lehetőségünk, hogy a szám tört részét is hasonló módon írjuk le. Ekkor a decimális pont jobb oldalán álló számjegyek helyiértékeinek negatív előjele lesz és a kitevő értéke jobbra haladva eggyel csökken. Nézzünk erre is egy példát.

3.2 Példa: Írd át a 23.476-ot a számjegyek helyiértékeinek felhasználásával.

$$23.476_{10} = 2*10^1 + 3*10^0 + 4*10^{-1} + 7*10^{-2} + 6*10^{-3}$$

A gyakorlatban sokszor azt is el kell döntenünk, hogy hány számjegyre van szükség ahhoz, hogy egy diszkrét változó minden lehetséges értékét képesek legyünk ábrázolni. Ezt könnyen megtehetjük, ha tisztában vagyunk a számjegyek által biztosított **tartománnyal**. Általánosan azt mondhatjuk, hogy N darab decimális számjegy felhasználásával 10^N különböző értéket tudunk leírni, aminek a tartománya: 0, 1, …, 10^N - 1. Például három decimális számjeggyel összesen 1000 különböző értéket tudunk leírni, aminek a tartománya 0-tól 999-ig tart.

3.2 Kettes számrendszer

A decimális számrendszerben egy számjegy 10 különböző értéket tudott felvenni (0-tól 9-ig). A kettes (bináris) számrendszerben egy számjegy mindössze két számot tud ábrázolni: 0 és 1. A korábban leírtak alapján, látni kell azt, hogy egy kettes számrendszerbeli szám számjegyét le lehet letárolni egy bit felhasználásával. Ebből fakadóan, a kettes számrendszer számjegyeire a **bit** kifejezést is használjuk. A kettes számrendszerben is hasonló módon határozhatjuk meg a számjegyek által nyújtott számtartományt, mint a decimális számrendszerben. Ehhez vesszük a számrendszer alapját, ami a kettő és azt akkora kitevőre emeljük, ahány számjegyünk van. Tehát, egy N bites számmal összesen 2^N különböző számot tudunk ábrázolni, amelynek a tartománya: 0, 1, …, 2^N -1.

A következő, amit fontos lesz megfigyelni, az a kettes és tízes számrendszerek működési elve közötti hasonlóság. A kettes számrendszerben is összekapcsolhatjuk a számjegyeket, itt is van alapja a számrendszernek, ami a kettő és itt is minden egyes számjegynek megvan a saját helyiértéke. Sőt, a helyiértékek kiszámításának az elve is megegyezik azzal, amit a decimális számrendszernél láttál. Ehhez itt is vennünk kell a számrendszer alapját, ami a kettő és azt kell a megfelelő hatványkitevőre emelni. A hatványkitevő értéke, egy

egész számnál itt is 0-tól kezdődik és jobbról balra féle haladva egyesével növekszik. Tehát ebben az esetben a helyiértékek nagyságai: 1, 2, 4, 8, 16, 32, 64, 128, 256, stb. Annak érdekében, hogy a későbbiekben gyorsan tudj váltani a számrendszerek között fejben is, érdemes megjegyezni 2 néhány pozitív és negatív hatványát. Ehhez az 1. Táblázat nyújt segítséget.

Pozitív hatványok (Egész számok)								**Negatív hatványok (Törtszámok)**					
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
256	128	64	32	16	8	4	2	1	1/2 0.5	1/4 0.25	1/8 0.125	1/16 0.0625	1/32 0.03125

1. Táblázat. Kettő néhány fontosabb hatványa.

A szemléltetés kedvéért nézzük meg, hogy hogyan lehet átváltani egy bináris számot decimális számmá.

3.3 Példa: Határozd meg az 11001 bináris szám decimális alakját.

$$11001 = 1*2^0+0*2^1+0*2^2+1*2^3+1*2^4 = 1+8+16 = 25$$

Figyeld meg az előző példában, hogy az átváltás során, elég lenne csak azokra bitpozíciókra fókuszálni, ahol az érték 1 és a hozzá tartozó helyiértékeket kell összeadni. Ne feledd el, hogy ha nullával szorzol egy számot az nulla lesz, így kiesik az összegézből. A kettes számrendszerben is lehet a szám tört részét is ábrázolni, hasonló módon, mint ahogyan azt a tízes számrendszernél láttuk. A tört részhez tartozó számjegyek helyiértékeinek a hatványkitevője itt negatív előjelű lesz. Nézzük ezt meg egy példán keresztül:

3.4 Példa: Határozd meg az 101.110 bináris szám decimális alakját.

$$101.110 = 1*2^{-2}+1*2^{-1}+1*2^0+1*2^2 = 0.25+0.5+1+4 = 5.75$$

Ha több számrendszerrel is dolgozunk, akkor célszerű lehet alsó indexben megadni azt a számrendszert, amiben a számot ábrázoljuk (3.2 Példa). Ezzel elkerülhető a számrendszerek felcseréléséből fakadó későbbi problémák. Mivel a legtöbb példákban a számrendszer egyértelmű, ezért csak indokolt esetben fogom jelölni.

Fordítsuk meg az előző példát és nézzük meg egy decimális szám átváltását a kettes számrendszerbe. Kezdetben erre a feladatra a maradékos osztás módszerét használhatjuk (3.5 Példa), de a későbbiek során az egyszerűbb átváltásokat akár fejben is elvégezheted. Ahhoz, hogy átváltsunk egy decimális számot binárissá a maradékos osztás módszerével, addig kell osztani a kiindulási számot és a kapott részeredményeket 2-vel, amíg el nem érjük a 0-át. Az egyes lépések (osztások) során leírjuk a maradékot és a maradék értékek adják a szám bináris alakját.

3.5 Példa: Határozd meg a 153 kettes számrendszerbeli alakját.

Osztás 2-vel	Hányados	Maradék
153 : 2 = 76	76	1
76 : 2 = 38	38	0
38 : 2 = 19	19	0
19 : 2 = 9	9	1
9 : 2 = 4	4	1
4 : 2 = 2	2	0
2 : 2 = 1	1	0
1 : 2 = 0	0	1

A bináris alakot a maradékok adják lentről felfelé haladva leírva: 10011001

Arra is van lehetőség, hogy a tízes számrendszerbeli tört értéket váltsuk át a bináris számrendszerbe. A decimális egész szám bináris átváltásához a kiindulási számot 2-vel osztjuk folyamatosan. Ezzel szemben, a tört rész átváltásához a kiindulási tört részt kell majd folyamatosan 2-vel szorozni, ameddig a részeredmény tört része 0 nem lesz vagy amíg el nem értük a kívánt pontosságot (bitszámban). A számolás során, a részeredmények egész értékeit külön lejegyezzük, mert ez fogja adni a szám bináris alakját. Nézzük meg ezt egy példán keresztül.

3.6 Példa: Határozd meg a 0.625 decimális szám kettes számrendszerbeli alakját.

Szorzás 2-vel	Eredmény	Egészrész
0,625 × 2 = 1,25 →	1,25	→ 1
0,25 × 2 = 0,5		→ 0
0,5 × 2 = 0,5 →	0.5	→ 0
0,5 × 2 = 1,0		→ 1
0,0 × 2 = 0,0 →	0.0	→ 0

A bináris alakot itt is a maradékok adják, de fentről lefelé haladva: 0.1001

3.3 Tizenhatos számrendszer

Az előzőek alapján, szerintem már nem lesz nehéz kitalálni a tizenhatos (hexadecimális) számrendszer működési elvét. Igen, itt a számrendszer alapja 16 lesz és egy számjegy 16 különböző értéket vehet fel. De honnan jön ez a 16 érték? Mivel a jól megszokott decimális számrendszerben csak 10 különböző számjegyünk van, ezért azt találták ki, hogy az angol abc első hat karakterét még hozzáveszik a 10 számjegyhez és így állt elő a hexadecimális számrendszer egy számjegyének 16 lehetséges értéke: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A B, C, D, E, F. Fontos azt látni, hogy ha a karaktereket átváltanánk decimális számrendszerbe, akkor az A = 10, B = 11 és így tovább. A számjegyeknek itt is megvan a saját helyiértéke, amit pont úgy határozunk meg, mint a kettes és tízes számrendszereknél. Szemléltetésként vess egy pillantást a 2. Táblázatra, ahol a hexadecimális számrendszer számjegyei és azok decimális és bináris alakjai látható.

Hex	Decimális	Bináris alak
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

2. Táblázat. A hexadecimális számrendszer számjegyei.

Gondolom azt mindenki látja, hogy miért fontos a kettes és tízes számrendszerek. De mi a helyzet a hexadecimális számrendszerrel? Ezt a számrendszert azért használják elterjedten, mert a kettes számrendszerben nem kényelmes a hosszú, sok számjegyből álló számokkal dolgozni. Mivel a kettes és a tizenhatos számrendszer között nagyon gyorsan tudunk átváltani, így a tizenhatos számrendszer egy olyan alternatíva a kettes helyett, ahol a nagy számok is kompakt formában ábrázolhatóak. Ez sok esetben megkönnyíti és gyorsítja a munkánkat. A gyors váltás a két számrendszer között annak köszönhető, hogy minden egyes hexadecimális számjegy ábrázolható 4 bit felhasználásával ($2^4 = 16$). Így, egy hexadecimális szám kettes számrendszerbe történő átváltása során csak a számjegyeket kell átváltani kettes számrendszerbe, 4 bit felhasználásával. Nézzünk erre is egy példát.

3.7 Példa: Határozd meg az 10111010 bináris szám hexadecimális alakját.

$$10111010 \rightarrow 1011\_1010 \rightarrow BA$$

A hexadecimális szám visszaváltása a bináris számrendszerbe is hasonlóan egyszerű. Annyit kell tenni, hogy egyesével vesszük a hexadecimális szám számjegyeit és azokat átváltjuk bináris számmá 4-bit felhasználásával.

3.8 Példa: Határoz meg az AFC3 hexadecimális szám bináris alakját.

$$AFC3 \rightarrow 1010\ 1111\ 1100\ 0011 \rightarrow 1010111111000011$$

Vegyünk egy másik példát, amiben a hexadecimális számot decimálissá alakítjuk. Ebben a példában azt is láthatod, hogy a programozási nyelvekben és egyéb alkalmazásokban a hexadecimális számok **0x**-el kezdődnek, ami mindössze annyit jelent, hogy ez a szám tizenhatos számrendszerben van.

3.9 Példa: Határozd meg az 0xA8F szám decimális alakját.

$$0xA8F = A*16^2+8*16^1+F*16^0 = 10*256+8*16+15*1=2703$$

3.4 Bináris számokhoz kapcsolódó fogalmak

Korábban említettem, hogy az információtárolás alapegysége a bit. Egy másik, gyakran használt mértékegység a **bájt**, ami 8 bitet jelent. Már tudod azt, hogy ha 8 bited van, akkor azzal $2^8 = 256$ különböző számot jelent, tehát egy bájton 256 különböző értéket tudsz tárolni. Egy másik gyakori mértékegység a **fél bájt**, amiről nem nehéz kitalálni, hogy 4

bitet foglal magába. 4 biten 16 különböző értéket tudsz tárolni, ami egyben lefed egy hexadecimális számjegyet. A fenti fogalmakon túl be kell még vezetnünk a **legkisebb helyiértékű (LSB)** és **legnagyobb helyiértékű (MSB)** bitpozíciók fogalmát (3. Ábra). Ahogy a neveik is jelzik, egy többszámjegyből álló bináris számban mindig a legjobboldalibb számjegy (legkisebb helyiérték) adja az LSB bitpozíciót, míg a legbaloldalibb (legnagyobb helyiérték) az MSB bitpozíció. Az LSB és MSB kifejezések nem csak bitekre értendő, hanem akár bájtokra is, ha olyan számokkal dolgozol, amelyek több bájtból állnak.

3. Ábra. A számítógépes rendszerek absztrakciós szintjei.

3.5 Bináris számokhoz összeadása

A bináris számokat hasonló módon adjuk össze, mint a decimális számokat egy darab papíron. Sőt, még annál egyszerűbb is, mivel itt egy számjegynek csak két értéke lehet. Ha az összeadás során az egyik oszlopban kapott részeredmény nem fér bele egyetlen számjegybe, akkor azt mondjuk, hogy átvitel érték keletkezett, amit majd az eggyel magasabb helyiértékű pozíciónál fogunk felhasználni. Annak érdekében, hogy tiszta képet kapj a bináris összeadásról, vess egy pillantást a 10. és 11. Példákra.

3.10 Példa: Számold ki az 4277 és az 5499 decimális számok összegét.

$$\begin{array}{cccccc} & & \color{red}{1} & \color{red}{1} & & \\ & 4 & 2 & 7 & 7 & \\ + & 5 & 4 & 9 & 9 & \\ \hline & 9 & 7 & 7 & 6 & \end{array}$$

3.11 Példa: Számold ki az 1001 és a 0101 bináris számok összegét.

			1	
	1	**0**	**0**	**1**
+	**0**	**1**	**0**	**1**
	1	**1**	**1**	**0**

Figyeld meg, hogy a 10. Példa első oszlopában elhelyezkedő számok összeadásakor a részeredmény 9+7=16. A 16-ból a 6-ot írtuk le az eredménybe, míg a fennmaradt tíz átvitel érték formájában lesz felhasználva a következő oszlopban. Ugyan ez történt a bináris esetben is. Az első oszlop számjegyeinek összeadásakor a részeredmény $1_2+1_2=10_2$ (itt kiraktam a számrendszer jelölő alsóindexeket, mert ennek hiányában az eredmény furán festene). Az 10_2 részeredményből (a decimális értéke 2) itt a 0 került be az eredménybe, míg az 1-ből átvitel érték lett, amit a következő oszlopban használunk fel.

A digitális rendszerekben többnyire fix mennyiségű bit áll a rendelkezésünkre a változók értékeinek tárolására. Amennyiben az összeadás során a kapott eredmény több bitből áll, mint a rendelkezésre álló bitek száma (egész számok esetén), akkor azt mondjuk, hogy **túlcsordulás** történt. Nézzünk erre is egy példát:

3.12 Példa: Számold ki az 1101+0101 összegét feltéve, hogy 4-biten kell tárolni az eredményt.

	1	**0**	**1**	**1**
+	**0**	**1**	**1**	**0**
1	**0**	**0**	**0**	**1**

A fenti példában, mivel 4-bit áll a rendelkezésünkre, így az eredmény MSB bitjét nem lehet letárolni. Ebből adódóan a tárolt érték a 0010, ami természetesen hibás. A túlcsordulás problémát úgy lehetne megoldani, ha megnövelnénk a bitmennyiséget, amin az eredményt tároljuk. 5 biten már helyesen lehet tárolni a kapott eredményt. Általánosan az mondható el, hogy ha két N-bites számot adunk össze, akkor az eredmény hibamentes tárolásához legalább **N+1-bitre van szükség!**

3.6 Előjeles számok ábrázolása

Eddig csak előjel nélküli számokkal foglalkoztunk. Magától érthető módon, valahogy azt is meg kell oldani, hogy előjeles számokat is képesek legyünk kezelni a digitális rendszerekben. Erre a feladatra több számábrázolási módszer is született, amelyek közül mi a két legelterjedtebbet fogjuk áttekinteni.

Kezdjük az egyszerűbbel, ami az **előjel-nagyság** számábrázolás. Ebben a számábrázolási módszerben egy N-bites bináris számnak az MSB bitje jelöli az előjelet, míg a fennmarad N-1-bit adja a szám nagyságát, ami a szám abszolút értéke. Ezt formulával a következő módon tudjuk leírni, ahol a_i az i. bitpozíció értéke (0 vagy 1):

$$A = (-1)^{a_{n-1}} \sum_{i=0}^{n-2} a_i 2^i \tag{2}$$

Ha az MSB bit értéke 1, akkor a szám negatív, ellenkező esetben pozitív. Nézzük egy példát az előjel-nagyság számábrázolásra:

3.13 Példa: Határozd meg a 6 és a -6 decimális értékek 4-bites előjel-nagyság ábrázolását.

- +6 = 0110
- **-**6 = 1110

Sajnos az előjel-nagyság számábrázolás túl egyszerű ahhoz, hogy általánosan használható legyen. Ennek a számábrázolásnak két problémája is van. A kisebbik, hogy a 0-t kétféleképpen ábrázolja. Például, 4-bite számoknál a 0000 (0) és az 1000 (-0) is 0-át jelent. A másik nagy probléma az, hogy az aritmetikai műveletek helytelen eredményt adnak az előjel-nagyság számábrázolás esetén. Ezt a probléma jól szemlélteti a következő példa.

3.14 Példa: Határozd meg a 6 és (-6) 4-bites előjel-nagyság ábrázolású számok összegét.

	1	**1**	**1**	**0**
+	**0**	**1**	**1**	**0**
1	**0**	**1**	**0**	**0**

A fenti példában kapott eredmény nem 0, tehát az eredmény helytelen. Ez akkor is fennáll, ha a vezető 1-est nem tároljuk le, mert 4-biten dolgozunk. Mivel 4 bit áll a rendelkezésünkre, így a kopott eredményből csak az alsó 4-bitet lehet letárolni, ami 0100. A másik előjeles számábrázolási módszer, amit a számítógépes rendszerek is használnak az a **kettes komplemens** számábrázolás. A kettes komplemens számok szinte azonosak az előjel nélküli számokkal annyi különbséggel, hogy itt az MSB bitpozíció helyiértékének az előjele negatív: -2^{N-1}, míg az előjel nélküli bináris ábrázolásnál ez pozitív: 2^{N-1}. Ezzel a minimális változtatással megoldódnak az előjel-nagyság számábrázolásnál látott problémák. Itt is fel lehet írni egy képletet, amely a segítségedre lesz egy kettes komplemens szám decimális számrendszerbe történő átváltásakor:

$$A = a_{n-1}(-2^{n-1}) + \sum_{i=0}^{n-2} a_i 2^i \tag{3}$$

A számábrázolás sajátossága miatt, az mondhatjuk, hogy ha az MSB bit értéke 1, akkor az a szám negatív, míg a pozitív számok esetében az MSB mindig 0. Ennek alapján az MSB bit egyben az előjelet is jelöli, de nem olyan módon, mint azt az előjel-nagyság számábrázolásnál láttad. Nézzünk erre egy példát.

3.15 Példa: Határozd meg a legnagyobb és a legkisebb 4-bites kettes komplemens számokat.

- 0111 = 7
- 1000 = -8

Azt fontos látni, hogy a pozitív számoknál a kettes komplemens ábrázolás megegyezik az előjel nélküli binárissal. Tehát csak a negatív előjelű számok jelentenek eltérést az egyszerű bináris ábrázolástól. Sok esetben lehet szükség egy bináris szám negatív előjelű ábrázolásának a meghatározására. Ez könnyen megoldható az **előjelfordítás** kétlépéses algoritmusával:

1. Invertáld a kiindulási szám bitjeit (0-ból 1 és 1-ből 0)
2. Adj egyet az előző lépésben kapott számhoz

Ennek az egyszerű algoritmusnak az első lépését úgy is nevezik, hogy a szám **egyes komplemense**. A név onnan származik, hogy az eredeti szám plusz annak a komplemense mindig csupa 1-eseket tartalmazó számot ad eredményül. Vegyünk

szemléltetésként egy 4-bites számot x = 0101, aminek a komplemense $\bar{x}$ = 1010. Összegük: x + $\bar{x}$ = 0101 + 1010 = 1111. Ha ezt az eredmény átváltjuk tízes számrendszerbe, akkor -1-et kapunk. Itt jön a képbe az algoritmus második lépése mivel, ha átrendezzük az előző x + $\bar{x}$ = -1 összefüggést, akkor azt kapjuk, hogy $\bar{x}$ + 1 = -x. Tehát, ha az egyes komplemenshez hozzáadunk 1-et, akkor megkapjuk a kiindulási érték negatív előjelű ábrázolását.

3.16 Példa: Határozd meg a -3 (3 - 0011) 4-bites kettes komplemens alakját.

	1	**1**	**0**	**0**
+				1
(-3)	**1**	**1**	**0**	**1**

3.17 Példa: Határozd meg az 1011 kettes komplemens szám decimális értékét.

$$1011 = -2^3 + 2^1 + 2^0 = -8 + 2 + 1 = -5$$

Hasonlóan a bináris számok ábrázolásához, egy N-bites kettes komplemens szám 2^N különböző értéket tud ábrázolni. Viszont, ez a tartomány pozitív és negatív számokra oszlik fel. Általánosan egy N bites kettes komplemens szám tartománya: $[-2^{N-1}, 2^N-1]$. A kettes komplemens számábrázolás nagy előnye, hogy az aritmetikai műveletek elvégzése során helyes eredményt kapunk. Szemléltetésként vegyük az előző fejezetben is használt példát:

3.18 Példa: Határozd meg a 6 és (-6) 4-bites kettes komplemens ábrázolású számok összegét.

	1	0	1	0
+	**0**	**1**	**1**	**0**
1	**0**	0	**0**	**0**

Figyeld meg, hogy az előző példa eredményében kaptunk egy átvitel bitet, ami túllóg a rendelkezésünkre álló 4-bites tartományon. Ha ilyen átvitel bit keletkezik az összeadás során, azt soha nem tároljuk le és így a kapott eredmény helyes lesz. Fontos azt is

kihangsúlyozni, hogy itt is előfordulhat túlcsordulás. Ez akkor következik be, ha azonos előjelű számokat adunk össze és az eredmény előjele ezzel ellentétes.

A másik fontos aritmetikai művelt, a **kivonás**, összeadással helyettesíthető. Vegyünk példaként a következőt műveletet: 6 – 3 = 6 + (-3). Ez jól szemlélteti azt, hogy a kivonást le lehet cserélni összeadásra, de ekkor a második operandus előjelét meg kell invertálni. Ezt viszont könnyen meg tudjuk tenni, a már korábban látott előjel cserélő módszerrel. Összegezve, kettes komplemens számok kivonásakor az első operandushoz hozzáadjuk a második operandust invertált előjellel.

Az egyszerűség kedvéért, a fenti példákban 4-bites számokkal dolgoztunk. Természetesen arra is lehetőségünk van, hogy ezt a bittartomány kibővítsük. Egy N-bites számot bármikor kibővíthetünk M bitesre, ahol M > N, ha az eredeti szám MSB bitjét másoljuk az eredeti szám bal oldalán megjelenő új bitpozíciókra. Ezt a kiterjesztési módszert **előjel kiterjesztésnek** nevezzük. Fontos megjegyezni, hogy a kiterjesztett szám értéke nem változik meg a kiterjesztést követően.

3.19 Példa: Terjeszd ki az 1011 4-bites kettes komplemens számot 8-bitesre.

$$1011 = \mathbf{1111}1011$$

A digitális világban, a bináris számokkal előjel nélküli, kettes komplemens vagy előjel-nagyság formában is találkozhatunk. A 3. Táblázat ennek a három számábrázolási módnak a tartományait mutatja be.

Számrendszer	Tartomány
Előjel nélküli	$[0, 2^N-1]$
Előjel/nagyság	$[-2^{N-1}+1, 2^{N-1}-1]$
Kettes komp.	$[-2^{N-1}, 2^{N-1}-1]$

3. Táblázat. Előjel nélküli, előjel-nagyság és a kettes komplemens számábrázolások tartományai.

4.Fixpontos és lebegőpontos számok

4.1 Fixpontos számok

A fix pontos számábrázolásnál egy feltételezett pont helyezkedik el az egész és a tört rész között. Hasonlóan a decimális ponthoz, ez is egy képzeletbeli határ szerepét tölti be a két rész között. Korábban már láttad, hogy a bináris számrendszerben lehetőségünk van arra is, hogy egy szám tört részét letároljuk. Ebből fakadóan a fix pontos számábrázolásnál, tudnunk kell, hogy hány bitet használunk az egész és hány bitet a tört rész tárolásához. Vegyünk egy példát:

4.1 Példa: Mennyi a decimális értéke a 01111010 számnak, ha tudjuk, hogy 4-4 bitet használunk az egész és a tört rész tárolására?

$$0110110 \rightarrow 0111.1010 = 7.625$$

Ha előjeles, tört résszel rendelkező számokat kell ábrázolni, akkor az előjel kezelésére használhatjuk a korábban már bemutatott előjel-nagyság és a kettes komplemens számábrázolást is.

4.2 Példa: Mi lesz a fix pontos ábrázolása (4-4 biten) a -3.375-nek, ha az előjelet előjel-nagyság számábrázolással kezeljük?

$$-3.375 = 1011.0110$$

4.3 Példa: Mi lesz a fix pontos ábrázolása (4-4 biten) a -3.375-nek, ha az előjelet kettes komplemens számábrázolással kezeljük?

$$-3.375 = 1100.1010$$

Az előző példában a 3.375 fixpontos számábrázolásából indultunk ki és arra alkalmaztuk az előjelfordítás lépéseit, amit a 3.6.2-es fejezetben mutattam be.

Fontos azt látni, hogy a fixpontos számok is csak bitek sorozata és a digitális rendszerekben sehol sem tároljuk a bináris pontot. Ebből adódóan, csak akkor tudunk dolgozni fixpontos számokkal, ha előtte leszögezzük az előjel és a tört rész tárolására felhasznált bitek számát. A szakirodalomban, az **U/Q*a.b*** kifejezést használják ennek megadására, ahol a kezdő karakter az előjelre utal (U - előjel nélküli, Q – előjeles), *a* az

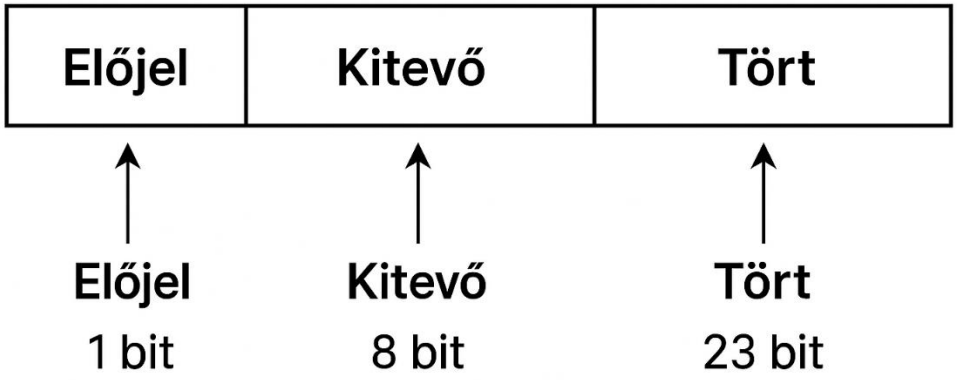
egész bitek száma, míg *b* a tört bitek száma. Annak ellenére, hogy a fixpontos ábrázolásnak megvan a hátránya, elterjedten használják a digitális jelfeldolgozásnál és a gépi-tanulásnál, mivel a fixpontos számokat használó rendszerek számítási sebessége gyorsabb és kevesebb energiát igényel, a lebegőpontos számábrázolást használó rendszerekhez képest.

4.2 Lebegőpontos számok

Ahhoz, hogy nagy, egész számokat tároljunk meglehetősen sok bitre van szükség és ez akkor is igaz, ha nagy pontossággal szeretnénk a tört részt tárolni.
A lebegőpontos számábrázolás a számok **tudományos alakjára** épít, amely a következő:

$$\pm M \times B^E \tag{4}$$

A fenti kifejezés azt mutatja, hogy a tudományos alak három részből áll: **mantissza (M)**, **alap (B)** és **exponens (E)**. Bármelyik számot át lehet írni tudományos alakba. Vegyük a 6124-et, aminek a decimális tudományos alakja 6.124 * 10^3. Figyeld meg, hogy az átalakítás során a decimális pontot a legnagyobb helyiértékű számjegy jobb oldalára mozgattuk át. Ahhoz, hogy a szám eredeti nagyságát megtartsuk, a pont átmozgatását követően, a kapott értéket 10 hatványával szoroztuk be. A hatvány kitevő értéke megegyezik a számjegyek számával, amennyivel a pontot balra mozgattuk. Azt is mondhatjuk, hogy a kitevő értéke azt mutatja, hogy hány számjeggyel kell a pontot mozgatni (innen jön a lebegő pont elnevezés). Ez az átalakítási módszer kiterjeszthető bármelyik számrendszerre. A bináris számrendszerben az IEEE 754-1985 standardot használják, amelynek három változata van: egyszeres precizitású (32 bit), kétszere precizitású (64 bit) és négyszeres precizitású (128 bit) tárolás. Ezek között a fő eltérés az, hogy hány bitet használnak fel a mantissza és az exponens tárolására. A programozási nyelvekben, ha létrehozol egy **float** vagy **double** típusú változót, akkor azok is az IEEE 754-1985 standard alapján lesznek tárolva a számítógép memóriájában. A hasonló tárolási elv miatt, mi csak az egyszeres precizitási tárolással fogunk foglalkozni, ahol 32 bitet használnak fel a lebegőpontos szám tárolására. A 32 bitből az MSB bit lesz az előjel bit (S), a következő 8 bit alkotja az exponens mezőt, míg a fennmaradt 23 bit lesz a mantissza mező (4. Ábra).



4. Ábra. Az IEEE 754-1985 standard egyszeres precizitású számábrázolás mezői.

Az exponens mező tartalma egy eltolt érték, amit úgy kapunk meg, hogy a tudományos alakból származó exponens értékhez hozzáadunk 127-et. Itt az eltolást a konstans 127 jelenti. Ennek az az előnye, hogy így az ábrázolható számok tartományát lényegesen ki lehet terjeszteni anélkül, hogy az exponens előjelét külön tárolnánk. Azt is fontos tudni, hogy a bináris tudományos alakban, a bináris pont bal oldalán (egész rész) szinte mindig 1-es érték szerepel egy pár kivételtől eltekintve. Ezt kihasználva ezt az 1-es értéket soha nem tárolni és ezzel megtakarítunk egy bitet! Annak érdekében, hogy teljes képet kapj a lebegőpontos szám tárolási folyamatáról, nézzük meg, hogy egy decimális számot hogyan kell átalakítani lebegőpontos formába.

4.4 Példa: Mi lesz a 228 decimális szám lebegőpontos alakja?

1. lépés: konvertáld át a számot bináris számrendszerbe: 11100100
2. lépés: alakítsd át tudományos alakba: 1.1100100×2^7
3. lépés: állítsd be a mezők értékeit: S=1 (pozitív), M=11001000000000000000000, E=7+127=134=10000110

Az IEEE lebegőpontos szabvány speciális esetként kezeli az olyan számokat, mint a nulla, a végtelen és az érvénytelen érték. Ezeknek a speciális értékeknek az ábrázolása látható a x. Táblázatban.

Szám	Előjel	Exponens	Mantissza
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000

NaN	X	11111111	Nem null

4. Táblázat. IEEE 754 speciális értékek

A lebegőpontos számábrázolás természetesen nem tesz lehetővé tetszőleges pontosságú számábrázolást. Ebből fakadóan, kerekítésre is szükség van, ami az LSB bit tartalmát érinti. Továbbá, túlcsordulás is bekövetkezhet. Ebben az esetben a +/- ∞ speciális értéke kerül beállításra.

Eddig csak azt néztük meg, hogy hogyan tudunk egy számot lebegőpontossá alakítani, de a lebegőpontos alakból vissza is kell tudni alakítani a számot. Ebben fog a segítségünkre lenni a következő formula:

$$(-1)^S(1+F)(2^{E-127}) \tag{5}$$

A formula három részre osztható (zárójeles tagok), ahol az első, a szám előjelét állítja be. Itt az S az előjel bit tartalmát jelenti, ami 0 vagy 1 lehet. A második tag állítja vissza a mantissza mező tartalmából (F) a tudományos alak mantissza értékét (M). Végül a harmadik tag állítja vissza a tudományos alak szorzó részét kompenzálva az átalakításkor használt 127-es eltolás értékkel. Figyeld meg, hogy a szám visszaalakításakor az (5) képletben visszarakjuk a tudományos alak nem tárolt vezető egyesét és kivonjuk az exponens mezőben elhelyezett eltolás értéket (127), amit a lebegőpontossá konvertálás során alkalmaztunk. Nézzük meg, hogy ez a gyakorlatban hogyan működik.

4.5 Példa: Határozzuk meg a 01001000110001110001000000000000 lebegőpontos szám decimális értékét.

$$1.10001110001 \text{ x } 2^{18} = 1100011100010000000 = 407680$$

Korábban említettem, hogy a fixpontos számábrázolás használatával az aritmetikai műveletek gyorsabban elvégezhetők, mint lebegőpontos számábrázolás használatakor. Ez abból ered, hogy a lebegőpontos számok esetében az összeadás folyamata sokkal összetettebb annál, mint amit a kettes komplemensnél láttál. Az összeadás folyamata lépésekbe szedve a következő:

1. Az exponens és a mantissza bitek kinyerése.
2. A vezető egyes visszaállítása a mantisszába
3. Hasonlítsuk össze az exponenseket és egységesítsük

4. Ha az egyik exponens értékét módosítottuk, akkor a hozzá tartozó mantisszát is módosítani kell, ahhoz, hogy a szám nagysága ne változzon meg
5. Add össze a mantisszákat
6. Az eredményt hozzuk tudományos alakra
7. Ha kell kerekítsünk
8. Állítsuk össze az eredmény mezőinek tartalmát

4.6 Példa: Add össze a 00111111110000000000000000000000 és a 01000000010100000000000000000000 lebegőpontos számokat?

1. lépés: E1= 01111111, M1=.1; E2=10000000, M2=.101
2. lépés: M1=1.1; M2=1.101
3. lépés: E1= 127, E2=128 → E1= 128
4. lépés: M1 = 0.11 (szoroztunk 2^{-1}-el)
5. lépés: 0.11 + 1.101 = 10.011
6. lépés: 10.011 = 1.0011 x 2^{1}
7. lépés: nem kell kerekíteni
8. lépés: E=128+1=129 (közös exponens plusz a tudományos alak exponense), M=00110000000000000000000

5.Bináris kódok

5.1 Binárisan kódolt decimális (BCD)

A BCD kódok célja az, hogy egy egyedi bináris kódot társítson a tízes számrendszer minden egyes számjegyéhez, ezzel felgyorsítva a számjegyek bináris átváltását. Mivel 10 különböző számjegyhez (0, 1, 2, …, 9) kell egyedi kódot rendelni, így minimum 4 bitre van szükség. Ebből adódóan a BCD kódok is 4 bitesek, és gyakorlatilag azt mondhatjuk, hogy a decimális számjegyhez társított BCD kód nem más, mint a számjegy bináris értéke 4-biten ábrázolva. A teljes lista a következő:

- 0: 0000
- 1: 0001
- 2: 0010
- 3: 0011
- 4: 0100
- 5: 0101
- 6: 0110
- 7: 0111
- 8: 1000
- 9: 1001

Figyeld meg, hogy 1001 után már nincs szükség a fennmaradt kódokra (1010, 1011, 1100, 1101, 1110, 1111), ezért ezeket a kódokat **érvénytelen BCD kódoknak** is szokták nevezni. Nézzük meg, hogy hogyan lehet átváltani egy decimális számot BCD kódokra.

5.1 Példa: Váltsd át a 2469-et BCD kódra.

$$2469 \rightarrow 0010\ 0100\ 0110\ 1001$$

A fenti példában egyesével végig mentünk a decimális szám számjegyein és leírtuk a számjegyek BCD kódjait. A példából az is látható, hogy a BCD kódok használatával a decimális szám átalakítása bináris formába sokkal gyorsabb, mint a kiindulási szám bináris értékének a megtalálása. A fenti átalakítást meg is tudjuk fordítani. Nézzünk erre is egy példát.

5.2 Példa: Határozd meg a következő BCD kód decimális alakját: 1001010001110000.

1001010001110000 → 9470

A 5.2-es példában a BCD kódot jobbról indulva 4-bites részekre bontottuk szét. Ezek a 4-bites részek adják a decimális szám számjegyeit. Az utolsó résznél előfordulhat, hogy már nincs 4-bit, ekkor a hiányzó bitpozíciók értékeit 0-nak feltételezzük.
A gyakorlatban számos eszközben használják a BCD kódokat decimális számjegyek megjelenítéséhez. Később ezzel még foglalkozunk.

5.2 Gray kód

A Gray kód nem aritmetikai kód, ami annyit jelent, hogy a kód bitpozícióinak nincsen súlya (helyiértéke). Ezeknek a kódoknak a legfontosabb jellem az, hogy a szomszédos kódszavak csak egyetlen bitpozíción térnek el egymástól. A gyakorlatban a Gray kódokat adattárolók vagy enkóderek (pozíció meghatározás) szektorainak címkézésére használják. Példaként tekintsd meg a 5. Táblázatot, ami 3-bites Gray kódokat tartalmaz és a hozzájuk tartozó bináris számokat.

Bináris	Gray kód
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

5. Táblázat. Bináris számok és azok Gray kódjai.

5.3 Paritás bit

Ha veszünk egy n bitből álló bitsorozatot, akkor abban az 1-es értékek száma páros vagy páratlan. Ha egyetlen bithiba történik a bitsorozat átküldése során A pontból B pontba,

vagy egy adott rendszerből a másikba, akkor az 1-esek száma megváltozik. Itt hiba alatt, egy bit értékének a megváltozását értjük 0-ról 1-re vagy 1-ről 0-ra egy adatcsatornán, az átküldés során. Ha kibővítjük az eredeti bitsorozatot egy további bittel, akkor fixálni tudjuk az egyesek számának paritását (páros, páratlan). Tehát, a paritásbitet egy bitsorozathoz rendeljük hozzá n+1. bitként úgy, hogy ezáltal az 1-esek száma páros vagy pont páratlan legyen az n+1 bites bináris vektorban. Az 1-esek száma alapján kétféle paritási rendszer létezik:

- **Páros paritás:** az 1-esek szám páros az n+1 bitben
- **Páratlan paritás:** az 1-esek szám páratlan az n+1 bitben

A paritásbitet, a korai digitális rendszerekben használták **hibaellenőrző kódként**. A paritás bit, az eredeti bitsorozat jobb és bal oldalára is helyezhető, de a legtöbb digitális rendszerben ezt az eredeti bitsorozat bal oldalára (az MSB bit után) helyezik.
A paritásbit lehetővé teszi egy (vagy páratlan számú) bithiba előfordulásának detektálását. Ha kettő vagy páros számú bithiba következik be, akkor a paritás bittel ezt nem tudjuk detektálni. Nézzük meg, hogy hogyan lehet paritásbitet rendelni egy bitsorozathoz.

5.3 Példa: Adj páros paritásbitet a következő bitsorozathoz: 1000111001001.

1000111001001 → **0**1000111001001

A fenti példában megszámoltuk az 1-esek számát a kiindulási bitsorozatban, ami 6. A 6 páros szám, tehát már páros számú 1-es van a bitsorozatban, így a paritásbitet 0-ra kell állítani. A paritásbitet a kiindulási bitsorozat MSB bitje elé helyezzük el. Most nézzük meg azt, hogy a paritásbit alapján hogyan lehet meghatározni azt, hogy bithiba történt az átküldés során.

5.4 Példa: Egy páratlan paritású rendszer a következő bitsorozatot fogadta: 1100010101010. A paritásbit alapján döntsd el, hogy keletkezett e hiba az átküldés során?

> **Válasz:** igen, mert a beérkező bitsorozat 6 egyest tartalmaz, ami páros. Viszont páratlan paritású rendszert használunk, ahol az 1-esek számának páratlannak kell lenni helyes működés esetén.

Ennek az alfejezetnek a lezárásként meg kell említeni, hogy a mai digitális rendszerekben a bithibák bekövetkezésének valószínűsége alacsony. Ennek következtében a paritás bit használata egyre kevesebb helyen fordul elő.

6.Logikai kapuk

A logikai kapuk egyszerű digitális áramkörök, amelyek logikai műveleteket valósítanak meg. A kapuk egy vagy több bináris bemeneti értéket kapnak és ennek alapján egy bináris kimeneti értéket adnak. Mindegyik logikai kapu saját szimbólummal rendelkezik, ami alapján a kapu típusa egyértelműen azonosítható. A bemenetek általában a kapu bal oldalán jelennek meg, míg a kimenetet a jobb oldalon. A bemenetek és a kimenet közötti kapcsolatot kétféleképpen is meg lehet adni. Az egyik lehetőség az **igazságtábla** használata, amiben a kapu összes lehetséges bemeneti kombinációját felsoroljuk és megadjuk hozzájuk a kapu kimeneti értékét. A másik lehetőség a **Boole algebrai egyenlet** használata.

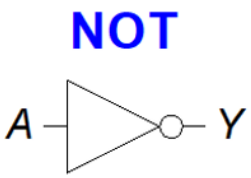
Az 1850-es években az ír matematikus, George Boole kifejlesztett egy a hagyományos algebrához hasonló matematikai rendszert, a logikai egyenletek leírására. Ennek segítségével egy áramkör logikai működését is le tudjuk írni, ahol az áramkör ki és bemeneteit bináris változókkal reprezentáljuk.

Áramköri szempontból a Boole algebrában használt műveleteket logikai kapukkal valósíthatók meg. A kapuk csoportosíthatók a bemenetek száma szerint, ahol a bemenetek száma megegyezik a logikai változók számával, amivel a kapu képes az adott logikai műveletet végrehajtani. Ennek alapján lesznek egy bemenetű és több bemenetű kapuk.

6.1 Egybemenetű kapuk

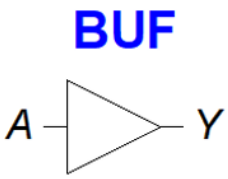
Az egybemenetű kapuk csoportjába mindösszesen két kapu tartozik, az **inverz** (**NOT) kapu** és a (**BUF) buffer**. A buffer-nek analóg elektronikai szempontból van jelentősége, de logikai szempontból ez helyettesíthető egy vezetékkel, mivel a bemenetén kapott bináris értéket továbbítja a kimenetére. Így ebben a jegyzetben ezzel a kapuval nem foglalkozunk. Az inverz kapu a bemenetére érkező logikai értéket invertálja és ezt adja kimenetként. Tehát, ha a bemenet 0, akkor a kimenete 1, ha a bemenete 1, akkor a kimenet 0. Az egyenletekben az invertálás műveletre a felülvonás jellel fogok hivatkozni: $Y = \overline{A}$, de a szakirodalomban más jelölések is előfordulnak, amik az inverzműveletet jelölik. Az egybemenetű kapuk szimbóluma, logika egyenlete és igazság táblája az 5. Ábrán látható. Jegyezd meg, hogy minden logikai kapunak saját szimbóluma van, ezért ránézésre meg

kell tudnod határozni a logikai kapu típusát a szimbóluma alapján. Függetlenül a logikai kapu típusától, a kapu bal oldalán helyezkednek el a bemenetek és a jobb oldalon a kapu kimenete. Továbbá, figyeld meg az 5. Ábrán, hogy a logikai kapu működését meg lehet adni a kapu szimbólumával, az egyenletekben használt logikai operátorral és a kapu igazság táblájával.



$$Y = \overline{A}$$

A	Y
0	1
1	0



$$Y = A$$

A	Y
0	0
1	1

5. Ábra. Egybemenetű kapuk

6.2 Több bemenetű kapuk

A több bemenetű kapuk csoportjába tartozik az **AND (ÉS)**, **OR (VAGY)**, és **XOR (kizáró VAGY)** kapuk, valamint ezek invertált változata. Az AND kapu sajátossága, hogy csak akkor ad a kimenetén logikai 1-es értéket, ha minden bemenete 1-es értékű. Minden más esetben a kimenete 0. Az egyenletekben, két változó (pl.: A, B) közötti AND műveletre, műveleti jel nélkül fogok hivatkozni (AB).

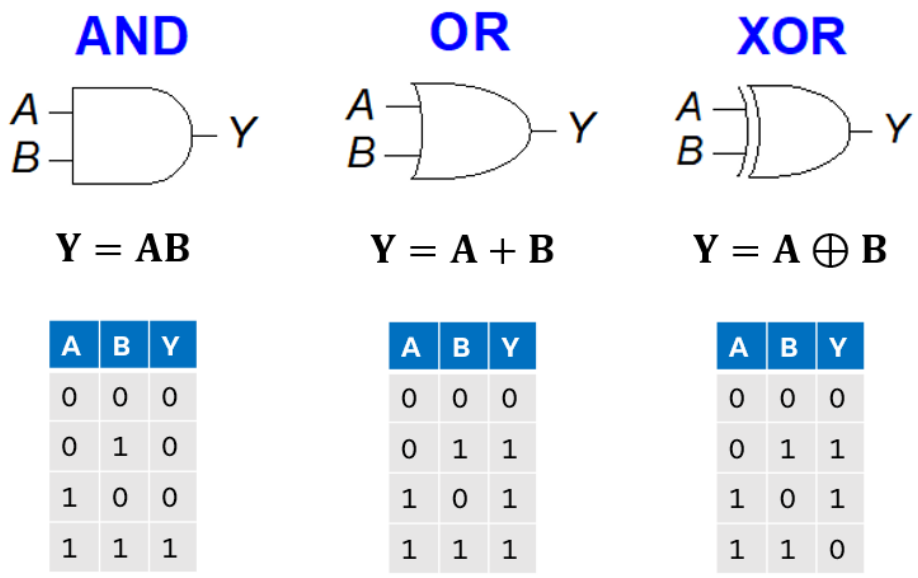
Az AND kapuval szemben az OR kapu szinte mindig logikai egyes értéket ad a kimenetén, csak egyetlen esetben nem, ha minden bemenete 0-s értékű. Az egyenletekben a logikai vagy műveletet a + operátor fogja jelölni.

Programozás során az OR és AND műveleteket **maszkolásra** is szoktuk használni. Ekkor az OR művelettel egy bitsorozat bizonyos bitjeit 1-re állíthatunk, míg a többit változatlanul hagyjuk. Tegyük fel, hogy 8-bites számokkal dolgozunk és egy 8-bites szám alsó 2-két bitjét 1-re szeretném állítani, míg a többit változatlanul akarjuk hagyni. Ebben az esetben, a 00000011 maszkot kell létrehozni és bitenkénti OR műveletet kell végrehajtani közötte és az eredeti szám között. Ahol a maszkban 1-es érték áll, ott az eredmény bitértéke is 1

lesz. Az AND művelettel, az eredeti bitsorozatban lévő 1-es értékek törölhetők egyszerűen az ehhez létrehozott maszk segítségével.

Az előző két kaputól eltérően, a kétbemenetű XOR kapu azokban az esetekben ad 1-es kimeneti értéket, ha a két bemenet értéke nem egyezik meg egymással. A több, mint két bemenettel rendelkező XOR kapunál, a kimenet akkor lesz 1-gyel egyenlő, ha a bemenetén páratlan számú 1-es értéket kapott. Az egyenletekben az XOR műveletre egy körben elhelyezett plusz jellel fogok hivatkozni.

Az AND, OR és XOR kapuk szimbóluma, igazság táblája és logikai egyenlete a 6. Ábrán látható. Figyeld meg, hogy az igazságtábla sorai minden lehetséges bemeneti kombinációt tartalmaznak, ami nem más, mint a bemeneti változókkal felírható számok tartománya. Két 1-bites bemenet esetén ez a tartomány 2^2 elemű (3.2 fejezet)! Az igazságtáblában, a lehetséges bemeneti kombinációkat mindig úgy adjuk meg, hogy a tartomány legkisebb értékével kezdünk (csupa 0-a), majd binárisan egyesével növekedve lépkedünk felfelé a tartomány legnagyobb értékéig (csupa 1-es).



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

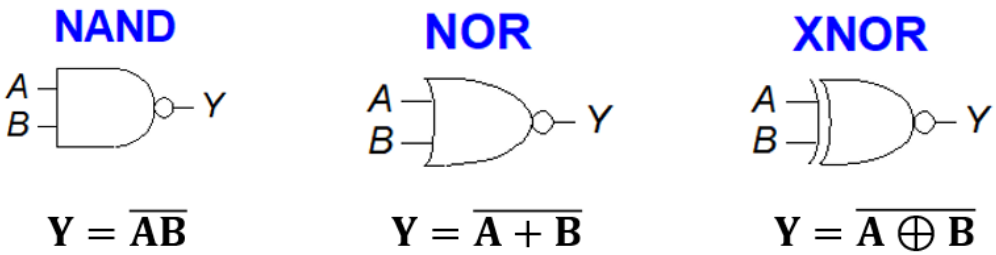
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

6. Ábra. Az AND, OR és XOR kapuk.

Ha kiegészítjük az AND, OR és XOR kapukat egy inverz művelettel a kapu kimeneténél, akkor további három elemi kapu hozható létre: NAND, NOR, és XNOR. Ennek a három kapunak a szimbóluma mindösszesen annyival tér el az AND, OR és XOR kapuk szimbólumától, hogy a kapu kimeneténél megjelenik egy „buborék”. Ez a „buborék” ekvivalens egy inverz kapuval, de digitális áramkörök megalkotásának szempontjából sokkal praktikusabb. Ez az inverz művelet megjelenik a kapuk működésében is. Vegyük

például a NAND kaput, ami csak abban az esetben ad 0-s kimenetet, ha minden bemenete 1-es értékű. Ezzel szemben az AND kapu csak ebben az egy esetben adott 1-es kimenetet, míg minden más bemeneti kombinációra 0-s kimenettel válaszolt. Ehhez hasonlóan, a kétbemenetű XOR kapu akkor adott 1-es kimeneti értéket, ha a két bemenet eltérő volt. Ezzel szemben az XNOR akkor ad 1-es kimenetet, ha a két bemeneti érték azonos. Ennek a tulajdonságnak köszönhetően az XNOR kaput **egyenlőség** vizsgáló kapunak is szokták nevezni. A NAND, NOR és XNOR kapuk szimbóluma, igazság táblája és logikai egyenlete a 7. Ábrán tekinthető meg.



$$Y = \overline{AB}$$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

$$Y = \overline{A + B}$$

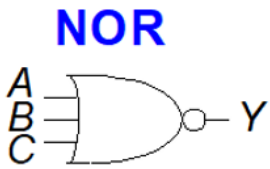
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

$$Y = \overline{A \oplus B}$$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

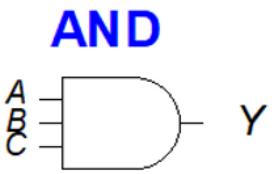
7. Ábra. A NAND, NOR és XNOR kapuk.

Az AND, OR, XOR, NAND, NOR és XNOR kapuk több, mint két bemenettel is rendelkezhetnek. Viszont, függetlenül a bemenetek számától, a kapuk logikai működése ugyan az marad. Ennek szemléltetésére tekintsd meg a 8. Ábrát, ahol három bemenetű NOR és AND kapukat látsz. Figyeld meg, hogy a három bemeneten összesen $2^3 = 8$ bemeneti kombinációt lehet felírni, így a kapuk igazság táblái is 8 sort tartalmaznak (plusz fejléc). Viszont ettől függetlenül, a kapuk működési elve nem változott. A NOR kapu csak akkor ad 1-es kimenetet, ha minden bemenete 0-s értékű volt, míg az AND kapu csak akkor ad 1-es kimenetet, ha minden bemenete 1-es értékű volt.



$$Y = \overline{A + B + C}$$

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



$$Y = ABC$$

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

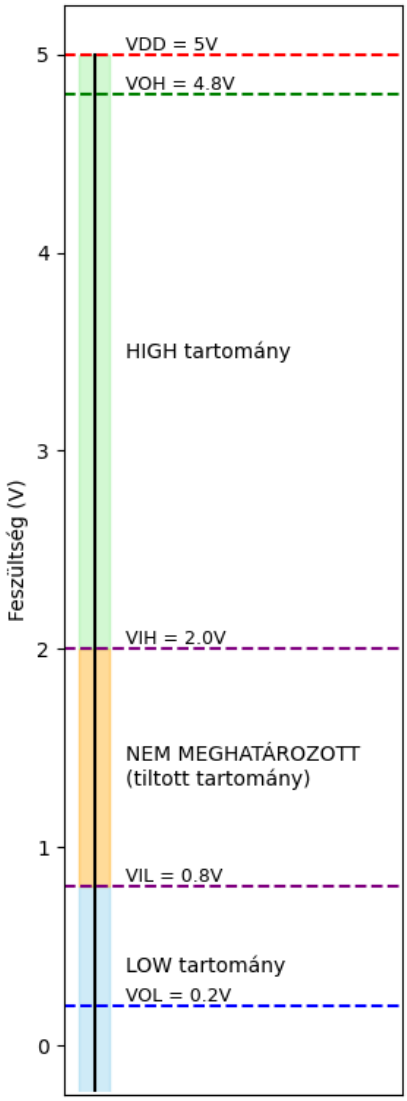
8. Ábra. Három bemenetű NOR és AND kapuk.

6.3 Logikai szintek

A digitális rendszerekben meg kell oldani a feszültség érték (folytonos) digitalizálását, hiszen a bináris változók gyakorlatilag feszültség értéket reprezentálnak. Ezt leegyszerűsítve, úgy képzelheted el, hogy egy olyan rendszerben, ahol a feszültségtartomány 0V-5V (Volt) közötti, ott a logikai 0 az 0V-ot jelent, míg a logikai 1 az 5V. Ez egy idealizált eset, mert a valós rendszerekben számolnunk kell a **zajjal** is. Mindent zajnak tekintünk, ami az eredeti feszültség értéket valamilyen formában módosítja.

A digitális áramköröket úgy kell megtervezni, hogy azok képesek legyenek ellenállni egy adott fokú zajmennyiségnek. A gyakorlatban ezt egyrészt úgy oldják meg, hogy a teljes feszültség tartományt három logikai szintre bontják. Lesz egy magas feszültség sáv, egy alacsony feszültség sáv és egy tiltott sáv (1. Ábra). Ha a feszültség értéke a magas sávban van, akkor a digitális rendszer azt logikai 1-esnek fogja tekinteni. Hasonlóan, ha a feszültség szint az alacsony sávban van, akkor az egy 0-s értékként fog megjelenni a rendszerben. Ahogy a neve is jelzi, a rendszer feszültség értéke nem eshet a tiltott sávba, mert ebben az esetben nem lehet eldönteni, hogy az a feszültség érték 0-át vagy 1-et reprezentál. Ha ilyen eset lép fel, az a rendszer hibás működését fogja eredményezni.

A három sáv kialakításán túl, a logikai áramkör eltérő bemeneti (Voltage-Input-High: **VIH,** Voltage-Input-Low: **VIL**) és kimeneti (Voltage-Output-High: **VOH,** Voltage-Output-Low: **VOL**) feszültségszinteket használnak (9. Ábra). Az elérő ki és bemeneti szintek is növelik az áramkör zajjal szembeni ellenálló képességét.



9. Ábra. A ki és bemeneti feszültség szintek szemléltetése.

Az áramkörön belül, az egymáshoz kapcsolt logikai kapuknak egymással kompatibilis logikai szinteket kell alkalmazniuk. Ebből adódóan a kapukat úgynevezett logikai családokba lehet szervezni. Az 1990-es évek végéig 4 ilyen alakult ki: TTL (Transistor-Transistor Logic), CMOS (Complementary Metal-Oxid Semiconductor), LVTTL (Low Voltage TTL), LVCMOS (Low Voltage CMOS). Ezeknek az áramköri családoknak a tulajdonságait foglalja össze a 6. Táblázat.

Logikai család	V_{DD}	V_{IL}	V_{IH}	V_{OL}	V_{OH}
TTL	5 (4.7-5.28)	0.8	2.0	0.4	2.4
CMOS	5 (4.5-6)	1.35	3.15	0.33	3.84
LVTTL	3.3 (3-3.6)	0.8	2.0	0.4	2.4
LVCMOS	3.3 (3-3.6)	0.9	1.8	0.36	2.7

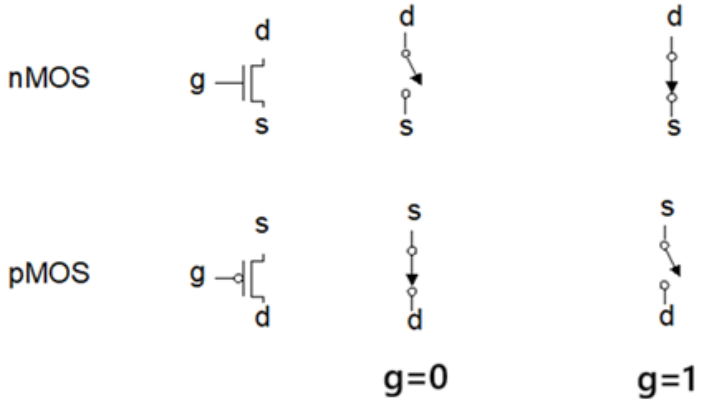
6. Táblázat. Logikai családok jellemzői.

4.5 Logikai kapuk tranzisztorokból

A korábban bemutatott elemi logikai kapuk mindegyikét meg lehet valósítani tranzisztorokból. A tranzisztorokat két fő kategóriába tudjuk sorolni a szerkezeti felépítésük alapján: **bipoláris tranzisztorok** és **térvezérlésű tranzisztorok**. Ennek a jegyzetnek a keretein belül, mi csak térvezérlésű tranzisztorokkal fogunk foglalkozni (és egyszerűen tranzisztorként fogok rájuk hivatkozni), amik a digitális áramkörök legkisebb építő elemei. A digitális áramkörök szempontjából a tranzisztorok vezérelhető kapcsolók, amik két állapotban lehetnek: nyitott vagy zárt. A térvezérlésű tranzisztorok kétféle változatban érhetőek el: n-csatornás (**nMOS**) és p-csatornás (**pMOS**). Mindkét típusnak három kivezetése van: **kapu (g), forrás (s) és nyelő (d)**. A kapura kapcsolt feszültséggel lehet beállítani azt, hogy a tranzisztor milyen állapotba legyen. Ezt leképezhetjük bináris értékekre (3.8 fejezet), így azt is mondhatjuk, hogy a tranzisztor kapuján beállított bináris értékkel tudjuk a tranzisztort vezérelni. Az egyik fő különbség az nMOS és pMOS tranzisztorok között ebben rejlik, amit a 10. Ábra is szemléltet. Elsőként azt figyeljük meg, hogy mi az eltérés a tranzisztorok szimbólumai között. Ahogy látható a pMOS esetében a kapu (g) bemenet egy „buborékkal” van ellátva, ami arra utal, hogy a kapu **negatív logikát** használ. Ezt leegyszerűsítve úgy fogalmazhatjuk meg, hogy az adott bemenet logika 0 értékkel aktiválható. Tehát a pMOS tranzisztor akkor lesz zárt állapotban, ha a kapu bemenete logikai nulla értéket kap, míg az nMOS tranzisztor akkor lesz zárt állapotban, ha a kapu bemenetén logikai 1 értéket kap.

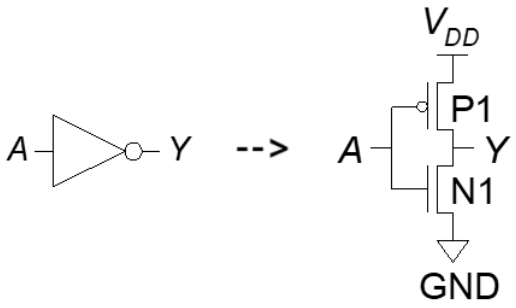
Miután megismerkedtél a kétféle térvezérlésű tranzisztorral, a segítségükkel elkezdheted a logikai kapuk tervezését. A két eltérő tranzisztor jellemzőiből adódóan, az p típusú tranzisztorokat a kapu kimenetének a tápfeszültségre kapcsolásához használják. Ezzel lehet elérni, hogy a kapu kimenetén logikai 1-es jelenjen meg. Ezzel szemben az n típusú

tranzisztorokat a kapu kimenetének földre (GND) kötésekor alkalmazzák (ekkor a kapu kimenete 0). Nézzünk erre egy példát.



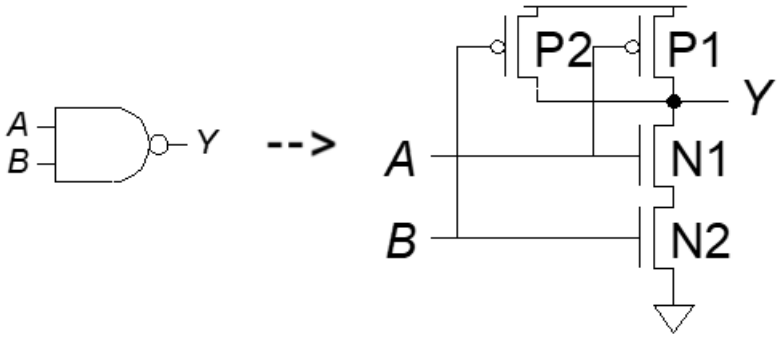
10. Ábra. nMOS és pMOS tranzisztorok működése.

6.1 Példa: Tervezz meg egy inverz kaput tranzisztorokból.



A 6.1 példában az A bemenet 0 vagy 1 értékű lehet. Ha A = 0, akkor a P1 tranzisztor zár, míg az N1 kinyit. Ekkor az Y kimenetet felköti a V_{DD}-re, ami logikai 1-es kimeneti értéket jelent. Ha A = 1, akkor P1 nyit, N1 zár. Ekkor az Y kimenet a GND-re lesz kötve, ami logikai 0 kimenetet eredményez. Nézzünk az előzőnél egy összetettebb példát.

6.2 Példa: Építsünk meg egy kétbemenetű NAND kaput tranzisztorokból.



A 6.2 példában két bemenet van, ezért itt már 4 darab tranzisztort kell felhasználni a kapu megvalósításához. A NAND kapuról már tudod, hogy csak akkor ad a kimenetén logikai 0

értéket, ha minden bemenetén 1-et kap. Itt a két bemenet két feltételnek tekinthető, amelyek közül mindkettőnek teljesülnie kell ahhoz, hogy a kimenet 0 legyen. Ez az oka annak, hogy az n típusú tranzisztorokat sorba kötöttük. Ezzel szemben, ahhoz, hogy a kimenet 1-es legyen elég, ha a bemenetek közül legalább az egyik 1-es. Ebből adódik, hogy a p típusú tranzisztorok párhuzamosan lettek bekötve. Tehát a tranzisztorok sorosan és párhuzamosan is beköthetők. Ha sorba kötjük, akkor minden tranzisztornak zárt állapotban kell lenni, ahhoz, hogy összeköttetés jöjjön létre a kimenet és a tápfeszültség valamelyik pólusa között. Ezzel szemben, ha párhuzamosan kötjük a tranzisztorokat, elég, ha az egyik kerül zárt állapotba, a kimenet, és a tápfeszültség valamelyik pólusa közötti kapcsolat kialakulásához.

Ha egy adott bemeneti kombináció hatására a V_{DD}-hez és a GND-hez is létrejön egy összeköttetés a kimenettel, akkor rövidzár keletkezik, ami tönkre teszi az áramkört. Egy másik problémás eset, ha egy adott bemeneti kombináció esetén a kimeneten sem a V_{DD}-hez, sem a GND-hez nem alakul ki összeköttetés. Ekkor a kapu kimenete **lebeg** és ez egy véletlen kimeneti értéket fog eredményezni.

Összegezve, ha logikai kapukat építesz tranzisztorokból, akkor mindkét tranzisztor típusra támaszkodni kell és ennek alapján az áramkör két részre bontható: felhúzó és lehúzó. Ahhoz, hogy elkerüld az imént említett hibás működést, ha az egyik áramkörrésznél a tranzisztorokat sorba kapcsolod, akkor a másik résznél a tranzisztorokat párhuzamosan kell bekötni.

7. Logikai áramkörök csoportosítása

A digitális technikában egy áramkör felfogható úgy, mint egy hálózat, amely **diszkrét változókkal** dolgozik. Diszkrét változókat kap a bemenetén és diszkrét változókat generál a kimenetén. A kimenet meghatározásához, az aktuális bemeneti értékek mellett a korábbi bemeneti értékeket is figyelembe lehet venni. Ennek alapján a logikai áramköröket két nagy csoportra lehet bontani: **kombinációs és szekvenciális**. A kombinációs logikai áramkörök kimenete csak az áramkör aktuális bemeneteitől függ. Ezt úgy is tekinthetjük, mint egy függvényt, ami megkapja a bemeneteket argumentumként és a bemeneti értékek alapján visszaad egy kimeneti értéket. Ha több értéket is vissza kell adni az áramkörnek, akkor minden kimenethez fel lehet írni egy-egy függvényt. Valójában a korábban bemutatott logikai kapuk is kombinációs logikai áramkörök.

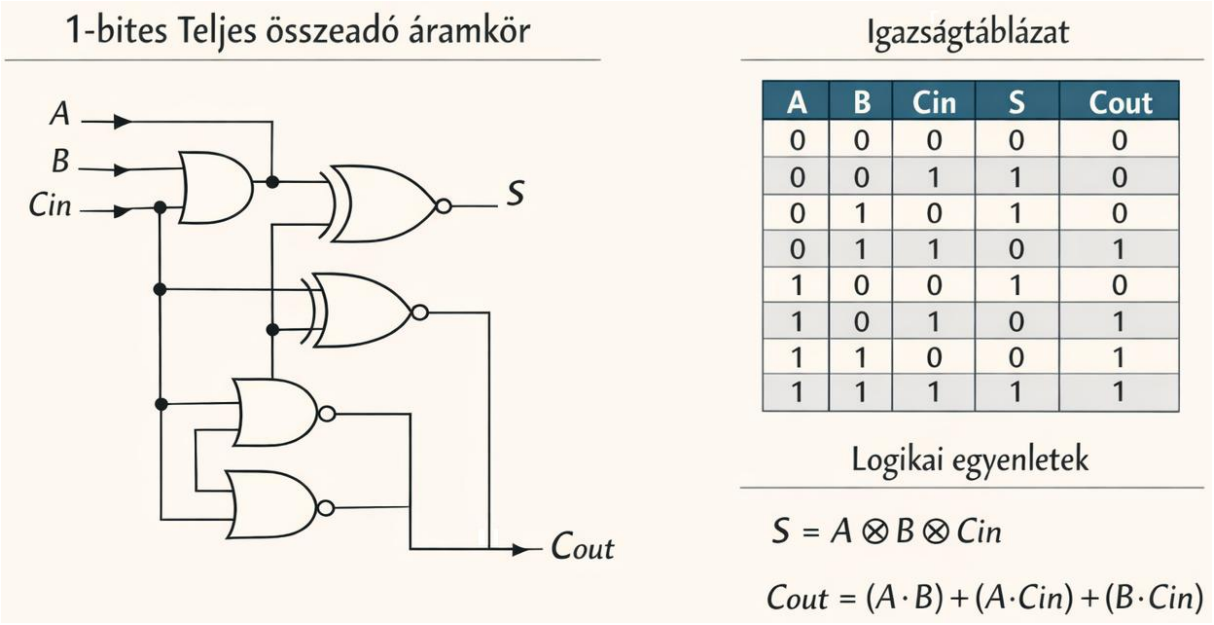
A kombinációs logikai áramkörökkel szemben a szekvenciális logikai áramkörök kimenete az aktuális bemeneti értékeken túl még függ az áramkör korábbi bemeneti értékeitől is. Ezt a függést a szekvenciális logikai áramkörben megtalálható **memória** biztosítja. Tehát, a szekvenciális logikai áramkör memóriával is rendelkezik, míg a kombinációs áramkörökben nincs memória.

Egy kombinációs logikai áramkör működése három féle módon is megadható: **igazságtáblával**, **logikai egyenlettel** és az áramkör **kapcsolási rajzával**. Erre mutat példát a 10. Ábra, ahol egy 1-bites összeadó áramkör rajzát láthatod, alatta a kimenetek logikai egyenletével és jobb oldalon az áramkör igazságtábláját.

A kombinációs logikai áramköröket össze tudjuk kapcsolni, annak érdekében, hogy bonyolultabb áramköröket hozzunk létre. A több komponensből álló áramköröket, kombinációs áramkörnek tekintünk, ha teljesülnek a következő megkötések:

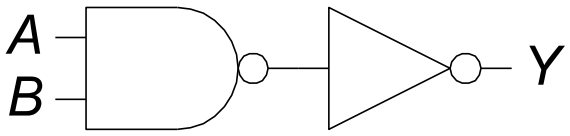
1. Minden egyes komponense kombinációs áramkör.
2. Nem tartalmaz visszacsatolást.
3. Minden egyes bemenet csak egyetlen bemeneti változót kap.

Ahogy korábban említettem, a kombinációs logikai áramkörök működése többféle módon leírható és az egyiket át lehet konvertálni a másikba. Ehhez kapcsolódóan, először azt nézzük meg, hogy hogyan lehet meghatározni az áramkör igazság tábláját az áramkör logikai áramköréből.



10. Ábra. Egy 1-bites összeadó áramkör leírásának lehetséges módjai.

7.1 Példa: Határozd meg a lenti áramkör igazságtábláját.

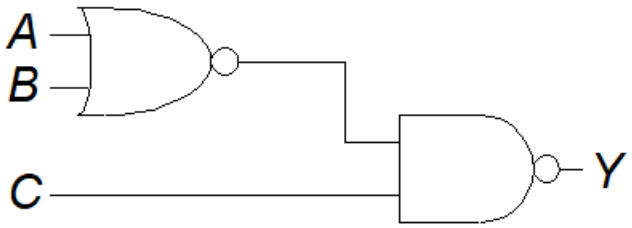


A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

A fenti feladatban az első lépés, hogy felrajzolod az igazságtábla „vázát”, amiben minden egyes bemeneti és kimeneti változóhoz létre kell hozni egy oszlopot. A táblában a bemeneti változók oszlopai a bal oldalon, míg a kimeneti oszlopok a jobb oldalon helyezkednek el. Ha megvan a tábla „sablonja”, a táblába be kell írni az összes lehetséges bemeneti kombinációt. A fenti példában két változó van. Ez $2^2 = 4$ kombinációt ad, aminek a tartománya [00, 01,10,11]. A táblázatba bemeneti kombinációkként, a tartomány elemeit kell beírni, a legkisebb elemétől haladva egyesével a tartomány legnagyobb eleme felé. A

következő lépésben be kell azonosítani a kapuk típusát. A példában az első kapu egy NAND kapu, míg azt követi egy NOT kapu. A kapuk ismeretében meg kell határozni minden egyes bemeneti kombinációhoz a kimeneti értéket és ez kerül az igazságtábla kimeneti oszlopának celláiba.

7.2 Példa: Határozzuk meg a lenti áramkör igazságtábláját.



A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

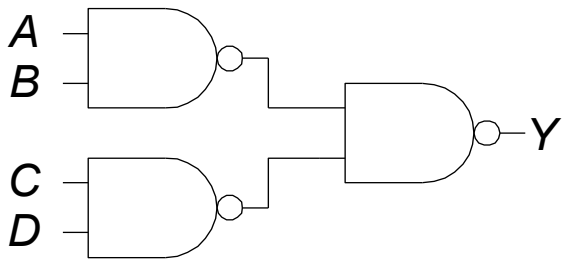
8.Boole algebra

Korábban említettem, hogy a kombinációs logikai áramkörök működése több módszerrel is leírható. Ezek közül az egyik a Boole algebrai egyenletek, ahol a változók csak két értéket vehetnek fel: $A \in \{0,1\}$. Az egyenletekben a változók invertált formában is szerepelhetnek. A változó invertáltját, a változó **komplemensének** is szoktuk nevezni, míg a **literál** kifejezés a változóra vagy annak komplemensére utal.

A Boole algebrai egyenletek segítségével képesek vagyunk formulákkal (függvényekkel) leírni egy kombinációs logikai áramkör minden egyes kimeneti értékének kiszámítását. Egy egyenlet a kimenetet befolyásoló bemeneti változókból és a közöttük elvégzendő logikai műveletekből tevődik össze. A hagyományos algebrához hasonlóan itt is megvan az egyes műveletek végrehajtási sorrendje, amit **precedenciának** is neveznek. Ez a sorrend a következő: invertálás, AND, OR, XOR. Fontos kihangsúlyozni azt, hogy a Boole algebra változói csak két értéket vehetnek fel, szemben a hagyományos algebrával, ahol egy változó végtelen sok értéket tud felvenni. Továbbá, jegyezd meg, hogy a Boolean algebrában a szorzás logikai AND műveletnek felel meg, míg az összeadás logikai OR műveletnek!

Az előző fejezet példáiban láttad, hogy hogyan lehet az áramkör rajzából az igazságtáblát meghatározni. A következő példákban azt nézzük meg, hogy hogyan lehet meghatározni az áramkör rajza alapján az áramkör egyenletét. Ezeknél a feladatoknál is fel kell ismerni az áramkört alkotó logikai kapukat. Ha ez megvan, akkor a bemenetektől kezdődően egyesével haladunk végig a logikai kapukon a kimenet irányába és felírjuk a kapuk kimenetére a kapu logikai egyenletét.

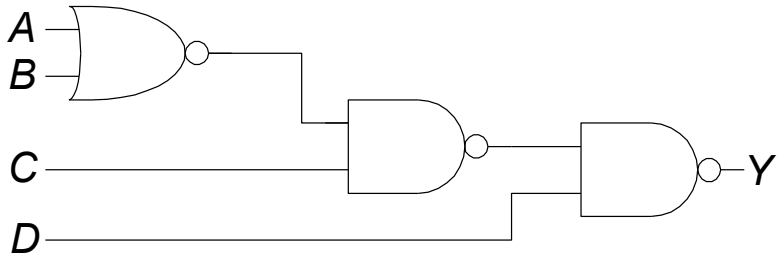
8.1 Példa: Határozd meg a lenti áramkör logikai egyenletét.



$$\boldsymbol{Y = \overline{(\overline{AB})(\overline{CD})}}$$

Az előző példában a bal felső logikai kapu egy NAND kapu, aminek a kimenetét az $\overline{AB}$ képlettel tudjuk leírni. Az alatta lévő kapu szintén NAND, aminek a kimenetét a $\overline{CD}$ írja le. Végül az utolsó kapu is egy NAND, ami bemenetként az előző két kapu kimenetét kapja meg. Tehát az $\overline{AB}$ és a $\overline{CD}$ kell összeszorozni és ennek venni az invertáltját.

8.2 Példa: Határozd meg a lenti áramkör logikai egyenletét.



$$\boldsymbol{Y = \overline{\left(\overline{(\overline{A+B})C}\right)D}}$$

8.1 SOP egyenlet

Most már tudod, hogy a kombinációs logikai áramkörök működését három módszerrel is le lehet írni és ezek közül az egyik, a Boole egyenletek. Mivel ugyan azt a működési elvet lehet leírni mindhárom módszerrel, így az egyik leírási mód átkonvertálható a másikba. Ebben az alfejezetben azt nézzük meg, hogy hogyan lehet egy igazságtáblából úgynevezett **SOP** (sum of products) egyenletet létrehozni. Az SOP kifejezés az egyenlet alakjára utal, ahol a változók szorzatainak vesszük az összegét. Vegyük példaként a lenti igazságtáblát, ahol két változónk van. A két változón összesen 2^2 bemeneti kombinációt tudunk felírni, így az igazságtáblának 4 sora lesz, benne a bemeneti kombinációk tartományának elemeivel, a legkisebb elemtől kezdődően a tartomány legnagyobb eleméig. Az igazságtábla minden sorához meg tudunk szerkeszteni egy úgynevezett **mintermet**, amely az összes bemeneti változó között vett szorzat. A változókat olyan formában (eredeti vagy az invertált (negált)) kell elhelyezni a szorzatban, hogy a szorzat igaz (1) értéket adjon. Szorzatok esetében, ez csak úgy állhat elő, hogy azt a változót, ami az adott sorban 0 értékű, negálva veszed fel a szorzatba. Az SOP egyenletek ezekre a mintermekre támaszkodnak, de a mintermek közül csak azokra van szükség az SOP egyenletbe, amelyeknél a kimeneti érték 1 (Y = 1). Ennek alapján, a lenti igazságtáblához létrehozott SOP egyenlet a következő: $Y = \bar{A}\bar{B} + A\bar{B}$.

A	B	Y	minterm
0	0	1	$\bar{A}\bar{B}$
0	1	0	$\bar{A}B$
1	0	1	$A\bar{B}$
1	1	0	AB

Itt az egyenlet két szorzattagból áll, mivel az igazságtáblának két sorában volt 1-es a kimeneti érték. Ezekből a sorokból vettem ki és adtam össze a mintermeket. Fontos kihangsúlyozni azt, hogy egy SOP egyenlet megszerkeszthető bármilyen igazságtáblához függetlenül az igazságtáblában szereplő bemeneti változók számától.

8.3 Példa: Határozd meg az SOP egyenletet a lenti igazságtáblához.

A	B	Y	minterm
0	0	1	$\bar{A}\bar{B}$
0	1	1	$\bar{A}B$
1	0	0	$A\bar{B}$
1	1	1	AB

$$Y = \bar{A}\bar{B} + \bar{A}B + AB$$

8.2 POS egyenlet

Egy másik lehetséges egyenlet forma, amit előállíthatunk az igazság táblázatokból az a **POS** (product of sum). Ahogy a név is jelzi, az egyenlet, összegek szorzataként áll elő. Szemléltetésként vegyünk itt is egy igazságtáblát. Az igazságtábla minden sorához meg lehet szerkeszteni egy úgynevezett **maxtermet**, ami az igazságtábla változóinak az összege. A maxtermeket úgy kell kialakítani minden egyes sorban, hogy az hamis (0) értéket adjon. Vedd észre, hogy a maxterm csak akkor lehet hamis, ha azokat a változókat, amik az adott sorban 1-es értékűek, negáltan használod fel a maxtermben. A POS egyenletbe azoknak a maxtermeknek vesszük a szorzatát, ahol a kimeneti érték 0 (Y = 0).

Ennek alapján a lenti igazságtáblához megszerkesztett POS egyenlet a következő: $Y = (A + \bar{B})(\bar{A} + \bar{B})$. A POS egyenletre is igaz, hogy akár egyetlen változó is lehet az egyenlet egy tagja és a negálás művelet csak a változókban jelenhet meg.

A	**B**	**Y**	**maxterm**
0	0	1	$A + B$
0	1	0	$A + \bar{B}$
1	0	1	$\bar{A} + B$
1	1	0	$\bar{A} + \bar{B}$

A fentieket összefoglalva, egy igazságtáblához mind az SOP, mind a POS egyenletformát használhatjuk a Boole egyenlet megszerkesztéséhez. Ebből fakadóan az egyenletek egymással ekvivalensek lesznek, még akkor is, ha kinézetre teljesen eltérőek. Mivel az SOP és POS egyenletek megszerkesztésének bemutatásához ugyan azt az igazságtáblát használtam, ezért ez könnyen ellenőrizhető: $Y = \bar{A}\bar{B} + A\bar{B} = (A + \bar{B})(\bar{A} + \bar{B})$. Felmerülhet az a kérdés, hogy a két egyenlet közül melyiket érdemes választani? A válasz, hogy azt, amelyik rövidebb (kevesebb tagból álló). Általánosan az mondható el, hogy az SOP egyenletet akkor érdemes választani, ha az igazságtábla kimeneti oszlopában kevesebb az 1-esek száma, mint a 0-s. Ellenkező esetben a POS egyenlet lesz a rövidebb.

8.3 Egyenletek egyszerűsítése

Az előbb megismert SOP és POS egyenletszerkesztés a legtöbb esetben nem optimalizált egyenleteket eredményeznek. Ez egyszerűen annyit jelent, hogy ezeket az egyenleteket át lehet írni egyszerűbb alakra. Ahhoz, hogy ezt megtedd, ismerned kell a Boole algebra **axiómáit** és **tételeit**. Hasonlóan a hagyományos algebrához, a Boole algebra segítségével képesek leszünk a kiindulási egyenleteket más alakba átírni. Később látni fogod, hogy az egyenletekből létre lehet hozni egy áramkört, ami az egyenlet alapján működik. Ha tudjuk csökkenteni az egyenlet méretét, akkor ezzel az áramkör is egyszerűsödik. Ha tudjuk egyszerűsíteni az áramkört, akkor ezzel az áramkör költségét és energiafelhasználását is csökkentjük és sok esetben a sebességét is növeljük.

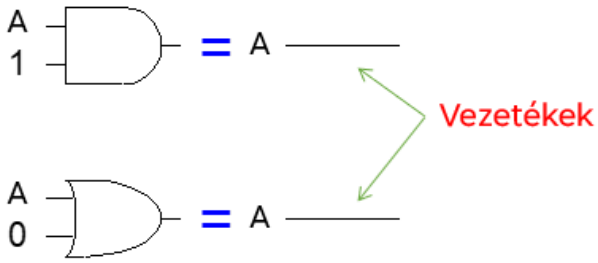
Először nézzük a Boole algebra axiómáit, amik olyan alapvető tételek, amelyek nem igényelnek bizonyítást. A Boole algebra 5 axiómáját és azok duális párjait mutatja be a 7. Táblázat. Figyeld meg, hogy minden axiómának megvan a duális párja. Egy adott axiómából a duálist párt úgy lehet előállítani, hogy lecseréljük a 0-t 1-re, az 1-et 0-ra, a *-ot +-ra és a + operátort *-ra. Vegyük például az A3-al jelölt axiómát. Ha a 0-kat lecserélem 1-re és a * operátort +-ra, akkor megkapom az A3'-t. Az axiómák közül az első leírja, hogy egy változó vagy 0 vagy 1 értéket vehet fel. Az A2-es axióma és duális párja az inverz műveletet definiálja, míg a többi axióma az AND és OR műveleteket írja le.

	Axióma		Duális pár	Név
A1	**A = 0 ha A ≠ 1**	A1'	**A = 1 ha A ≠ 0**	Bináris mező
A2	$\overline{\mathbf{0}} = \mathbf{1}$	A2'	$\overline{\mathbf{1}} = \mathbf{0}$	NOT
A3	**0 * 0 = 0**	A3'	**1 + 1 = 1**	AND/OR
A4	**1 * 1 = 1**	A4'	**0 + 0 = 0**	AND/OR
A5	**0 * 1 = 1 * 0 = 0**	A5'	**1 + 0 = 0 + 1 = 1**	AND/OR

7. Táblázat. Boole algebra axiómái.

Az axiómák után folytassuk a Boole algebra egy-változós tételeivel, amit a 8. Táblázat tartalmaz. Röviden nézzük végig a jelentésüket. A T1-es tétel az állítja, hogy ha az AND műveletnél az egyik operandus 1-es, akkor a kimenet csak a másik operandustól függ. A duális párja ugyan ezt szemlélteti OR műveletre, ha az egyik operandus 0. A T2 tétel azt mondja, hogy ha az AND műveletnél az egyik operandus 0, akkor az eredmény is 0 függetlenül a másik operandustól. Az OR műveletnél, ha az egyik operandus 1, akkor az eredmény is 1 függetlenül a másik operandustól. A T3-as tétel azt mutatja, hogy ha az AND és az OR műveleteknél a két operandus ugyan az, akkor el sem kell végezni a műveletet, mert a műveletek eredménye megegyezik a változó értékével. A T4 a dupla tagadás tétele, miszerint, ha kétszer tagadunk egy változót, akkor visszakapjuk az eredeti értékét. Végül a T5-ös tétel arra mutat rá, hogy ha egy változó és annak invertáltja között végzünk AND műveletet, annak eredménye mindig 0, míg az OR művelet eredménye mindig 1 lesz. Az egyváltozós tételek esetében fontos azt látni, hogy a T1, T2, T3 és T5 tételek alkalmazásával megspórolunk egy logikai műveletet (kaput), míg a T4-es tételnek

köszönhetően két egymást követő NOT kaput tudunk kihagyni az áramkörből. Ennek szemléltetéseként tekintsd meg a 11 Ábrát, ahol a T1-es tétel felhasználásával a logikai kaput helyett egy vezetéket használtunk fel az áramkörben, amely a változó értékét szállítja.



11. Ábra. T1-es tétellel elért egyszerűsítés áramkörre kifejtett hatása.

	Tétel		Duális pár	Név
T1	**A * 1 = A**	T1'	**A + 0 = A**	Azonosság
T2	**A * 0 = 0**	T2'	**A + 1 = 1**	Null elem
T3	**A * A = A**	T3'	**A + A = A**	Idempotencia
T4	$\overline{\overline{A}} = A$			Dupla tagadás
T5	**A * $\overline{A}$ = 0**	T5'	**A + $\overline{A}$ = 1**	Komplemens

8. Táblázat. Boole algebra egyváltozós tételei

Végül vegyük sorra a Boole algebra többváltozós tételeit, amit a 9. Táblázatban találsz. Ezek közül a T6 és T7 a **kommutativitás** és **asszociativitás** tételek, amik ismerősek lehet a hagyományos algebrából. A kommutativitás azt mondja, hogy az AND és OR műveleteknél a változók sorrendje felcserélhető és nem befolyásolja az eredményt. Az asszociativitás tételét a szabadon zárójelezhetőség tételeneke is szokták hívni. Ennek értelmében, mindegy, hogy milyen sorrendben végezzük el az AND műveleteket, valamint az OR műveleteket, mert ez nem változtat az eredményen. A T8 a **disztributivitási** tétel, ami azt mutatja meg, hogy hogyan lehet összevonni szorzattagok összegét. Ez a későbbiek során egy igen fontos tétel lesz, amit a Boole algebrai egyenletek egyszerűsítésénél fogunk kihasználni. Fontos megjegyezni, hogy a T8' tétel csak a Boole algebrában érvényes, a hagyományos algebrában nem teljesül! A T9, T10 és T11-es tételek segítségével a

redundáns változókat tudjuk elhagyni az egyenletből. Végül, a **De Morgan** tételek arra világítanak rá, hogy a változók szorzatának invertáltja egyenlő a változók invertáltjainak összegével. Továbbá, a változók összegének invertáltja egyenlő a változók invertáltjainak szorzatával. Áramköri szempontból, a De Morgan tételek azt jelentik, hogy a NAND és NOR kapuk helyettesíthetők negatív OR és negatív AND kapukkal. A „negatív" jelző azt jelenti, hogy a kapuk minden bemenetét invertáljuk.

	Tétel		Duális pár	Név
T6	**A * B = B * A**	T6'	**A + B = B + A**	Kommutativitás
T7	**(A * B) * C = A * (B * C)**	T7'	**(A + B) + C = A + (B + C)**	Asszociativitás
T8	**A * B + A * C = A * (B + C)**	T8'	**(A + B) * (A + C) = A + (B * C)**	Disztributivitás
T9	**A * (A + B) = A**	T9'	**A + (A * B) = A**	Lefedés
T10	**(A * B) + (A * $\overline{B}$) = A**	T10'	**(A + B) * (A + $\overline{B}$) = A**	Kombinálás
T11	**(A * B) + ($\overline{A}$ * C) + (B * C) = A * B + $\overline{A}$ * C**	T11'	**(A + B) * ($\overline{A}$ + C) * (B + C) = (A + B) * ($\overline{A}$ + C)**	Konszenzus
T12	$\overline{A_0 * A_1 * A_2 * \dots} = \overline{A_0} + \overline{A_1} + \overline{A_2} + \cdots$	T12'	$\overline{A_0 + A_1 + A_2 + \cdots} = \overline{A_0} * \overline{A_1} * \overline{A_2} * \dots$	De Morgan
T13	**(A + B) * $\overline{B}$ = A * B**	T13	**A * $\overline{B}$ + B = A + B**	Redundancia

9. Táblázat. Boole algebra többváltozós tételei

A fenti táblázatokban bemutatásra került a Boole algebra legfontosabb axiómái és tételei. Amiről még nem volt szó, az a tételek bizonyítása. Erre két lehetőségünk is van. Az egyik a **teljes indukció** alkalmazása, amely során megvizsgálunk minden lehetséges bemeneti kombinációt és ha az egyenlet bal és jobb oldala is ugyan azokat az értékeket adják minden bemenetre akkor azt mondjuk, hogy a két oldal egyenlő. A másik módszer a tételekre és axiómákra épül és ezek felhasználásával kell átalakítani egy egyenlet egyik oldalát a másik oldal formájába. Nézzünk mindkét módszerre 1-1 példát.

8.4 Példa: Bizonyítsd be az A(A+B) = A tételt teljes indukcióval.

A	B	A+B	A(A+B)
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

8.5 Példa: Bizonyítsd be az A(A+B) = A tételt az axiómák és tételek felhasználásával.

$$A(A+B) = AA + AB \quad \text{(T8)}$$
$$= A + AB \quad \text{(T3)}$$
$$= 1A + AB \quad \text{(T2)}$$
$$= A(1 + AB) \quad \text{(T8)}$$
$$= A1 \quad \text{(T2)}$$
$$= A \quad \text{(T1)}$$

A Boole algebra axiómáira és tételeire támaszkodva egy kiindulási egyenletet át lehet írni minimális (egyszerűbb) alakra. Egy egyenletet akkor nevezünk **minimális**nak, ha a lehető legkevesebb tagból áll és a tagok a lehető legkevesebb változóból állnak. Az egyszerűsítés folyamata általában több lépést igényel, és akár több szabályt is fel kell használni amire megtaláljuk a legegyszerűbb formát. Természetesen a kiindulási egyenlet nem feltétlenül SOP formában lesz, viszont a tételek és axiómák a kiindulási egyenlet alakjától függetlenül alkalmazhatók. Nézzünk két példát az egyenletek egyszerűsítésére.

8.6 Példa: Határozzuk meg az Y = AB + AC + BB + BC egyenletet minimális alakját a Boole algebra szabályainak felhasználásával.

$$AB + AC + BB + BC = AB + AC + B + BC \quad \text{(T3)}$$
$$= AB + AC + B \quad \text{(T9')}$$
$$= AC + B \quad \text{(T9')}$$

8.7 Példa: Határozzuk meg az Y = A(AB + ABC) egyenletet minimális alakját a Boole algebra szabályainak felhasználásával.

$$A(AB + ABC) = A(AB(1 + C)) \quad \text{(T8)}$$
$$= A(AB1) \quad \text{(T2')}$$
$$= A(AB) \quad \text{(T1)}$$

$$= AA(B) \quad \text{(T7)}$$
$$= AB \quad \text{(T3)}$$

Nézzünk egy olyan példákat is, ahol a De Morgan szabályokra támaszkodunk és azok segítségével egyszerűsítjük le a kiindulási egyenletet.

8.8 Példa: Határozzuk meg az $Y = \overline{\overline{(A+B)} + \bar{C}}$ egyenletet minimális alakját a Boole algebra szabályainak felhasználásával.

$$\overline{\overline{(A+B)} + \bar{C}} = \overline{\overline{(A+B)}}\,\bar{C} \quad \text{(T12')}$$
$$= (A+B)C \quad \text{(T4)}$$

8.9 Példa: Határozzuk meg az $Y = \overline{\left(A + B\bar{C}\right) + D\left(\overline{E+\bar{F}}\right)}$ egyenlet minimális alakját a Boole algebra szabályainak felhasználásával.

$$\overline{\left(A + B\bar{C}\right) + D\left(\overline{E+\bar{F}}\right)} = \overline{\left(A + B\bar{C}\right)}\ \overline{D\left(\overline{E+\bar{F}}\right)} \quad \text{(T12')}$$
$$= (A + B\bar{C})\ \overline{D\left(\overline{E+\bar{F}}\right)} \quad \text{(T4)}$$
$$= (A + B\bar{C})\ \left(\bar{D} + \overline{\left(\overline{E+\bar{F}}\right)}\right) \quad \text{(T12)}$$
$$= (A + B\bar{C})\ (\bar{D} + E + \bar{F}) \quad \text{(T4)}$$

8.4 Standard SOP egyenlet

A korábban bemutatott SOP egyenletről már tudod, hogy több szorzattagot foglalhat magába, amiket Boole-összeadással összegzünk. Azt is érdemes megjegyezni, hogy az SOP egyenlet szorzattagjai nem feltétlenül tartalmazzák az egyenlet **tartományának** minden változóját. Az egyenlet tartománya vagy **domain**-ja (◧) azoknak a változóknak a halmaza, amelyek invertált vagy nem invertált formában megjelennek az egyenletben. Szemléltetésként vegyük az $Y = \bar{A}C + A\bar{B}\bar{C}$ egyenletet, aminek a domain-ja: ◧ = {A, B, C}. Az SOP egyenletben olyan szorzattag is előfordulhat, ami egyetlen változóból áll. Ennek a formátumnak egy további megkötése, hogy az invertálás művelet csak a változók szintjén jelenhet meg. Például, egy $\overline{AB}$ kifejezés nem szerepelhet az SOP egyenletben, mert itt az invertálást már a szorzaton kell végrehajtani.

Az SOP egyenlet forma azért nagyon hasznos a gyakorlatban, mert bármelyik SOP formában megadott egyenlethez össze tudunk rakni egy logikai áramkört, ami csak AND és OR kapukból áll. Az egyenlet szorzat tagjait AND kapuval (vagy kapukkal) hozzuk létre,

míg a szorzatok összegzését OR kapuval valósítjuk meg. Az OR kapu az AND kapuk kimenetét kapja meg bemeneti értékkekként.

A **standard SOP** egyenleteknél egy további megkötés az SOP egyenletformátumhoz képest, hogy a domain minden változójának szerepelni kell az egyenlet minden szorzattagjában. A standard SOP alak azért fontos, mert ha az egyenlet ebben a formában van, akkor sokkal egyszerűbben és gyorsabban tudjuk felírni az egyenletből az áramkör igazságtábláját. Ezen felül, később a Karnaugh tábla megszerkesztése is egyszerűbb lesz, amennyiben az egyenlet standard SOP alakban van. Ahhoz, hogy átírjunk egy nem standard SOP formában lévő egyenletet standard formátumúra a következő lépéseket kell követni:

1. Ha az egyenlet egy szorzattagjából hiányzik a domain egy változója, akkor azt a tagot be kell szorozni a változó plusz a változó invertáltjával. Például, ha az A változó hiányzik, akkor $(A + \bar{A})$ -al (a Boole algebrai szabályok alapján $(A + \bar{A}) = 1$ és, ha 1-el szorzok egy tagot, az nem módosít semmit!)
2. Addig kell az 1. lépést ismételni, amíg az eredeti egyenlet minden tagja tartalmazni fogja a domain összes változóját invertált vagy nem invertált alakban. Fontos megjegyezni, hogy minden egyes hiányzó változó bevitele a kiindulási szorzattagba két új szorzattagot fog eredményezni.

8.10 Példa: Határozzuk meg a standard SOP alakot a következő nem standard SOP egyenlethez: $\mathrm{Y} = \mathrm{A}\bar{\mathrm{B}}\mathrm{C} + \bar{\mathrm{A}}\bar{\mathrm{B}} + \mathrm{AB}\bar{\mathrm{C}}\mathrm{D}$.

$$\text{Standard SOP: } \mathrm{Y} = \mathrm{A}\bar{\mathrm{B}}\mathrm{CD} + \mathrm{A}\bar{\mathrm{B}}\mathrm{C}\bar{\mathrm{D}} + \bar{\mathrm{A}}\bar{\mathrm{B}}\mathrm{CD} + \bar{\mathrm{A}}\bar{\mathrm{B}}\mathrm{C}\bar{\mathrm{D}} + \bar{\mathrm{A}}\bar{\mathrm{B}}\bar{\mathrm{C}}\mathrm{D} + \bar{\mathrm{A}}\bar{\mathrm{B}}\bar{\mathrm{C}}\bar{\mathrm{D}} + \mathrm{AB}\bar{\mathrm{C}}\mathrm{D}$$

A fenti példában, venni kell a kiindulási egyenlet minden tagját sorba és meg kell nézni, hogy hiányzik-e a domain valamelyik változója a szorzattagból. A példa első szorzattagjából a D változó hiányzik ezért: $\mathrm{A}\bar{\mathrm{B}}\mathrm{C}\,(\mathrm{D} + \bar{\mathrm{D}}) = \mathrm{A}\bar{\mathrm{B}}\mathrm{CD} + \mathrm{A}\bar{\mathrm{B}}\mathrm{C}\bar{\mathrm{D}}$. A D változó felvétele után két új szorzattagot kaptunk. Ezekben már minden változó szerepel, tehát felvehetjük őket a standard SOP egyenletbe. Majd vettem a következő szorzattagot, amiből a C és a D változó hiányzott. Miután felvettem ezeket a változókat, 4 új szorzattag keletkezett. Végül a kiindulási egyenlet utolsó szorzattagja minden változót tartalmaz, így ezt változatlanul vettem fel a standard SOP egyenletbe.

8.5 SOP egyenletből igazságtábla

Ha létre szeretnénk hozni egy SOP egyenlet igazságtábláját, az első lépés az egyenlet domain-jének meghatározása. Ebből jól látható lesz, hogy hány változó szerepel az egyenletben és értelemszerűen ezeket a változókat az igazságtáblába is fel kell venni külön oszlopokban. Továbbá kell majd egy oszlop a kimeneti értéknek is.
Korábban láttad, hogy egy SOP egyenlet kimenete 1 lesz, ha az egyenlet szorzat tagjai közül legalább az egyik 1 értékű. Ezt kihasználva, ha az egyenlet standard SOP formában van, akkor végig kell menni minden egyes szorzattagján és meg kell határozni, hogy milyen bemeneti kombináció esetén lesz a szorzat tag 1-es értékű. Ha ez megvan, meg kell keresni ezeket a bemeneti kombinációkat az igazságtáblában is és a hozzá tartozó kimeneti változó cellájába 1-es értéket kell írni. Miután az 1-es értékeket beírtuk a megfelelő kimeneti cellákba, a fennmarad kimeneti cellák értékeit töltsd fel 0-val.

8.11 Példa: Határozzuk meg a következő standard SOP egyenlet igazságtáblázatát: $Y = \overline{A}\overline{B}C + A\overline{B}\overline{C} + ABC$

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

A fenti példában figyeld meg, hogy a standard SOP egyenlet 3 szorzattagból állt és így 3 darab 1-es került az igazságtábla kimeneti oszlopába.
Felmerülhet a kérdés, hogy mi van akkor, ha az egyenlet nincs standard SOP alakban. Ilyenkor célszerű az egyenletet átkonvertálni standard formára, mielőtt elkezdjük feltölteni

az igazságtábla kimeneti oszlopát. Azért célszerű az átkonvertálás, mert ha az egyenlet valamelyik szorzat tagjából hiányzik változó, akkor az a szorzattag több bemeneti kombinációra is 1-est ad. Vegyük például az ABC szorzatot, ahol tudjuk, hogy a domain még egy D változót is tartalmaz. Ekkor az ABC szorzattag egyes értéket fog adni az 1111 (ABCD) és az 1110 ($ABC\overline{D}$) kombinációkra is!

9. Egyenletegyszerűsítés Karnaugh táblával

A **Karnaugh tábla** egy másik lehetséges módszer az egyenletek egyszerűsítésére. Itt is fontos kihangsúlyozni, hogy egy minimalizált SOP kifejezés a lehető legkevesebb tagot tartalmazza, a lehető legkevesebb változóval tagonként. Áramköri szempontból, egy minimális SOP kifejezés általában kevesebb logikai kapuval valósítható meg, mint egy standard kifejezés. Ahogy a neve is jelzi, itt grafikusan, táblázatos formában fogjuk reprezentálni a kiindulási egyenlet tagjait.

A Karnaugh táblás egyszerűsítés a T10-es ($AB + A\bar{B} = A$) tételen alapul, ahol a tábla célja az, hogy a kiindulási egyenlet szorzat tagjait, úgy rendezze el a táblázatban, hogy a tábla szomszédos tagjai csak egy változóban térjenek el. Ennek illusztrálására vegyük a következő igazságtáblázatot és készítsünk belőle egy Karnaugh táblát.

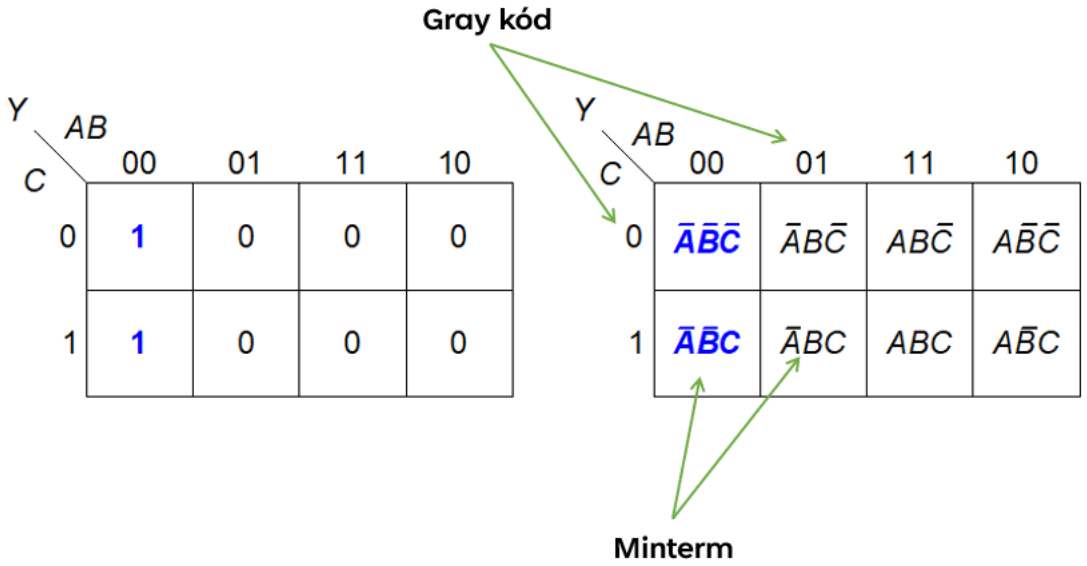
A	**B**	**C**	**Y**
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

A Karnaugh táblának ugyan annyi cellája lesz, mint az igazság tábla sorainak száma. A cellák a tábla kimeneti oszlopának értékeit kapják meg, ahogy az a 12. Ábrán is látható. Az ábrán figyeld meg, hogy az igazság táblában szereplő változók és azok lehetséges kombinációi, a tábla tetején és a bal oldalán vannak feltüntetve. A változók kombinációi **Gray kód** sorrendet követnek. A Gray kódok sajátossága az, hogy a szomszédos bináris kódok csak 1 bitpozíción tértnek egy egymástól. Erre a sorrendre azért van szükség, mert a cellákhoz megszerkeszthetünk egy mintermet (ugyan az a minterm, mint az igazságtábla

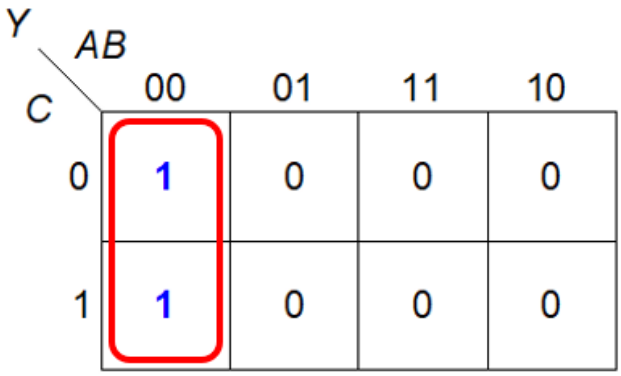
adott soráé), és ez a sorrend biztosítsa azt, hogy a szomszédos mintermek csak egy változóban térnek el egymástól (12. Ábra). Ha egyszerűen bináris sorrendben írnánk fel a változók kombinációit, akkor ez a kritérium nem teljesülne.

A változók elhelyezésének sorrendje tetszőleges, de arra figyelni kell, hogy a cellák tartalmát mindig ennek alapján kell feltölteni. Azt is ki kell emelni, hogy a táblázat első és utolsó sorának elemei, valamint az első és utolsó oszlopának elemei is csak egy változóban térnek el egymástól, így ezeket is szomszédoknak tekintjük. Képzeletben kössük össze a táblázat első és utolsó sorát, valamint az első és utolsó oszlopát. Ez a későbbiekben fontos lesz ahhoz, hogy a Karnaugh tábla alapján meg tudd határozni a minimalizált egyenletet.

A minimalizált egyenlet megtalálásának következő lépése a táblázatban lévő 1-eseket tartalmazó cellák lefedése körökkel (vagy téglalapokkal). A körök kialakítása során figyelembe kell vennünk néhány szabályt. Ez egyik a már korábban említett szomszédsági viszony az első és utolsó sor és oszlop között. A második szabály azt mondja, hogy egy körön belül csak szomszédos cellák lehetnek és a kör méretének a lehető legnagyobbnak kell lenni. Egy további szabály értelmében, a cellák száma 2 hatványa (1, 2, 4, stb.) kell, hogy legyen. Ezek alapján a fenti igazságtáblához megszerkesztett Karnaugh táblában egyetlen kört kell létrehozni, ahogy az a 13. Ábrán is látható.



12. Ábra. Példa Karnaugh táblára.



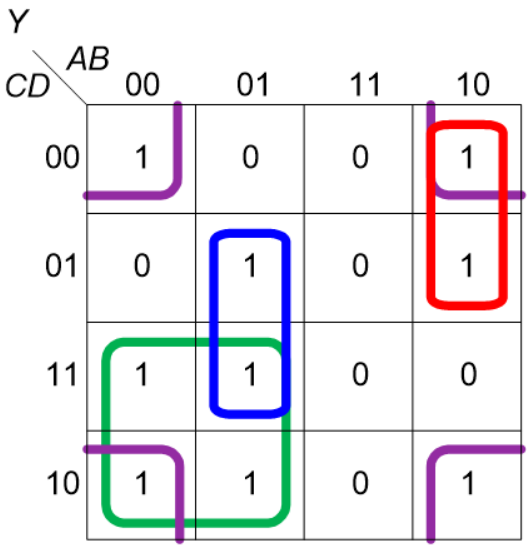
13. Ábra. 1-es értékű cellák lefedése a Karnaugh táblában.

A körrel lefedett két cella mintermjének SOP egyenlete a következő: $Y = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C$. A T10-es tétel használatával, ezt át lehet írni a következő alakba: $Y = \bar{A}\bar{B}(\bar{C} + C) = \bar{A}\bar{B}$. Figyeld meg, hogy az a változó (C), ami a körön belüli cellákban szerepelt normál és invertált alakban is elhagyható a szorzatból. Ennek alapján a körhöz megszerkeszthető egyenlet az $\bar{A}\bar{B}$ szorzat lesz. A minimalizált egyenletbe a körökhöz megszerkesztett szorzatokat kell összegezni, hasonlóan egy SOP egyenlethez. Mivel ebben a példában egyetlen körrel le tudtuk fedni az összes 1-est tartalmazó cellát, így a minimalizát egyenlet: $Y = \bar{A}\bar{B}$. Összefoglalva a Karnaigh tábla kialakításakor és a minimalizált egyenlet meghatározásakor a következő szabályokat kell követni:

1. Találd meg a minimális számú köröket, amik lefedi az összes 1-est.
2. A körön belül az összes cellának 1-est kell tartalmaznia.
3. A körben lévő cellák száma kettő hatványa kell, hogy legyen.
4. Minden körnek olyan nagynak kell lennie, amekkora csak lehet.
5. A kör kialakításakor a Karnaugh tábla vízszintes és függőleges széleit egymás szomszédjainak tekintjük.
6. Egy 1-est tartalmazó cella több körnek is a része lehet.

9.1 Példa: Határozd meg a minimalizált egyenletet a lenti igazságtáblához Karnaugh tábla segítségével. A lent megadott táblában figyeld meg, hogy a négy sarok cella is egymás szomszédjai, ezért az egyik kört ezeken alakítsd ki.

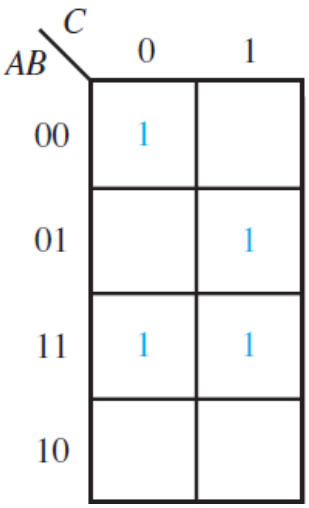
A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0



$$Y = \bar{A}C + \bar{A}BD + A\bar{B}\bar{C} + \bar{B}\bar{D}$$

Természetesen a Karnaugh táblát arra is tudjuk használni, hogy egy kiindulási egyenlethez meghatározzuk a minimalizált alakot. Ekkor első lépésként célszerű megvizsgálni, hogy az egyenlet standard SOP alakban van e. Ha nem akkor alakítsd át standard alakúra. Ezt követően, most is meg kell határozni, hogy az egyenlet szorzattagjai milyen bemeneti kombináció esetén adnak 1-es értéket és a Karnaugh táblában ezeknek a bemeneti kombinációknak a celláiba is 1-es értéket kell írni. Innentől kezdve a feladat ugyan az, mint amit korábban leírtam.

9.2 Példa: Határozd meg a minimalizált egyenletet a következő SOP egyenlethez Karnaugh tábla segítségével: $Y = \overline{A}\overline{B}\overline{C} + \overline{A}BC + ABC + AB\overline{C}$



$$Y = \overline{A}\overline{B}\overline{C} + AB + BC$$

A Karnaugh tábla fő problémája, hogy csak 4 logikai változóval használható kényelmesen. A gyakorlatban sokszor jóval több, mint 4 logikai változóval kell dolgozni, éppen ezért, a Karnaugh táblát elsősorban oktatási célra használják.

Egy másik egyszerűsítési módszer, amelyben nincs megkötés a változók számára a **Quine-McCluskey**. Ez is az egyenlet szorzat tagjainak táblázatos formába rendezésén alapul, amit relatíve egyszerűen lehet algoritmizálni és ez lehetőséget biztosít a szoftveres egyenlet optimalizálásra. A módszer hátránya az idő és memória igénye, ami a változók számának növekedésével exponenciálisan növekszik. A Quine-McCluskey módszer működésének részletes leírása a [2] könyvben található.

A Quine-McCluskey módszer hátrányai miatt, a gyakorlatban az **Espresso** algoritmust használják a logikai függvények minimalizálására, ami egyben az de facto standard. Ebből adódóan a programozható logikai áramkörökhöz készített tervező szoftverek többségében is megtalálható.

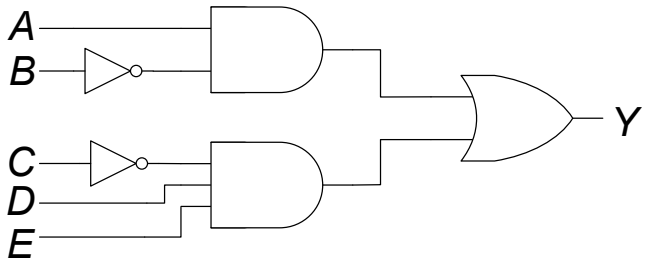
10. Logikai áramkörök kapcsolási rajza

A logikai áramkör rajza magába foglalja az áramkör komponenseit és a komponenseket összekötő vezetékeket. Az áramkör megrajzolása során szem előtt kell tartani néhány alapvető szabályt. Ezek a következők:

1. A bemenetek az áramkör bal oldalán vagy tetején helyezkednek el.
2. A kimenetek az áramkör jobb oldalán vagy alján kapnak helyet.
3. A vezetékeket egyenesen rajzoljuk, illetve elágaztatáskor 90 fokban törjük meg.
4. Ha a vezetékek keresztezik egymást, akkor egy ponttal jelöljük, ha a vezetékek között fizikai kapcsolat van. Különben azt feltételezzük, hogy nincs kapcsolat a vezetékek között (képzeletben az egyik vezeték a másik alatt fut el)
5. A T összeköttetéseknél, mindig azt feltételezzük, hogy a vezetékek között fizikai kapcsolat van.

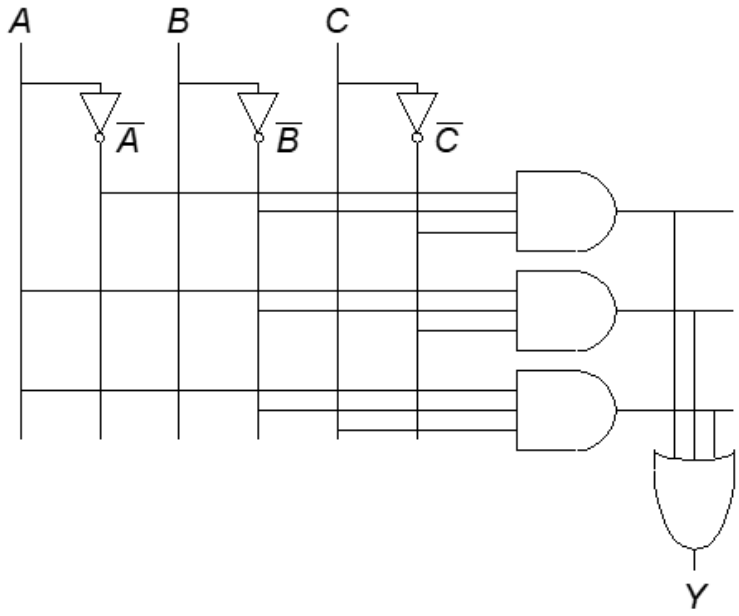
Korábban már volt arról szó, hogy a kombinációs logikai áramkörök működése leírható három különböző módon és az egyik leírási mód, a másikba átkonvertálható. Egy példán keresztül nézzük meg azt, hogy hogyan lehet megszerkeszteni az áramkör rajzát, ha ismerjük az áramkör logikai egyenletét.

10.1 Példa: Határozd meg az áramkör rajzát, a következő logikai egyenlet alapján: $Y = AB + CDE$.



Figyeld meg, hogy az előző példában az egyenlet két szorzat tagja AND kapukkal lett valósítva, míg az összegüket, ami egyben az egyenlet kimenete is, egy OR kapuval határoztuk meg.

10.2 Példa: Határozd meg az áramkör rajzát a következő logikai egyenletnek: $Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$.

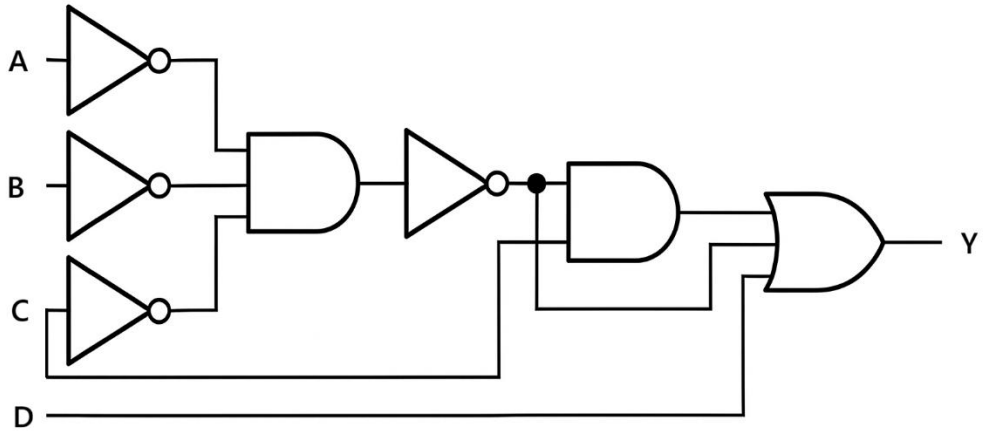


A 10.2-es példa egyenletében három szorzat tag van, amihez három AND kaput kellett felhasználni, míg az összegüket itt is OR kapuval állítottuk elő. Hasonlítsd össze a 10.1-es és a 10.2-es példában megadott áramköri rajzok stílusát. A 10.2-es példában látható áramköri rajz megszerkesztésének stílusát **PLA** (programmable logic array) **stílusnak** nevezik. Itt a bemenetek a rajz tetején helyezkednek el, ahol minden bemeneti változónak a negáltját is előállítjuk egy inverz kapuval. A kimenetek a rajz alján kapnak helyet. A stílus neve onnan származik, hogy a PLA egy olyan programozható logikai áramkör, ami csak AND és OR kapukat tartalmaz. Miért lehet ez hasznos? Gondolj bele abba, hogy az SOP formájú egyenletek szorzat tagok összegeként szerkeszthető meg. Az áramkörben a szorzat tagokat AND kapukkal valósítjuk meg, míg az összegzést OR kapukkal hozzuk létre. Tehát, bármilyen SOP egyenletet megvalósítható AND és OR kapuk felhasználásával! Az SOP egyenleteket **kétszintű egyenleteknek** is szoktuk nevezni, mert az „első szinten" állítjuk elő a szorzattagokat az AND kapukkal, míg a „második szinten" az OR kapuk végzik el az összegzést és adják a kimeneteket. Az SOP mellett a POS egyenletekre is igaz a kétszintű egyenlet kifejezés, viszont itt az első szinten OR kapuk vannak, míg a második szinten AND kapuk foglalnak helyet.

Miután megismerkedtél az áramkörök rajzolásával és a logikai egyenletek egyszerűsítésével, a kettőt kombinálva, lehetővé válik egy kiindulási áramkör optimalizálása és az optimalizált áramkör felrajzolása. Ehhez meg kell határozni a

kiindulási áramkör logikai egyenletét. Ha ez megvan, akkor a tanult módszerek közül valamelyikkel az egyenletet minimalizálni kell. Miután megkaptad a minimalizált egyenletet, ennek alapján fel kell rajzolni az új áramkört.

10.3 Példa: Határozzuk meg az optimalizált áramkör rajzát a lent látható áramkörhöz.



Eredeti egyenlet: $Y = \left(\overline{\bar{A}\bar{B}\bar{C}}\right)C + \overline{\bar{A}\bar{B}\bar{C}} + D$

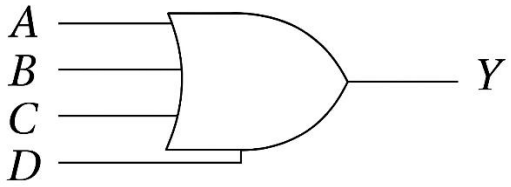
$$Y = \left(\bar{\bar{A}} + \bar{\bar{B}} + \bar{\bar{C}}\right)C + \bar{\bar{A}} + \bar{\bar{B}} + \bar{\bar{C}} + D =$$

$$AC + BC + CC + A + B + C + D =$$

$$AC + BC + C + A + B + D$$

$$C(A + B + 1) + A + B + D$$

$$A + B + C + D$$



11. Többszintű logikai áramkörök

A fent bemutatott kétszintű áramkör felépítése nagyon egyszerű, ezért széles körben használják a gyakorlatban, de nem mindig optimális választás. Erre egy jó példa lehet, a **paritásbit** generáló áramkör megvalósítása, amit egy példán keresztül szemléltetek.

11.1 Példa: AND és OR kapuk felhasználásával implementálj egy páros paritású kódgenerátort 4-bites kódokhoz.

1. Készíts igazságtáblát az áramkör működéséről:

A	B	C	D	P
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

2. Az igazságtáblához határozd meg a minimalizált egyenletet:

$$P = \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}C\bar{D} + \bar{A}B\bar{C}\bar{D} + \bar{A}BCD + A\bar{B}\bar{C}\bar{D} + A\bar{B}CD + AB\bar{C}D + ABC\bar{D}$$

3. Az egyenlet alapján fel kell rajzolni az áramkör rajzát. Anélkül, hogy a rajzot elkészítenénk, jól látható, hogy az áramkörbe 8 darab 4 bemenetű AND kapura lenne szükség a 8 szorzattag előállításához és OR kapukra a szorzat tagok összegzéséhez.

A szemléltetés kedvéért, az előző feladat megoldása lépésekre van bontva. Figyeld meg, hogy az előző feladatban nem tudtuk egyszerűsíteni P egyenletét. De most mi lenne, ha nem AND és OR kapukban gondolkodnánk, hanem mondjuk XOR kapuk felhasználásával készítenénk el az áramkört. Kezdjük azzal, hogy az XOR művelet SOP alakja két változóra a következő $Y = A \oplus B = A\bar{B} + \bar{A}B$. Erre építve próbáljuk meg lecserélni a fenti egyenlet AND és OR műveleteit XOR műveletekre az egyenlet átírásával:

$$P = \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}C\bar{D} + \bar{A}B\bar{C}\bar{D} + \bar{A}BCD + A\bar{B}\bar{C}\bar{D} + A\bar{B}CD + AB\bar{C}D + ABC\bar{D} =$$

$$\bar{A}\bar{B}(\bar{C}D + C\bar{D}) + \bar{C}\bar{D}(\bar{A}B + A\bar{B}) + CD(\bar{A}B + A\bar{B}) + AB(C\bar{D} + \bar{C}D) =$$

$$\bar{A}\bar{B}(C \oplus D) + \bar{C}\bar{D}(A \oplus B) + CD(A \oplus B) + AB(C \oplus D) =$$

$$(C \oplus D)(\bar{A}\bar{B} + AB) + (A \oplus B)(\bar{C}\bar{D} + CD) =$$

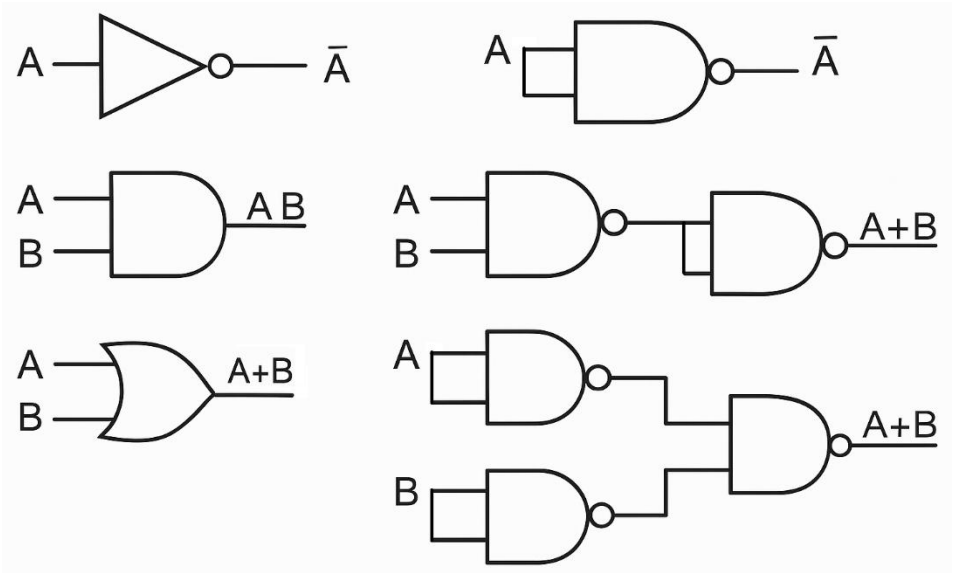
$$(C \oplus D)\overline{(A \oplus B)} + (A \oplus B)\overline{(C \oplus D)} =$$

$$(A \oplus B) \oplus (C \oplus D)$$

Szemben a P kimenetnek a korábbi SOP alakjával, a fent átírt egyenlet csak három XOR kaput igényel, így a kapuk számát tekintve, ez az áramkör jóval kevesebbet kapuból építhető meg.

12. Univerzális Logikai Kapuk

Az elemi logikai kapuk közül két kaput, a NAND és a NOR-t **univerzális kapuknak** nevezzük. Azért univerzálisak, mert kizárólag NAND vagy NOR kapukból az összes többi elemi logikai kapu megépíthető. Mindkét univerzális kapunak létezik egy úgynevezett **duális párja**, aminek a működése az eredeti kapuéval megegyező. A NAND kapu párja a negatív OR kapu, míg a NOR kapu párja a negatív AND kapu. Itt a negatív kifejezés azt jelenti, hogy a kapu mindegyik bemenete invertált. Szemléltetésként tekintsd meg a 14. Ábrát, ahol az látható, hogy a NOT, AND és OR kapukat hogyan lehet megépíteni kizárólag NAND kapuk felhasználásával.



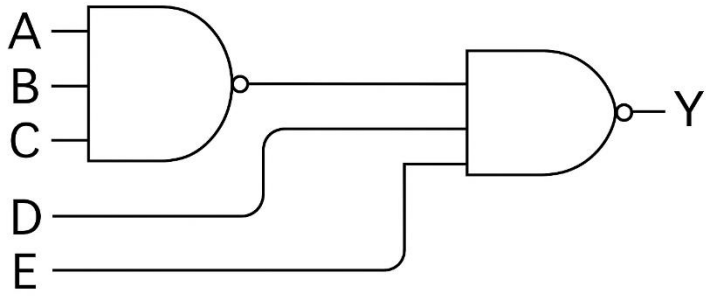
14. Ábra. A NOT, AND és OR kapuk megvalósítása NAND kapukkal.

Mivel minden logikai kapu felépíthető kizárólag NAND vagy kizárólag NOR kapukból, ebből adódóan egy logikai egyenlethez olyan áramköri rajzot is készíthetünk, amely csak univerzális kapukat tartalmaz. Mielőtt nekiesnék egy áramkör felrajzolásának univerzális kapuk használatával, először célszerűbb a kiindulási egyenlet átírása másik alakra, amely alapján az univerzális kapukból álló áramköri rajz sokkal könnyebben megszerkeszthető. Ehhez három egyszerű lépést kell elvégezni:

1. Duplán invertáljuk a kiindulási egyenletet
2. Alkalmazzunk a DeMorgan szabály a belső inverz művelet ketté bontására
3. A dupla negációkat távolítsuk el az egyenletből

12.1 Példa: Határozd meg a következő egyenlet áramköri rajzát, amely kizárólag NAND kapukat tartalmaz: $Y = ABC + \overline{D} + \overline{E}$.

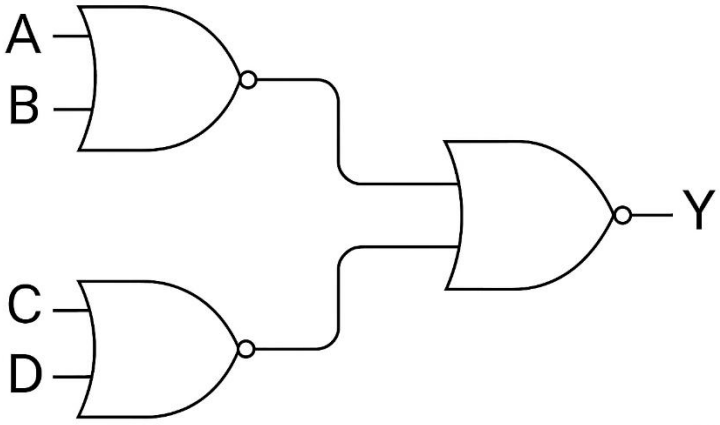
$$Y = ABC + \overline{D} + \overline{E} = \overline{\overline{ABC + \overline{D} + \overline{E}}} = \overline{(\overline{ABC})\overline{(\overline{D})}\,\overline{(\overline{E})}} = \overline{\overline{ABC}DE}$$



A fent leírt lépéssorozat akkor is használható, ha az áramkört úgy szeretnénk megtervezni, hogy az csak NOR kapukat tartalmazzon.

12.2 Példa: Határozd meg a következő egyenlet áramköri rajzát, amely kizárólag NOR kapukat tartalmaz: $Y = (A + B)(C + D)$.

$$Y = (A + B)(C + D) = \overline{\overline{(A + B)(C + D)}} = \overline{\overline{(A + B)} + \overline{(C + D)}}$$

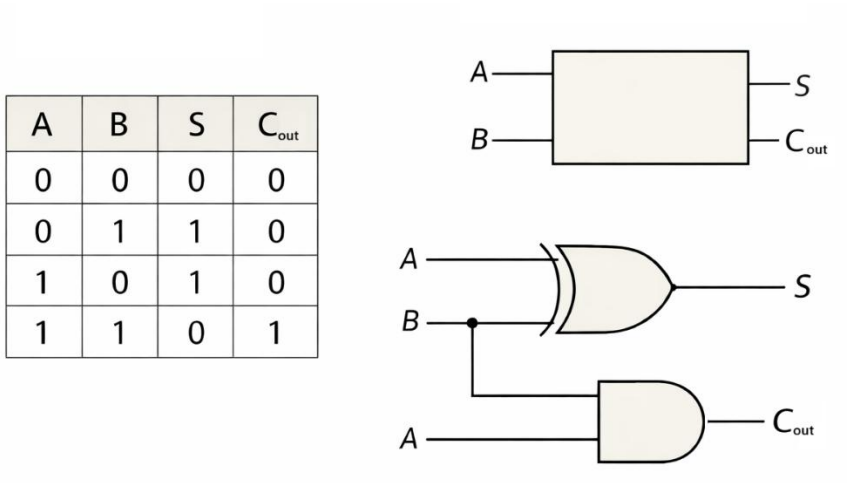


13. Alapvető kombinációs logikai áramkörök

13.1 1-bites összeadók

A számítógépes rendszerekben és azokban a digitális rendszerekben, ahol numerikus értékekkel kell dolgozni, az összeadó áramkörök fontos szerepet töltenek be. Két elemi összeadó áramkört lehet használni egy-bites változók összeadására. Az egyik a **félösszeadó**, amely két egybites bemenettel (A, B) és két egybites kimenettel rendelkezik. Az egyik kimenet az összeg (S), míg a másik az átvitel (carry - C). A félösszeadó áramkör igazságtáblája, szimbóluma és áramköri rajza a 15. ábrán látható. A C kimeneten akkor kapunk 1-est, ha mindkét operandus értéke 1 volt (A=1, B=1). Ebben az esetben, a bináris eredmény $1 + 1 = 10_2$. Tehát az összeg 0 és keletkezett egy átvitel értékünk, amit majd a következő bitpozíciónál kell figyelembe venni, ha több bites számokat adnánk össze. Ne feledd el, hogy az igazságtábla alapján minkét kimenethez fel tudunk írni egy logikai egyenletet (C_{out} = AB, S = A ⊕ B), amelyek felhasználásával megrajzolhatjuk az áramkör kapcsolási rajzát (15. ábra).

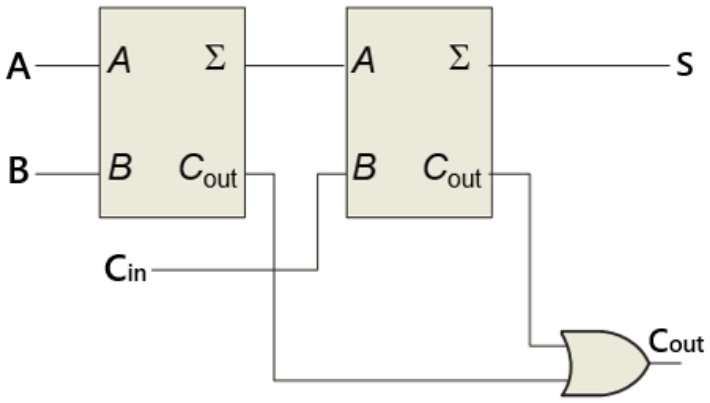
A	B	S	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



15. Ábra. A félösszeadó áramkör igazságtáblája, szimbóluma és áramköri rajza.

A másik elemi összeadó áramkör a **teljes összeadó** (10. ábra). A teljes összeadó is ugyan azzal a két kimenettel rendelkezik, mint a félösszeadó (S és C_{out}), viszont itt három egybites bemenet van. Ezek közül kettő az A és B operandusé, míg a harmadik a carry bemenet (C_{in}). A carry bemenet jelenti a fő eltérést a fél és a teljes összeadó között. Ezzel a

bemenettel lehetővé válik a teljes összeadók láncba kapcsolása és így több-bites összeadók létrehozása. A harmadik bemenet miatt az áramkör kimeneteinek működését leíró logikai egyenletek jócskán eltérnek a félösszeadóéhoz képest: $S = (A \oplus B) \oplus C_{in}$ és $C_{out} = AB + (A \oplus B) C_{in}$. Végül azt is fontos megjegyezni, hogy egy teljes összeadó megépíthető két félösszeadó és egy OR kapu felhasználásával, ahogy azt a 16. Ábra is mutatja.



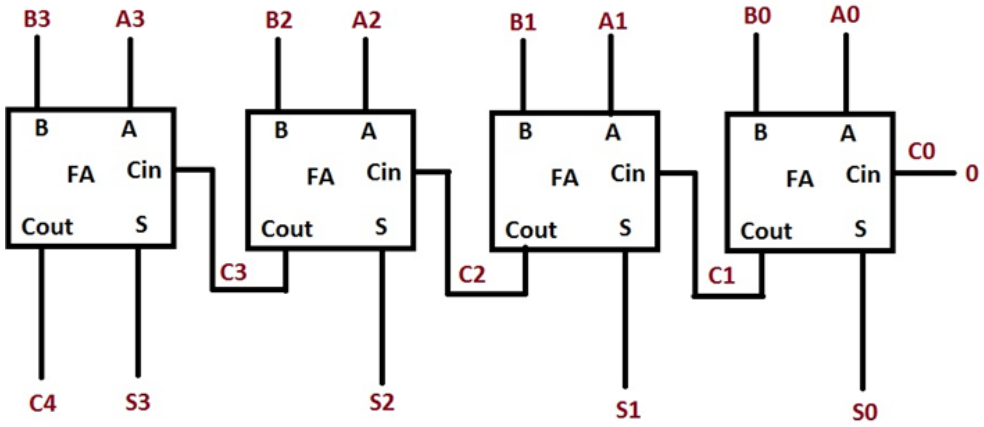
16. Ábra. Teljesösszeadó létrehozása két félösszeadóval.

13.2 Ripple-carry összeadó

Az előző fejezetben ismertetett elemi összeadók önmagukban még nem túl hasznosak, mivel a legtöbb esetben több bites számokkal végzünk aritmetikai műveleteket. N-bites számok összeadására több eltérő működési elvű áramkör létezik. Ezek közül a legegyszerűbb a **Ripple carry összeadó**. Az áramkör sajátossága az, hogy annyi teljes összeadót igényel, ahány bites összeadót szeretnénk létrehozni. Például egy 4 bites operandusokkal dolgozó összeadóhoz 4 teljes összeadót kell láncba kapcsolni. A láncba kapcsolás azt jelenti, hogy a teljes összeadókat sorba rendezzük és egy adott összeadó carry kimenetét a rákövetkező összeadó carry bemenetére kötjük (17. ábra). Ez az összeköttetés biztosítja egy adott bitpozíción keletkező átvitel értéknek a továbbítását, a következő (eggyel magasabb helyiértékű) bitpozíció felé. Innen ered az összeadó áramkör neve is. Mivel az első bitpozíción nincs carry bemenet összeadáskor, így a lánc első eleme akár félösszeadó is lehet (de ha az összeadóval kivonást is végzünk, akkor itt is teljes összeadót kell használni).

Az összeadáshoz kapcsolódóan fontos megjegyezni, hogy ha az operandusok kettes számrendszerben ábrázolt értékek, akkor két N bites szám összeadásának eredménye

legfeljebb N+1 bites lehet. Az összeadó áramkör N+1-edik eredmény bitje a legnagyobb helyi értékű (baloldali) teljesösszeadó carry kimenete, ami így az eredmény MSB bitje is egyben. A gyakorlatban egy jól ismert 4-bites összeadó IC a **74LS283**, amelynek a késleltetési ideje az összeg kimeneten, megközelítőleg 20ns (viszont ez carry look-ahead összeadó és nem ripple-carry). Amennyiben N-bites összeadót szeretnénk létrehozni több 74LS283 IC felhasználásával, akkor a láncba szervezésük logikailag megegyezik a teljes összeadók láncba kapcsolásáéval.

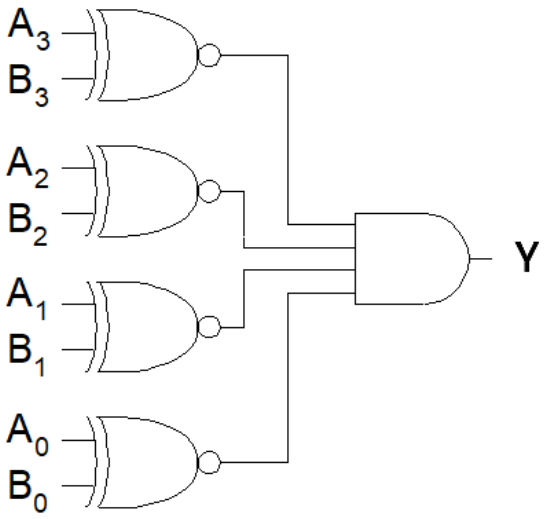


17. Ábra. 4-bites Ripple-carry összeadó.

13.3 Egyenlőség vizsgáló áramkör - komparátor

Egy **komparátor** alapvető feladata két bináris szám nagyságának az összehasonlítása. A legegyszerűbb komparátor csak azt tudja megállapítani, hogy a két operandusa egyenlőek vagy sem. Ilyenkor az összehasonlítás bitenként történik. A bitek egyenlőségének vizsgálatát egy kétbemenetű XNOR kapuval tudjuk megvalósítani, mivel az XNOR kimenete 1-es, ha a két bemenő értéke megegyeik. Ezért is nevezik az XNOR kaput egyenlőség vizsgáló kapunak. Általánosan az mondható el, hogy egy egyenlőséget vizsgáló áramkörbe annyi XNOR kaput kell használni ahány bitesek a bemeneti értékek (operandusok). Ezen felül, azt is ellenőrizni kell, hogy az egyenlőség minden bitpozíción teljesüljön. Erre egy AND kaput használunk, amely bemenetként az XNOR kapuk kimeneti értékét kapja meg. Az AND kapuról már tudod, hogy csak abban az esetben lesz 1-es a kimenete, ha a két operandus minden bitpozíción megegyezett, azaz minden XNOR kapu 1-es értéket generált. Szemléltetésként a 18. ábrán látható egy 4-bites egyenlőségvizsgáló áramkör rajza. A gyakorlatban egy 4-bites komparátort rejt a **74LS85** IC, ami kaszkádolható, így létrehozva nagyobb bitszélességű komparátorokat. A 74LS85 az

összehasonlítás során kapható mindhárom lehetséges kimenetet képes meghatározni, azaz, ha az operandusokat A és B jelöli, akkor: A==B, A > B és A < B.

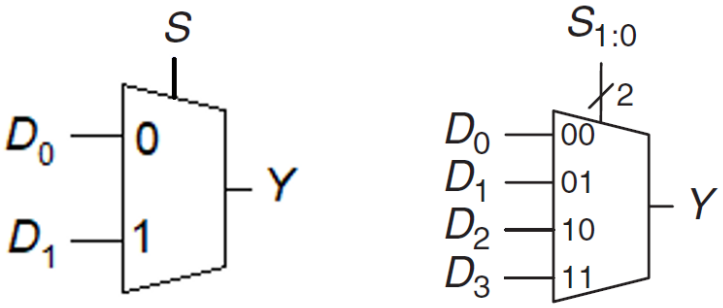


18. Ábra. 4-bites egyenlőségvizsgáló áramkör.

13.4 Multiplexer

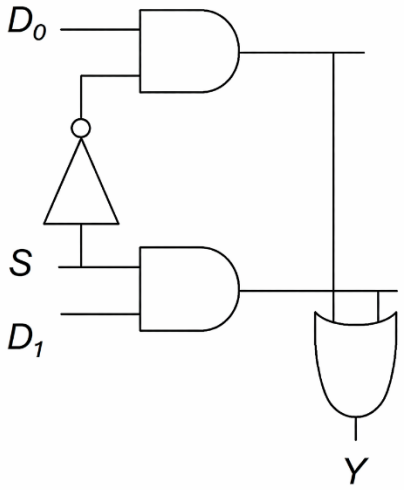
A **multiplexerek (MUX)** az egyik leggyakrabban használt kombinációs logikai áramkörök. A MUX feladata, hogy az N darab bemenete közül kiválasszon egyet és azt engedje tovább a kimenetére. A működési elve alapján, a MUX-t N állásos kapcsolónak is szokták nevezni. Azt, hogy a bemenetek közül melyik lesz a kimenetre tovább engedve, a MUX **kiválasztó (select)** bemenetére adott érték dönti el.

A legegyszerűbb MUX a 2:1-es (19. ábra), ahol az áramkör két adat bemenettel (D0 és D1) és egy kimenettel rendelkezik (Y). Az adatbementeken felül, természetesen ez is rendelkezik kiválasztó bemenettel (S), ami itt 1 bites. A kiválasztó bemenet szélessége (bitekben) attól függ, hogy a MUX hány adatbemenettel rendelkezik. Azt képzeld el, hogy minden adatbemenet egy egyedi kóddal kap és ezt a kódot kell beállítani a kiválasztó bemeneten. Ebből adódóan a kiválasztó bemenet szélességét $\log_2(N)$-re kell választani, ezzel biztosítva, hogy minden adatbemenethez lehessen saját kódot rendelni. A 18. Ábrán látható 2:1-es MUX-nél ha S=0, akkor az $Y = D_0$, máskülönben $Y = D_1$.



19. Ábra. 2:1-es multiplexer (bal oldalt) és 4:1-es multiplexer.

Az absztrakciót kihasználva, a multiplexereknek saját szimbóluma van a digitális technikában, de tartsd észben, hogy a multiplexerek is elemei logikai kapukból épülnek fel. Ezt szeretné szemléltetni a 20. Ábra is, ahol a 2:1-es multiplexer felépítése látható.



20. Ábra. 2:1-es multiplexer megvalósítása logikai kapukkal.

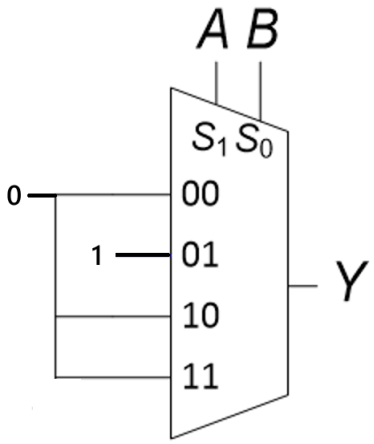
A gyakorlatban sokszor használnak multiplexereket az osztott csatornák kezelésére. Egy osztott csatornához több forrás csatlakozik, de ezek közül, egy adott időpillanatban, csak egy használhatja a csatornát. A csatornamegosztásre jó példa lehet, egy olyan a mikrovezérlő, aminek csak egy analóg-digitális átalakításra képes bemenete van (ami a szenzorból jövő feszültséget egy számmá alakítja), de több analóg szenzort szeretnék hozzákötni. Ezt egy MUX-al meg lehet oldani, ahol a szenzorok kimenetét a MUX bemeneteire kötjük, míg a MUX kimenetét a mikrovezérlő analóg-digitális átalakításra képes bemenetére. A MUX vezérlő bemenetét a mikrovezérlővel irányítva mindig ki lehet választani, hogy melyik szenzortól érkező jel legyen digitalizálva.

A szemléltetés kedvéért nézzünk egy 4:1-es MUX-ot is (19. Ábra), amely már négy adatbemenettel (D0, D1, D2, és D3) rendelkezik. Mivel itt négy adatbemenethez kell egyedi kódot rendelni, így a kiválasztó bemenet szélessége 2-bit.

Azon kívül, hogy a MUX elsődleges feladata az adatbemenetek közötti választás, **keresőtáblaként (lookup table)** is használható. Általánosan az mondható el, hogy egy 2^N bemenettel rendelkező MUX N darab logikai függvény implementálását teszi lehetővé az által, hogy a MUX bemeneteire fixen logikai 0 vagy 1 értékeket adunk. Ezt egy példával fogom szemléltetni:

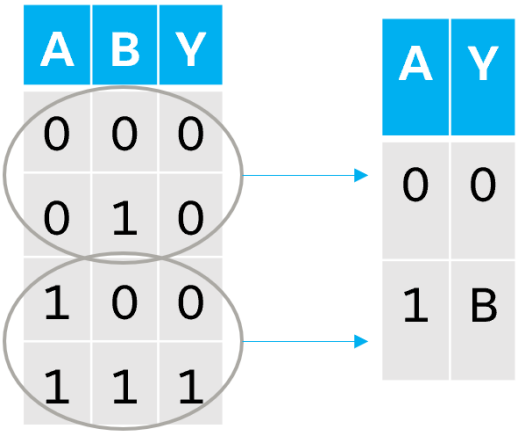
13.1 Példa: Implementáld a lenti igazságtábla alapján működő áramkört 4:1-es MUX felhasználásával.

A	**B**	**Y**
0	0	0
0	1	1
1	0	0
1	1	0



Először is figyeld meg, hogy a fenti példában az igazságtábla változóit a MUX kiválasztó bemenetére kötöttük. Mivel az igazságtáblában meg van adva minden bemeneti kombinációhoz a kimeneti érték, eztért csak annyi a dolgunk, hogy egy adott bemeneti kombinációval kiválasztott MUX bemenetet 0-ra vagy 1-re kell állítani az igazságtábla kimeneti értéke alapján.

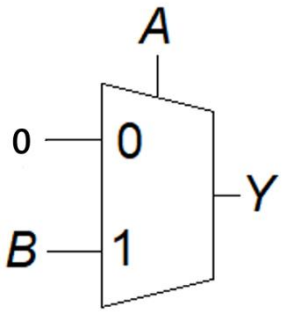
Egy egyszerű ötlettel, a korábban használt multiplexer méretét a felére csökkenthetjük és így elég egy 2^{N-1} bemenetű MUX az N változós függvény implementálásához. Az ötlet az, hogy egy bemeneti változót távolítsunk el az eredeti igazságtáblából és ha kell, használjuk fel a kimenet beállításához az új igazságtáblában. Ezt az ötletet szemlélteti a 21. Ábra.



21. Ábra. Az igazságtábla méretének csökkentése.

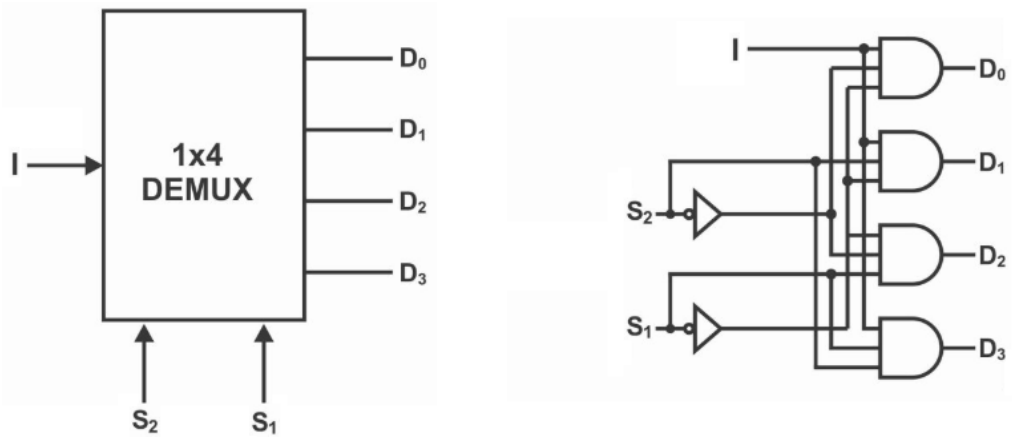
A fenti ábrán az látható, hogy a jobb oldali bemeneti változót (B) elhagyjuk és csak egy (A) változó marad a redukált igazságtáblában. Nézzük meg, hogy mi a kimenet az eredeti igazságtáblában, ha A = 0. Azt látjuk, hogy ekkor Y = 0 függetlenül B értékétől. Most nézzük meg, hogy mi lesz a kimenet, ha A = 1. Ebben az esetben Y = B. Ezekre a megfigyelésekre támaszkodva töltöttük fel a redukált igazságtábla kimeneti oszlopának értékeit. Vedd észre, hogy a redukált igazságtáblában leírt működési elv már egy 2:1-es MUX-al is megvalósítható!

13.2 Példa: Implementáld egy 2:1 MUX-al a 21. Ábrán látható redukált igazságtába működési elvét.



13.5 Demultiplexer

A **demultiplexer (DEMUX)** a multiplexer működésének a fordítottját végzi el. A feladata, hogy digitális adatokat kapcsoljon egy bemeneti vonalról több kimeneti vonalra. Működéséből adódóan, adat-elosztónak is szokták nevezni. Hasonlóan a multiplexerhez, a demultiplexer is rendelkezik **kiválasztó (select)** bemenetekkel és az itt beállított kód határozza meg, hogy a bemenő digitális érték melyik kimeneten legyen továbbítva. Példaként egy 1:4-es DEMUX szimbóluma és áramköri rajza látható a 22. Ábrán. Az ábrán figyeld meg, hogy a demultiplexer megvalósítása AND kapukon alapul, ahol minden AND kapu egyik bemenete a DEMUX adatbemenete, míg a többi a select vezérlő bemenet bitjei invertált vagy nem invertált formában. Nem meglepő módon, a gyakorlatban olyan IC-kel is találkozhatunk, amelyeket demultiplexálásra használhatóak. Egy ilyen például a **74LS138**.



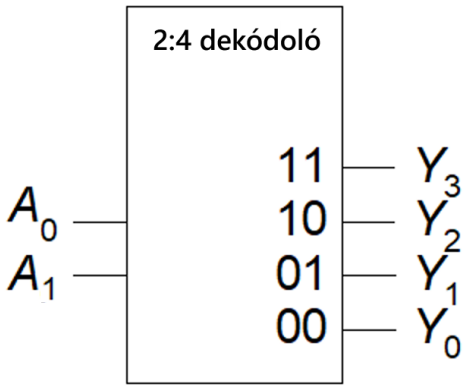
22. Ábra. 1:4 DEMUX szimbóluma (baloldalt) és áramköri rajza.

13.6 Dekódoló

A dekódolók szintén széles körben alkalmazott kombinációs logikai áramkörök. Feladata a bemenetére érkező bitkombináció (kód) detektálása és ennek jelzése egy előre meghatározott kimenetén. A dekódoló N bemenettel rendelkeznek és 2^N kimenettel. Általában a kimenetek közül csak egy kimenet az aktív, amit a bemeneten megadott kód határoz meg. A dekódolók kimenetére **one-hot** kifejezéssel is szoktak utalni, ha az összes kimenet közül mindig csak egy kimeneten lesz logikai 1-es érték (hot), míg a többi 0. Ebből könnyű kitalálni, hogy a **one-cold** kifejezést pedig abban az esetben használjuk, ha a dekódoló aktív kimenetét 0-s érték jelöli, míg a többi kimenet értke 1-es. A számítógépes

rendszerekben, a dekódolók végzik a memóriák címzését. Képzeld el azt, hogy adatot akarunk kiolvasni a memóriából. Ekkor meg kell mondani, hogy a memória melyik „rekeszéből” szeretnénk kiolvasni az információt. Egy „rekeszre” az egyedi címével tudunk hivatkozni és ezt kapja meg a dekódoló. Ennek alapján a dekódoló kiválasztja a megfelelő rekeszt és annak a tartalma lesz kiolvasva.

Egy 2:4-es dekódoló szimbólumát és annak igazság tábláját mutatja be a 23. Ábra.

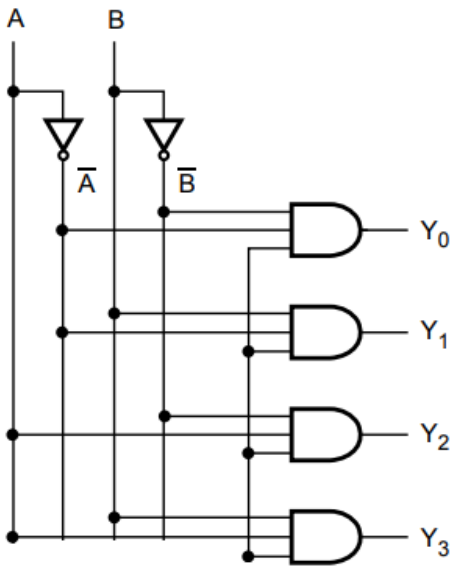


A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

23. Ábra. 2:4-es dekódoló és annak igazságtáblázata.

Mivel a dekódoló is kombinációs logikai áramkör, így egy igazságtáblával egyértelműen le tudjuk írni a működését. Habár itt is alkalmazzuk az absztrakciót a logikai kapukból felépülő áramkör elfedésére, de azt meg kell jegyezni, hogy egy $N{:}2^N$ dekódoló megvalósításához 2^N darab N bemenetű AND kapura van szükség plusz inverterekre a változók negáltjainak előállításához. Szemléltetésként, a 2:4-es dekódoló kapu szintű felépítése a 24. Ábrán látható.

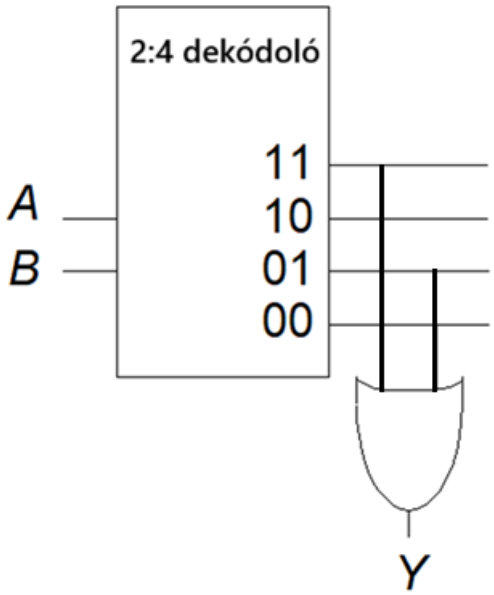
A dekódolókról azt is érdemes tudni, hogy OR kapukkal kiegészítve ezek az áramköri komponensek is lehetőséget biztosítanak logikai egyenletek implementálására. Mivel a dekódoló AND kapui mind egy mintermet állítanak elő, így az SOP formátumú egyenletek megvalósítása kézenfekvő. Általánosan ez úgy fogalmazható meg, hogy egy N változós függvény, amely M mintermből áll egy $N{:}2^N$ dekódolóval és egy M bemenetű OR kapuval valósítható meg. Nézzük meg egy példán keresztül, hogy ez a gyakorlatban hogyan működik.



24. Ábra. 2:4-es dekódoló kapu szintű felépítése.

13.3 Példa: Implementáld a lenti egyenletet dekódoló felhasználásával.

$$Y = AB + \bar{A}B$$



Számos különböző dekódolót tudunk tervezni, attól függően, hogy milyen problémát kell megoldani vele. Ennek a fejezetnek a lezárásaként, néhány népszerű dekódoló típust tekintünk át. Ezek közül az első a 4-bites **bináris-decimális dekódoló**, ami egy 4-bites kódot vár a bemenetén és ennek alapján aktiválja a 16 kimenete közül valamelyiket. Vedd észre az összefüggést a bemeneti kód szélessége és a kimenetek száma között! Mivel 4 biten 16 különböző kódot lehet tárolni, innen ered, hogy ez a dekódoló 16 kimenettel

rendelkezik. Integrált áramköri formában a **74HC154** használható bináris-decimális dekódolóként.

Egy másik népszerű dekódoló típus a **BCD-decimális**. Ahogy a név is jelzi, itt a bemenet BCD kód, ami szintén 4-bites, mint a 4-bites bináris-decimális dekódolónál, de ennél a dekódoló típusnál csak 10 kimenet van. Azért csak 10 kimenettel rendelkezik, mert csak 10 érvényes BCD kód van. Nem túl bonyolult kitalálni, hogy az a kimenet lesz az aktív, amelyiknek a BCD kódja megjelent az áramkör bemenetén. BCD-decimális kódolóként a **74HC42**-es IC-t lehet használni az áramkörökben.

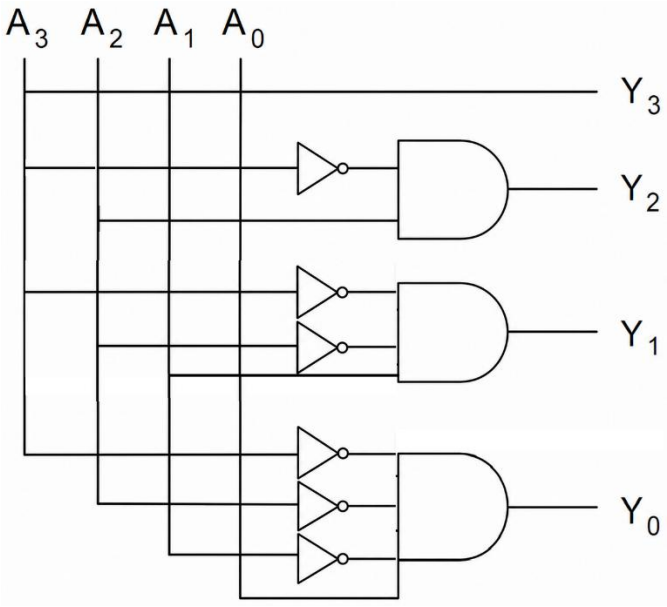
A következő gyakran használt dekódoló típus a **BCD-hétszegmenses dekódoló**. A dekódoló nevéből könnyen ki lehet találni, hogy a bemenetén ez is egy BCD kódot vár. Viszont, ellentétben a legtöbb dekódolóval, ez a dekódoló, a 7 kimenete közül, többet is aktív állapotba helyez. Erre azért van szükség, mert a dekódoló kimenete azt jelzi, hogy egy 7-szegmenses kijelző (következő fejezet) mely szegmenseit kell bekapcsolni, ahhoz, hogy a kíván számjegy jelenjen meg a kijelzőn. Egyszerűen megfogalmazva, ennek a dekódolónak az a feladata, hogy a BCD kóddal reprezentált tízes számrendszerbeli számjegy megjelenítéséhez szükséges szegmenseket bekapcsolja.

Végül az utolsó dekódoló típus, amiről szeretnék szót ejteni az a **prioritásos dekódoló**. Ez is kissé eltér az „általános” értelemben vett dekódolóktól azért, mert a legtöbb esetben itt a kimenetek száma megegyezik a bemenet szélességével. Azt, hogy melyik kimenet lesz az aktív, a bemeneti kódban bitjeinek prioritási sorrendje dönti el. Ennek szemléltetésére nézzünk egy példát.

13.4 Példa: Egy professzor, egy docens, egy adjunktus és egy tanársegéd időről időre ugyan azt az előadótermet szeretné használni. Az ütközések elkerülése érdekében tervezz meg egy prioritásos teremfoglalási rendszert, ahol a foglalási prioritás megegyezik az oktatók beosztásával. Határozd meg a prioritásos dekódoló igazságtáblázatát és ennek alapján rajzold fel az áramkört, amely ezt a funkciót végrehajtja.

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0

0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

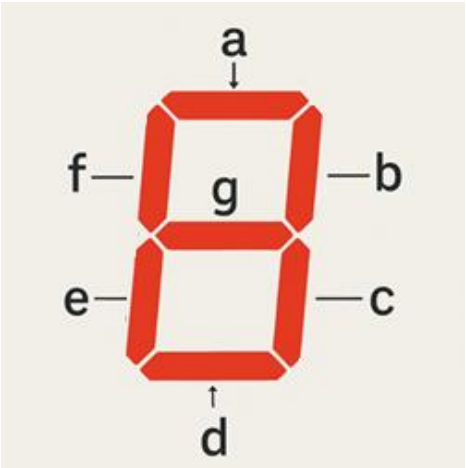


A fenti példa igazságtáblájában A_3 jelenti a professzort és lefelé haladva A_0 a tanársegédet. A kimeneteknél Y_3 jelöli azt az esetet, amikor a professzor kapja meg a termet és lefelé haladva Y_0 jelöli azt, ha a tanársegéd. Az igazságtábla már egyszerűsített alakban van, ahol néhány cella X-et tartalmaz. Ez azt jelenti, hogy abban a sorban az adott bemeneti bit mindegy milyen értékű, mert nincs befolyása a kimenetre. Vedd észre, hogy a kimenetet a vezető 1-es helye határozza meg (beosztás). Az ezt követő bitpozíciókon elhelyezkedő értékek már nem szólnak bele a kimenet alakulásába. Ennek alapján az mondható, hogy a bitek prioritása az MSB-től az LSB bit felé csökken.

13.6 7-szegmenses kijelző

A hétszegmenses kijelzővel már számos helyen találkozhattál, mivel rendkívül elterjedt áramköri komponens a beágyazott rendszerek világában (25. Ábra). A nevéből is kitalálhattad, hogy a kijelző 7 darab szegmensekből áll, ami kiegészül egy programozható ponttal, amit a kijelző jobb alsó sarkába helyeztek el. A pontra tipikusan akkor van szükség, ha törtrésszel rendelkező valós (lebegőpontos) számokat szeretnénk megjeleníteni.

Ahogy az a 25. Ábrán is látható, minden egyes szegmensnek megvan a saját azonosítója, ami az angol abc első néhány karaktere (**a, b, c, d, e** és **f**). Ezt a legegyszerűbben úgy lehet memorizálni, hogy a szegmensek neve fentről indulva óramutató járásával megegyező irányba követik az angol abc karaktereit a középső szegmenssel bezáróan.



25. Ábra. 7-szegmenses kijelző.

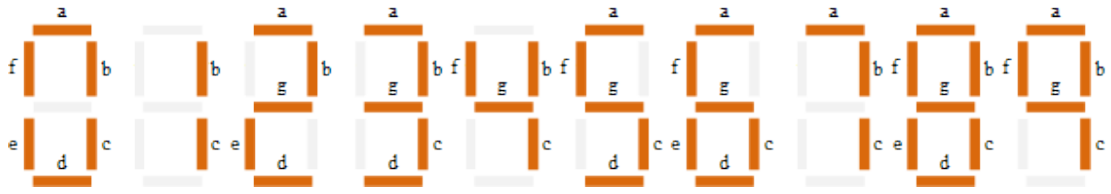
Minden egyes szegmens mögött egy LED (Light Emittind Diode) található. Mivel a LED-ről, már az elektronika tantárgynál volt szó, így azt már mindenki tudja, hogy a LED polaritásos áramköri komponens. Tehát van egy anód (+) és egy katód (-) vezetéke. Ebből kifolyólag a 7-szegmenses kijelzőknek két változata érhető el a gyakorlatba: a közös anódos és a közös katódos. A közös anódosnál minden szegmens LED-jének az anód vezetéke össze van kötve, tehát a 7-szegmenses kijelző egyetlen közös anód lábbal rendelkezik. A közös katódosnál ez meg van fordítva és a LED-ek katódjai vannak egyetlen lábhoz bekötve.

A legtöbb alkalmazásban, a 7-szegmenses kijelzőt, a tízes számrendszer számjegyeinek a megjelenítésére használják. Mindemellett karaktereket is meg tudunk jeleníteni, mint például az A, b, C, d, E és F. Ezzel lehetőségünk nyílik egy hexadecimális szám számjegyeinek a megjelenítésére is.

A kombinációs logikai áramkörök területén megszerzett ismereteink összekapcsolásaként, egy 7-szegmenses kijelzőhöz foguk tervezni dekódoló áramkört. Az áramkör megtervezésénél fel fogjuk használni a Karnaugh táblát, az egyes szegmensek működtetéséért felelős logikai egyenletek felírásához.

13.5 Példa: Készíts egy dekódoló áramkört a 7-szegmenses kijelzőhöz, amely a bemenetén BCD kódot kap ($D_{3:0}$) és ennek alapján vezérli a szegmensekben elhelyezett LED-ek állapotát, hogy megjelenítse a számjegyeket 0-tól 9-ig. A lenti igazságtábla alapján készíts Karnaugh táblát és ennek segítségével határozd meg a szegmensek (a-g) egyenleteit. Azt feltételezzük, hogy a kijelző közös katódos és a használaton kívüli értékek (10–15) nem kapcsolják be a kijelző szegmenseit. A számjegyek mintáinak megjelenítésében segítséget nyújt a 26. Ábra.

$D_{3:0}$	a	b	c	d	e	f	g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
többi	0	0	0	0	0	0	0



26. Ábra. Számjegyek megjelenítése a 7-szegmenses kijelzőn.

A fenti példában egyes szegmenseknél több egyforma hosszúságú és helyes egyenlet létezik! Ezek közül itt csak az egyiket adom meg. Ennek fényében az szegmensek egyenletei a következők:

$$a = \overline{D_3}D_1 + \overline{D_3}D_2D_0 + D_3\overline{D_2}\,\overline{D_1} + \overline{D_2}\,\overline{D_1}\,\overline{D_0}$$

$$b = \overline{D_3}\,\overline{D_2} + \overline{D_2}\,\overline{D_1} + \overline{D_3}\,D_1D_0 + \overline{D_3}\,\overline{D_1}\,\overline{D_0}$$

$$c = \overline{D_2}\,\overline{D_1} + \overline{D_3}\,D_2 + \overline{D_3}\,D_0$$

$$d = \overline{D_2}\,\overline{D_1}\,\overline{D_0} + \overline{D_3}\,\overline{D_2}D_1 + \overline{D_3}\,D_1\overline{D_0} + \overline{D_3}\,D_2\overline{D_1}D_0$$

$$e = \overline{D_3}\,\overline{D_1}\,\overline{D_0} + \overline{D_3}\,D_2\overline{D_0}$$

$$f = \overline{D_2}\,\overline{D_1}\,\overline{D_0} + \overline{D_3}\,D_2\overline{D_1} + D_3\overline{D_2}\,\overline{D_1} + \overline{D_3}\,D_2\overline{D_0}$$

$$g = \overline{D_2}\,D_2\,\overline{D_1} + D_3\overline{D_2}\,\overline{D_1} + \overline{D_3}\,\overline{D_2}D_1 + \overline{D_3}\,D_1\overline{D_0}$$

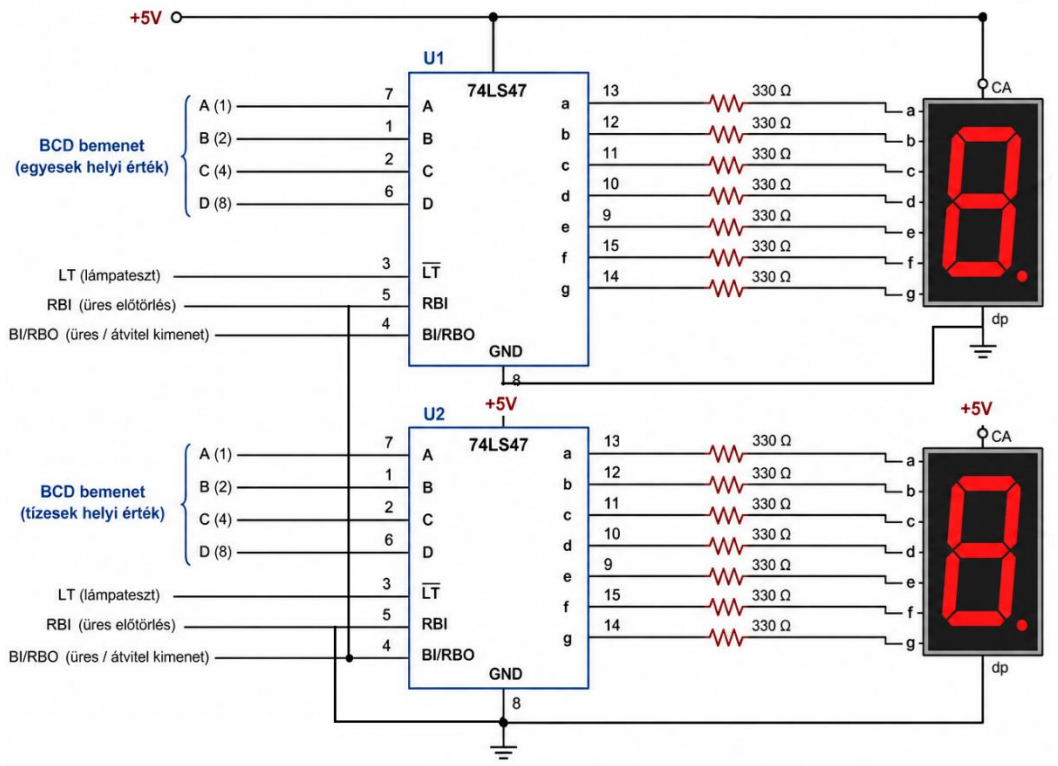
A 7-szegmenses kijelzők bemutatásakor, fontos kiemelni a kijelzők vezérlésére szolgáló BCD-hétszegmenses dekódolót, amiről az előző fejezetben már szó esett. A gyakorlatban ezt a feladatot tölti be a **74LS47**-es IC. Ez az IC, 0-ás értékkel jelöli a kiválasztott szegmenseket (**negatív logikát használ**). Ebből következik, hogy közös anódos 7-szegmenses kijelzőkhöz ajánlott. A dekódoló a BCD bemeneteken túl, még további három vezérlő/tesztelő vezetékkel rendelkezik. Az LT (lámpa teszt) a kijelző összes szegmensének tesztelését teszi lehetővé, míg az RBI és a BI/RBO vezetékek a vezető nullák elhagyásában játszanak szerepet. Ahhoz, hogy a vezető nullák ne jelenjenek meg, az egyik 74LS47 dekódoló BI/RBO kimenetét a következő dekóder RBI bemenetéhez kell csatlakoztatni. Ez rávilágít arra is, hogy a felhasználóknak lehetősége van több 7-szegmenses kijelző egyidejű vezérlésére, a megfelelő számú 74LS47 IC összekapcsolásával, ahogy az a 27. Ábrán is látható. Az ábrán feltüntettem az áramkorlátozó ellenállásokat is, amikkel a szegmensek mögött álló LED-eken átfolyó áram mennyiségét korlátozhatjuk. Ezeken felül, még további ellenállásokra is szükség lehet. Ezzel kapcsolatban további részleteket, az általad használt BCD-hétszegmenses dekódoló adatlapjában találsz.

13.7 Kódoló

A kódoló áramkör a dekódolóhoz képes fordítottan működik. Általában a bemenetei közül egy bemenet az aktív és ez szabja meg, hogy az áramkör milyen kimeneti kódot generál. Sok esetben a kódoló bemenetei számokat vagy szimbólumokat reprezentálnak és a kimeneten ennek a bináris kódja jelenik meg. Példaként vegyük a **decimális-BCD kódolót**, amelynek a bemenetei a decimális számokat reprezentálják, míg a kimenete a szám BCD kódja. Ennek a kódolónak az áramköri rajza látható a 28. Ábrán. Figyeld meg,

hogy az ábrán nincs dedikáltan 0-s bemenet. Ez egyszerűen azért van, mert, ha egyetlen bemenete sincs kiválasztva a kódolónak, akkor a kimenetén a 0000-ás kódot adja vissza, ami a decimális nulla BCD kódja. A korábban tanultak alapján próbáld meghatározni a kódoló kimeneteit leíró logikai egyenleteket.

Ha a saját áramkörödben szeretnél decimális-BCD kódolót használni, akkor erre a feladatra a **74HC147**-es IC-t használhatod. Ez tíz aktív alacsony bemenettel rendelkezik (negatív logika) és a BCD kódok komplemensét adja vissza a kimeneten. Ez az eszköz további rugalmasságot kínál, mivel **prioritáskódolóként** is funkcionál. Ez azt jelenti, hogy ha egynél több bemenet aktív (0-ás), akkor a legmagasabb rendű decimális számjeggyel rendelkező lesz a meghatározó, amelytől a kimeneti érték függ.

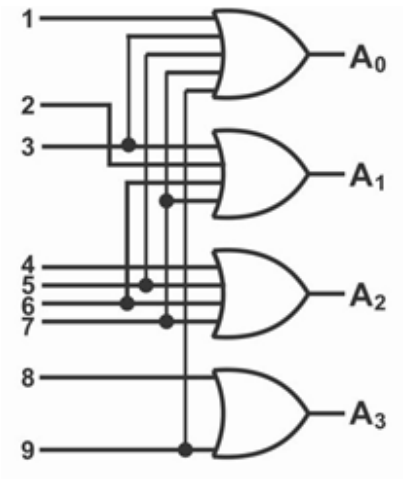


27. Ábra. 74LS47-es IC-k összekapcsolása.

Egy másik gyakori dekódoló típus a 8-3-as, aminek 8 bemenete van és 3-bites kódot generál a kimenetén. A bemenetek és a kimenetek szélessége itt sem véletlenül alakult így. Ne feledd el, hogy a 8 bement (állapot) 3-bittel kódolható le.

Későbbi tantárgyak során tanulni fogsz a magasszintű programozási nyelven elkészített program lefordításáról gépi-kódra, amit a processzor értelmezni tud. Ez egy többlépcsős folyamat lesz, amelynek az egyik komponense az **assembler**. Az assembler fogja az elemi

utasításokat gépi-kódra fordítani. Az assembler egy szoftveresen megvalósított kódolónak is tekinthető, amely minden elemi utasításhoz egy kódot generál le.



28. Ábra. Decimális-BCD kódoló áramköri rajza.

13.6 Példa: Tegyük fel, hogy a 28. ábrán látható decimális-BCD kódoló 3. bemenete 1-es értékű, míg a többi 0. Határozd meg, hogy mi lesz a kódoló kimenete.

$$A_3 = 0, A_2=0, A_1=1, A_0=1$$

14. Tárolók

Az előző fejezetben láthattad, hogy hogyan lehet létrehozni alapvető kombinációs logikai áramköröket, ahol az áramkör kimenete csak az aktuális bemenetek értékétől függ. Ebben a fejezetben a **szekvenciális** logikai áramkörökkel fogunk megismerkedni, ahol az áramkör kimeneti értéke az aktuális bemeneti értékeken túl, az áramkör korábbi bemeneti értékeitől is függ. A korábbi bemenetektől való függést, az áramkörbe megtalálható memória teszi lehetővé. Ez azért fontos, mert a korábbi bemenetek alapján dől el, hogy egy adott időpillanatban, mi a rendszer aktuális állapota. A rendszer állapotát egy bináris értékként fogjuk leírni, ami a szekvenciális logikai áramkör memóriájában tárolódik. Ennek a fejezetnek az elején azt nézzük meg, hogy hogyan tárolódik az adat a szekvenciális áramkörben.

14.1 SR latch

A memóriák elemi építő kövei úgynevezett **bistabil** memória cellák. Ez egyszerűen azt jelenti, hogy a memória cella két állapotban lehet (0, 1). Azt is mondhatjuk, hogy egy ilyen cella 1-bit információ tárolására képes. Az egyik legegyszerűbb bistabil áramkör az **SR latch**, amely két NOR kapuból vagy két NAND kapuból építhető meg. Mi itt a NOR kapukból álló SR latch-re koncentrálunk, mivel ez pozitív logikát használ a NAND kapuból megépíthető változattal szemben. Az áramkör sajátossága, hogy a NOR kapuk kimenete vissza van csatolva a bemenetre (29. ábra). Ahogy a neve is jelzi, az SR latch két benetettel rendelkezik. Az S a beállító (set) bemenet, míg az R a törlő (reset) bemenet. A két bemenet mellett két kimenete is van: a Q és ennek a negáltja. A Q kimeneten mindig azt látjuk, hogy aktuálisan milyen érték van letárolva a latch-ben. A 29. ábrán az áramkör igazságtáblája is látható. Ennek alapján, ha mindkét bemenet 0 értékű (S=0 és R=0), akkor a latch megőrzi a korábban letárolt értékét és nem történik semmilyen változás. Azt is mondhatjuk, hogy ekkor a latch emlékezni fog a korábbi állapotára – memóriaként funkcionál. Ha az S=0 és R=1, akkor gyakorlatilag töröljük a latch tartalmát (azaz 0-ra állítjuk). Ha S=1 és R=0, akkor a latch tartalmát 1-re állítjuk és ez az érték jelenik meg a Q kimeneten is. Végül, ha mindkét bemenet 1-es értéket kap, akkor a latch mindkét kimenete nullázódik, ami hibás működést jelent. Ebből adódóan az S=1 és R=1 bemeneti

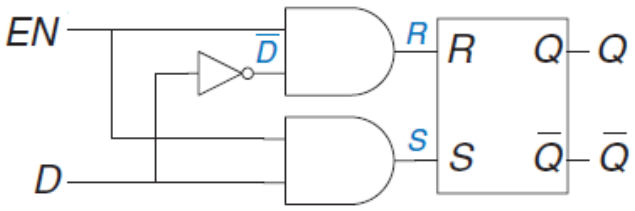
kombinációt érvénytelen bemenetnek tekintjük! Ha belegondolunk, akkor ennek nincs is értelme, mert egyszerre szeretnék 1-es értékűre állítani a latch tartalmát és közben törölni is. Ne felejtsd el, hogy helyes működés esetén a Q negált kimenetnek mindig ellentétes értékűnek kell lennie a Q kimenethez képest.

S	R	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
0	1	0	1
1	0	1	0
1	1	0	0

29. Ábra. SR latch felépítése NOR kapukból (baloldalt) és a működését leíró igazságtábla (jobboldalt).

14.2 D latch

A D latch egy másik 1-bites tároló, amely egy D bemenettel és két kimenettel rendelkezik. A D vezérlő bemenet sok esetben kiegészül egy engedélyező (EN) bemenettel is, amivel engedélyezhetjük a latch-ben lévő aktuális érték frissítését, a D bemeneten keresztül (30. ábra). Az engedélyező bemenettel rendelkező D latch-et kapuzott lech-nek is nevezzük. A D latch előnye, hogy kiküszöböli az SR latch-nél látott érvénytelen bemeneti kombinációt. Ahogy a 30. ábrán is látható, a kapuzott D latch áramköri felépítése gyakorlatilag az SR latch-nek egy kibővített változata. Az SR latch-re itt a latch szimbólumával hivatkozunk a kapuszintű rajz helyett, ami ki lett egészítve két AND kapuval és egy inverterrel. Az ábrán az áramköri felépítés mellett a latch igazságtáblája is látható. Figyeld meg, hogy ha EN=0, akkor teljesen mindegy, hogy hogyan változtatjuk a D bemenetet, mert annak nem lesz hatása a latch-ben letárolt értékre. Ilyenkor a latch megőrzi a korábbi állapotát. Abban az esetben, ha EN=1, a D bemeneten beállított érték lesz letárolva a latch-ben.

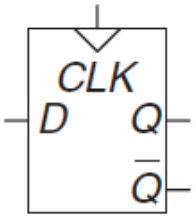


EN	D	Q	$\overline{Q}$
0	X	Q	$\overline{Q}$
1	0	0	1
1	1	1	0

30. Ábra. Kapuzott D latch felépítése (baloldalt) és a működését leíró igazságtábla (jobboldalt).

14.2 D flip-flop

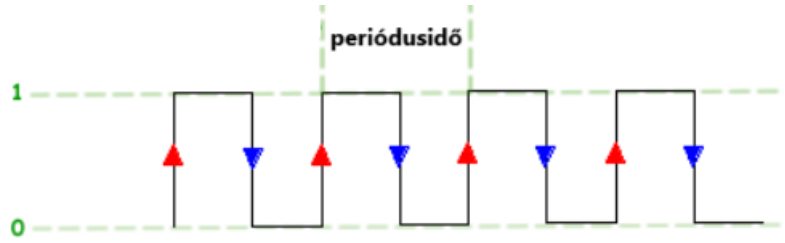
A latch-hez hasonlóan, a flip-flop is egy 1-bites tároló típus, de a működési elve eltér a letch-től. A flip-flop is bistabil, azaz két stabil állapotban lehet (0 vagy 1). Egy adott időpillanatban a két tárolható érték közül csak az egyiket tárolja (vagy a 0-át vagy az 1-et). Azt is meg kell említeni, hogy hasonlóan a latch-hez a flip-flop is egy **„felejtő”** tároló. Ez azt jelenti, hogy ha megszüntetjük a flip-flop tápfeszültségét, akkor a tartalma elvész. Ha egyszerűen csak azt mondjuk, hogy flip-flop, akkor tipikusan a D flip-flopra gondolunk, amelynek a szimbóluma a 31. ábrán látható.



31. Ábra. A D flip-flop szimbóluma.

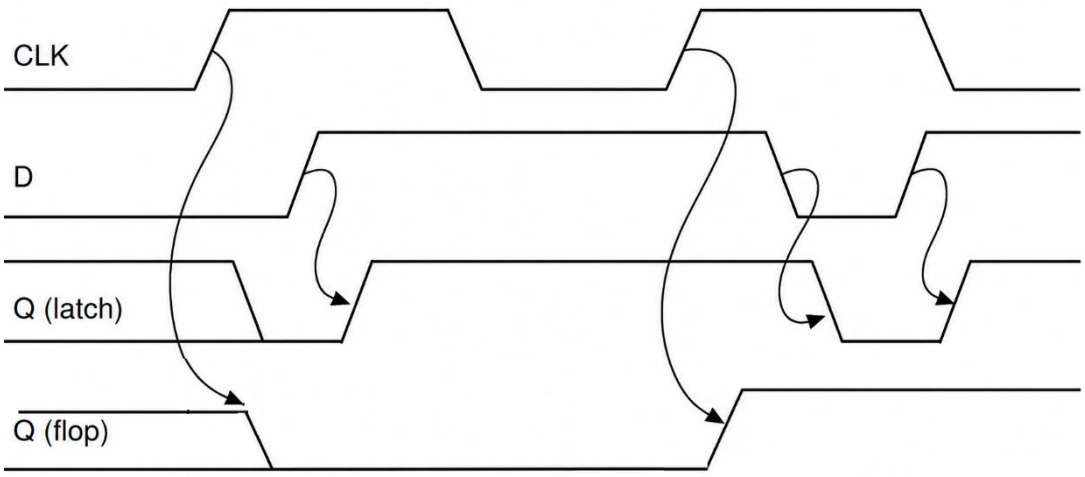
Ami a fő különbséget jelenti a latch és a flip-flop között, az az órajel bemenet, amit a flip-flop-ok szimbólumán egy háromszög jelöl (31. ábra). A flip-flop-ok tartalmának frissítése órajelhez van kötve. Itt felmerülhet a kérdés, hogy mi az órajel? Az órajelet, a legtöbb esetben, **periodikus négyszögjelként** szokták ábrázolni, hasonlóan ahhoz, amit a 32. ábrán látsz. A periodikus jel olyan hullámformát ír le, amely fix időközönként ismétlődik. Tehát ugyan azt a mintát írja le minden egyes periódusban. Figyeld meg, hogy az órajel egy 1-bites jel, amiben a 0 és 1 értékek szabályosan váltakoznak. Azt az időintervallumot, amíg a jel egymást követően 0 és 1-es állapotban is van **periódusidőnek (T)** nevezzük. A jel **frekvenciája (f)** az ismétlődési sebességet írja le, és **Hertzben (Hz)** mérjük. Úgy is megfogalmazhatjuk, hogy a frekvencia azt mutatja meg, hogy a jel hány periódust tesz meg 1 másodperc alatt. Például egy 1 Hz-es jel 1 másodperc alatt egyetlen periódust tesz meg. A periódus idő és a frekvencia közötti összefüggést az f = 1 / T képlettel adhatjuk meg.

Egy perióduson belül az órajelnek lesz egy felfutó és egy lefutó éle. A felfutó él akkor keletkezik, amikor a jel 0-ból 1-re vált, míg a lefutóél az 1-ből 0-ba váltáskor jelenik meg. A 32. ábrán a felfutó éleket a piros, míg a lefutó éleket a kék nyíl jelöli.



32. Ábra. Az órajel szemléltetése.

Az órajel élei kitüntetett fontossággal bírnak a flip-flop-ok esetében, mivel a flip-flop tartalmának frissítése az órajel éleihez van kötve. A flip-flop típusától függően, a tároló tartalmának frissítése az órajel felfutó vagy lefutó élének a bekövetkeztekor frissíthető. Amíg az adott él nem következik be az órajelben, addig a flip-flop megőrzi a korábban letárolt értékét. Ebből adódóan a flip-flop szinkron működésű, mert az állapot változása szinkronban van az órajellel. Ezzel szemben a latch **szint vezérelt**, ami azt jelenti, hogy ha változás történik a latch vezérlő bemenetén, akkor a latch-ben letárolt érték és egyben a latch kimenete („szintje") azonnal követi ezt a változást. Valójában itt is van egy minimális késleltetés, mivel a logikai kapuknak is van egy kicsi (néhány pikoszekundum) késleltetése, de ezzel itt nem foglalkozunk és egyszerűen csak azt mondjuk, hogy a kimenet azonnal követi a bemenet változását. Annak érdekében, hogy jobban megértsük a flip-flop és a latch közötti működésbeli eltérést, tekintsük meg a 33. ábrán látható idődiagramot. Az **idődiagram** digitális hullámformák grafikonja, amely két vagy több hullámforma tényleges időbeli viszonyát mutatja. Továbbá azt is láthatjuk rajta, hogy az egyes hullámformák hogyan változnak a többihez képest.



32. Ábra. A D latch és a D flip-flop időzítési diagrammja.

Az időzítési diagrammon azt feltételezzük, hogy a D bemenet közös, tehát ennek az értéke jelenik meg a flip-flop és a latch D bemenetén is. A CLK jelöli a flip-flop órajelét, ami a latch szempontjából nem releváns. A lenti Q értékek pedig a flip-flop és a latch kimenetét jelölik. Először figyeld meg azt, hogy a CLK első felfutó élére váltott át a flip-flop kimenete meghatározatlanból 0-ra. Ezt követően a D változik meg 0-ról 1-re. Figyeld meg, hogy a latch kimenete „azonnal" követi D változását és 0-ról 1-be vált. Ezzel szemben a flip-flop-ra semmilyen hatással nincs ez a változás és a flip-flop letárolt értéke marad 0, mindaddig, amíg a következő felfutó él be nem következik. Ekkor a D értéke 1-es volt és ennek következtében a flip-flop kimenete is átvált 1-re. A következő esemény, a D bemenetnek 1-ről 0-ba állása. Itt is figyeld meg, hogy a latch kimenete azonnal követi ezt a változást és 1-ről 0-ra vált. Majd ezt követően, amikor a D bemenet ismét visszavált 1-re, a D latch ezt is azonnal követi. Ezzel szemben, a flip-flop-ban letárolt értékre, a D bemenet változása nem volt semmilyen hatással, mivel eközben nem történt felfutó él az órajelben.

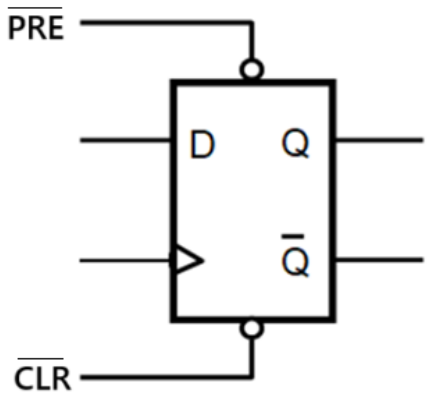
14.2 JK flip-flop

A másik gyakori flip-flop típus a JK flip-flop. A nevéből is adódóan egy J és egy K bemenettel rendelkezik az órajelbemeneten felül, amelyek szinkron bemenetek, mivel ezeken a bemeneteken lévő adatok csak az órajel élének bekövetkeztekor lesznek mintavételezve. A J és K bemenetek elnevezése nem a funkcióikról kapták a nevüket, hanem az áramkör feltalálójáról Jack Kilby-ről. Amikor a J bemenet 1-es és K 0, a Q kimenet 1-es állapotba kerül az órajel élének bekövetkeztekor. Amikor a J=0 és K=1, a Q kimenet alacsony állapotba (0) kerül. Amikor mind a J, mind a K bemenet 0, a kimenet nem változik és a flip-flop megőrzi a korábbi állapotát. Amikor J és K is 1, a flip-flop periodikusan állapotot vált. Ezt nevezik váltó (toggle) üzemmódnak. Ebben az állapotban, a letárolt értékét órajel periódusonként invertálódik. Természetesen ez jelenik meg a flip-flop Q kimenetén is. Amikor a JK flip-flop váltó üzemmódban működik, úgy is szoktak rá hivatkozni, hogy **T flip-flop**.

Természetesen a JK flip-flop-ot is ki lehet bővíteni további bemenetekkel, mint például a törlő (CLR) és a beállító (PR). A 74HC76-os IC két ilyen kibővített JK flip-flop-ot foglal magába, amit gyakran alkalmaznak szekvenciális logikai áramkörök megvalósításához.

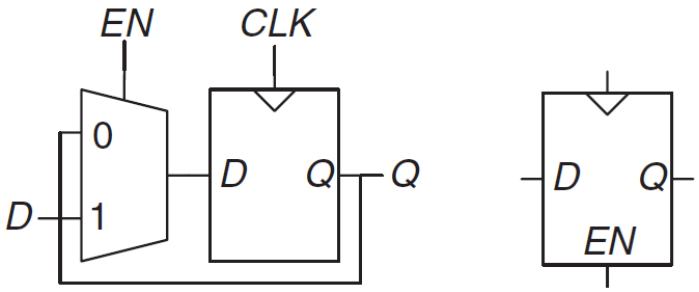
14.3 D flip-flop változatok

Az alap D flip-flop-ot ki lehet egészíteni több elérő bemenettel, így létrehozva a D flip-flop eltérő típusait. Ebben az alfejezetben ezek közül csak egy néhányat tekintünk meg. Mielőtt ezekre rátérünk, fontos kihangsúlyozni, hogy a további bemenetek, amivel kiegészíthetünk egy flip-flop-ot az lehet **szinkron** és **aszinkron** működésű. A szinkron működésű bemenetek a flip-flop órajeléhez vannak szinkronizálva, ami azt jelenti, hogy az órajel élének bekövetkezésekor lesz a szinkron bemenet mintavételezve. Tehát, teljesen mindegy, hogy a szinkron bemenetet hogyan változtatom addig, amíg az órajelnek az éle nem következik be, mert addig nem lesz mintavételezve. Ezzel szemben, az aszinkron bemenetek függetlenek az órajeltől. Ebből adódóan, az aszinkron bemenet változása azonnal hatással van a flip-flop-ban letárolt értékre és nem kell várni az órajel élének bekövetkezésére. Azt is fontos látni, hogy a D flip-flop esetében a D bemenet mindig szinkron működésű, míg a JK flip-flop-nál a J és a K bemenetek is szinkron működésűek mivel ezek az órajel élének bekövetkeztekor lesznek mintavételezve. Az aszinkron bemenetek közül a két gyakori bemenet a beállító (preset - PRE) és a törlő (clear - CLR). Egy aszinkron PRE és CLR bemenetekkel rendelkező flip-flop szinbóluma látható a 33. ábrán. Figyeld meg az ábrán látható „buborékokat” és a felülvonásokat a PRE és a CLR bemeneteknél. Ezek arra utalnak, hogy mindkét bemenet **negatív logikával** működik. Azaz, ha törölni szeretnénk a flip-flop tartalmát a CLR bemeneten keresztül akkor a CLR bemenetet 0-ra kell állítani. Hasonlóan, ha 1-et akarunk tölteni a flip-flop-ba a PRE bemeneten keresztül, akkor a PRE bemenetet 0-s értékkel kell vezérelni.



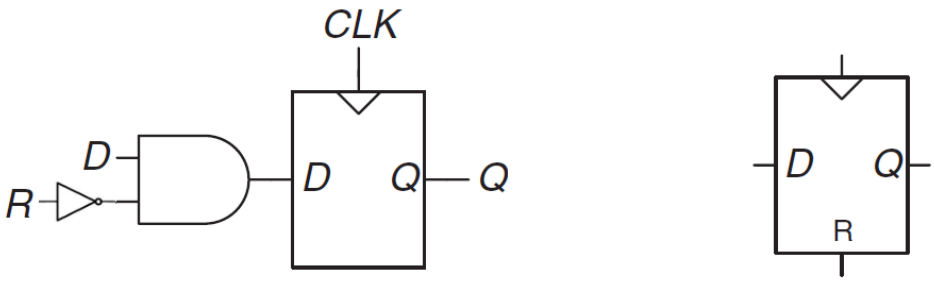
33. Ábra. A D flip-flop aszinkron PRE és CLR bemenetekkel.

Az első módosított D flip-flop-ot egy további engedélyező (enable) bemenettel láttuk el, ami a flip-flop szimbólumában is megjelenik (34. ábra). Az engedélyező bemeneten keresztül lehet beállítani, hogy mikor írhatja felül a D bemeneten megadott érték a flip-flopban korábban letárolt értéket. Ha az engedélyező bemenet (EN) értéke 1, akkor a flip-flop ugyan úgy működik, mint egy sima D flip-flop. Ha viszont az engedélyező bemenetet 0-ra állítjuk, akkor a flip-flop megőrzi a korábban letárolt értékét és a D bemeneten keresztül ezt nem tudjuk módosítani. A 33. ábrán látható, hogy hogyan lehet megvalósítani az engedélyező bemenettel ellátott flip-flop-ot, amit kapuzott flip-flop-nak is neveznek. Egyszerűen az alap D flip-flop-ot kiegészítettük egy 2:1-es multiplexerrel, aminek a kiválasztó bemenetére kötöttük az engedélyező értéket. Ha EN=1, akkor D értékével felül lehet írni a flip-flop aktuális értékét, ha EN=0, akkor visszatöltjük a flip-flop-ba a korábbi értéket, amit úgy is mondhatunk, hogy a tároló megőrzi a korábbi értékét.



33. Ábra. A kapuzott D flip-flop felépítése (baloldalt) és szimbóluma (jobboldalt).

Egy másik változat az alap D flip-flop-ot, törlő bemenettel ruházza fel. A névből könnyű kitalálni, hogy a törlő bemenet feladata a flip-flop tartalmának 0-ba állítása. Gyakorlati szempontból, ez a plusz bemenet rendkívül fontos, mivel bekapcsoláskor a flip-flop kezdeti állapota nem ismert. A törlő bemeneten keresztül, a kezdeti értéket nullába tudjuk billenteni, így nem érhet majd kellemetlen meglepetés. Egy szinkron törlő bemenettel rendelkező flip-flop megvalósítása látható a 34. ábrán. Figyeld meg, hogy az alap D flip-flop ki lett egészítve egy AND kapuval, aminek az egyik bemenete a D vezérlőjel, míg a másik az R (reset) törlő bemenet.

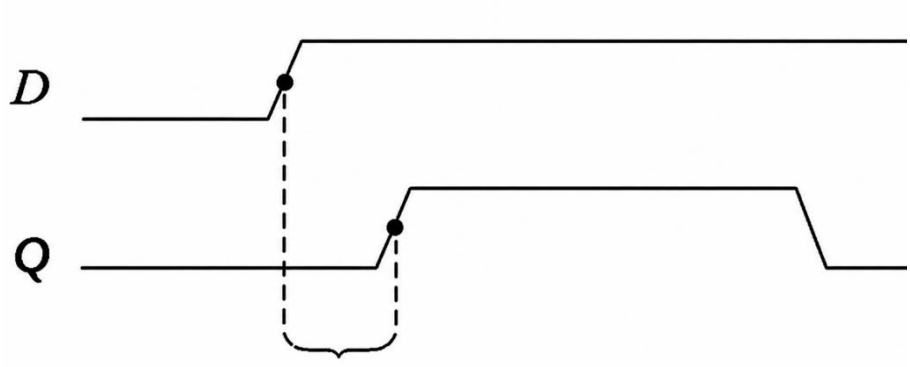


34. Ábra. Törlő bemenettel ellátott D flip-flop.

14.4 Flip-flop karakterisztika

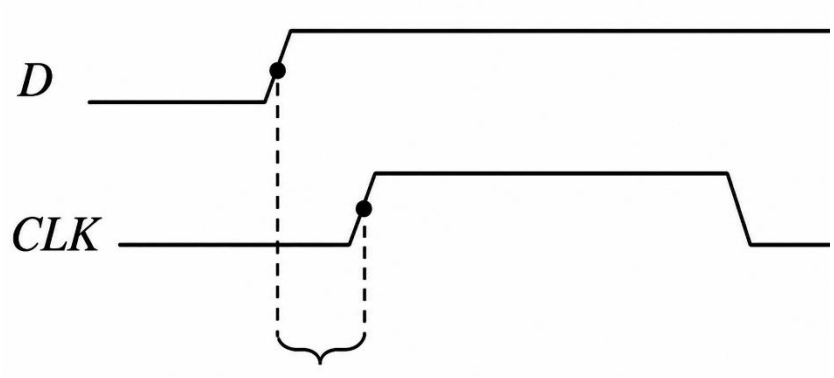
A digitális áramkörök számos fontos jellemzővel (karakterisztika) rendelkeznek és ez igaz a flip-flop-ra is. Minden áramköri komponens karakterisztikájának részletes leírása az áramkör adatlapjában található. Ezek közül csak a legfontosabbakra fogunk koncentrálni ebben az alfejezetben.

- **Terjedési késleltetés:** A terjedési késleltetési azt az időmennyiséget jelenti, amely a bemeneti jel megjelenése után szükséges ahhoz, hogy ennek hatása a kimeneten megjelenjen. Szigorúan véve, ez a bemeneti változás 50%-os pontjától a kimeneti megjelenő változás 50%-os pontjáig eltelt idő (35. ábra). A terjedési késleltetés bemenetenként eltérő lehet, sőt eltérő lehet ugyan annak a bemenetnek a 0-ból 1-be vagy 1-ből 0-ba váltakozásakor is.



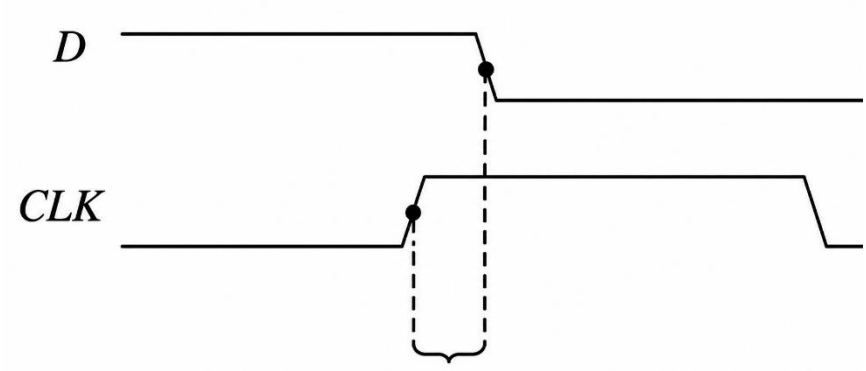
35. Ábra. Terjedési késleltetés.

- **Beállási idő:** A beállási idő az a minimális időmennyiség, amennyivel korábban kell stabilizálódnia egy bemenet (pl.: J, K, vagy D) logikai szintjének az órajel élének bekövetkezése előtt. Ezt az időmennyiséget is a jelek változásának 50%-os pontjai között mérjük (36. ábra).



36. Ábra. Beállási idő.

- **Tartási idő:** A tartási idő az a minimális időtartam, ameddig a bemenet, logikai szintjének még stabilnak kell maradni az órajel élének bekövetkezése után (37. ábra).



37. Ábra. Tartási idő

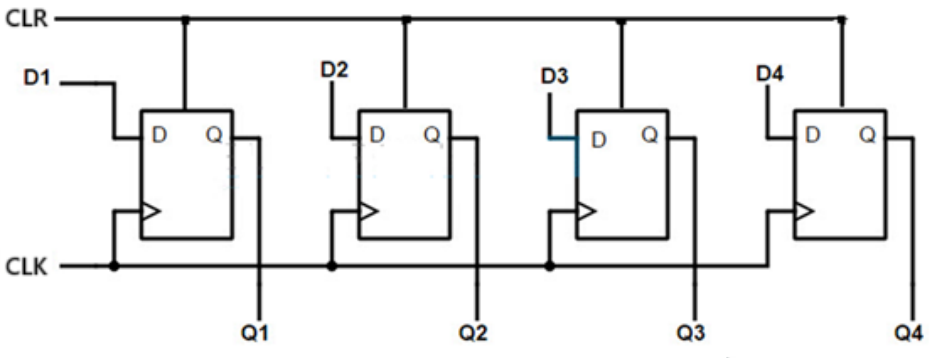
- **Maximális órajel frekvencia:** Az a legmagasabb órajelfrekvencia, amellyel egy flip-flop megbízhatóan működtethető.

- **Teljesítmény:** az áramkör teljes energiafelhasználása, ami leírható a következő egyszerű formulával, ahol P a teljesítmény (wattban), V a feszültség (voltban) és I az áramerősség (amperben): $P = V*I$

15. Flip-flop-ok gyakorlati alkalmazása

15.1 Regiszter

Ahogy korábban láttuk, a flip-flop-ok elsődleges feladata az információ tárolás. 1 flip-flop egyetlen bit információt tud letárolni, ami önmagában igen kevés. Szerencsére a flip-flop-okat sorba kapcsolva N-bites tárolókat lehet létrehozni. A megfelelően sorba kapcsolt flip-flop-ok tömbjét **regiszternek** nevezzük. Egy N-bites regiszter létrehozásához N darab flip-flop-ot kell felhasználni és minden flip-flop-hoz ugyan azt az órajelet kell bekötni úgy, ahogy az a 38. ábrán is látható. Ezzel azt érjük el, hogy a sorban minden egyes flip-flop-nak egyszerre lehet frissíteni a tartalmát. A regiszter egy fontos alkotóeleme a szekvenciális logikai áramköröknek, mivel az tölti be az áramkör memóriájának szerepét.



38. Ábra. 4-bites regiszter.

15.2 Aszinkron számláló

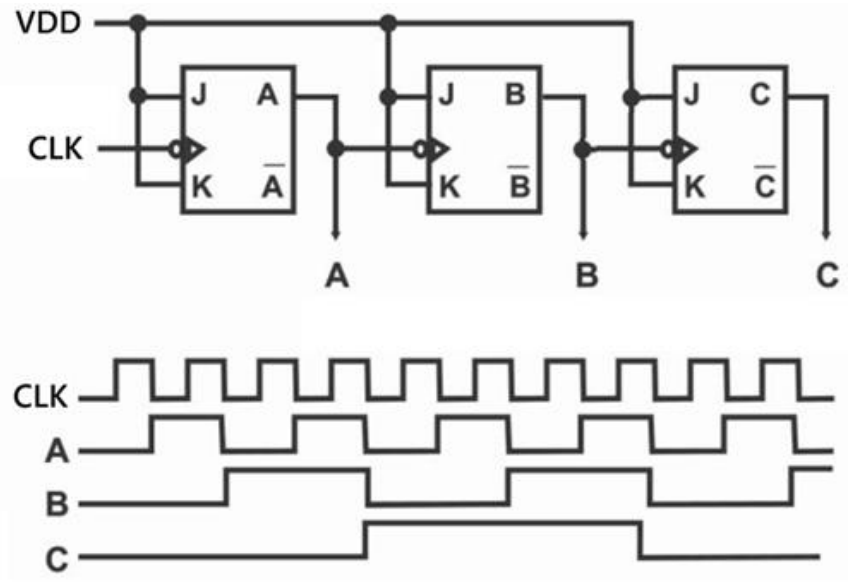
Az adattároláson túl, a flip-flop-okat számlálókként is lehet használni, amennyiben helyesen kötjük láncba őket. A számlálókat **szinkron** és **aszinkron** kategóriákba tudjuk sorolni a flip-flop-ok láncba kötésének alapján. Ebben az alfejezetben az aszinkron számlálókkal fogunk foglalkozni, amelyeket terjedő (**ripple**) számlálóknak is szoktak nevezni. Az aszinkron elnevezés onnan ered, hogy a számláló láncban helyet foglaló flip-flop-ok nem ugyan azt az órajelt használják, így nincsenek is szinkronban. Az aszinkron számlálók tervezésének bemutatásához tekintsd meg a 39. ábrát, ahol egy 3-bites számláló áramköri rajza látható. Általánosan itt is elmondható, hogy egy N-bites számláló létrehozásához N darab flip-flop-ot kell összekapcsolni. Figyeld meg, hogy a flip-flop-ok

lefutó órajelre frissülnek (triggereltek), amit a flip-flop szimbólumán is látható buborék jelez. Ezen felül azt is látni kell, hogy az első flip-flop-nak külső órajele van, míg a láncban elhelyezkedő többi flip-flop órajelét az őt megelőző flip-flop kimenete adja. A láncban elhelyezett minden flip-flop billegő (váltó) üzemmódban működik. Ezt a JK flip-flop-ok esetén úgy érjük el, hogy a J és K bemenetekre logikai 1-es értéket adunk (bekötjük a tápfeszültségre).

A láncban balról indulva a flip-flopok adják a számláló bitjeit az LSB-től az MSB felé haladva. A 39. ábrán egy 3-bites számlálót látunk, aminek 3-bites kimenete lesz. A C változó jelöli a kimenet 2. bitpozícióját, a B az 1. bitpozíciót, míg az A a 0.-at (CBA). Ahhoz, hogy megértsük a számláló működését vegyük szemügyre az ábrán látható időzítési diagrammot. Itt először azt figyeld meg, hogy a flip-flopok a CLK lefutó élére váltanak állapotot. Ez első órajel periódusban a három flip-flop-ban tárolt érték 000. A következőben 001, majd 010. Ha végig haladunk az időzítési diagrammon, akkor azt láthatjuk, hogy a számláló 000-ból indult, majd egyesével növekedett a bináris értéke egészen 111-ig. Végül, a flip-flop-ok értékei visszaálltak 000-ba és a számlálási folyamat indul az elejétől. Ez nem meglepő mivel 3-biten a rendelkezésre álló tartományunk binárisan: [000, 001, 010, 011, 100, 101, 110, 111].

A 39. ábra időzítési diagrammján is jól látható, hogy a számlálóval gyakorlatilag megszámoljuk az első flip-flop-hoz beérkező órajelciklusok számát. Ebből az is következik, hogy az órajel frekvenciájával lehet szabályozni a számláló sebességét.

Az aszinkron számlálóknak van egy lényeges hátránya. Minél több flip-flop-ot kapcsolunk láncba annál nagyobb lesz a lánc legvégén található (MSB) flip-flop késleltetési ideje az első flip-flop órajeléhez képest. Tehát a láncban lévő flip-flop-ok késleltetése összeadódik! Ahhoz, hogy a számláló működőképes legyen, az összegzett késleltetésnek kisebbnek kell lennie, mint az órajel periódusideje! Ebből a hátrányból az is következik, hogy azokban az alkalmazásokban, ahol a számlálót nagy frekvenciájú órajellel kell léptetni, ott szinkron számlálót érdemes használni.



39. Ábra. 3-bites aszinkron számláló.

15.1 Példa: Tegyük fel, hogy egy 4-bites aszinkron számlálót szeretnék megtervezni D flip-flop-ok felhasználásával, ahol tudjuk, hogy a flip-flop késleltetési ideje 20 ns. Határozd meg a számláló terjedési késleltetését (t_p) és a maximális órajel frekvenciáját (f_{max}).

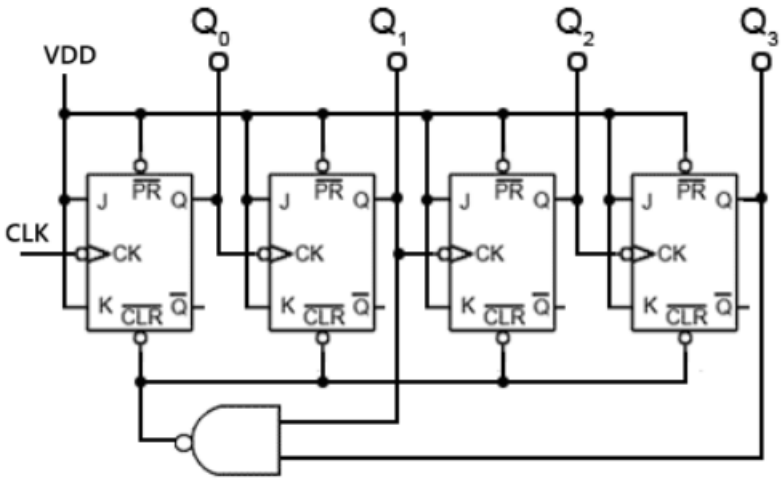
$$t_p = 4 \times 20\ \text{ns} = 80\ \text{ns}$$

$$f_{max} = \frac{1}{t_p} = \frac{1}{80\ \text{ns}} = \frac{1}{0{,}00000008\ \text{s}} = 12500000\ \text{Hz} = 12.5\ \text{MHz}$$

Egy N-bites aszinkron számlálót többféleképpen meg tudunk valósítani. Használhatunk JK, de akár D flip-flop-okat is a számláló láncban. A lényeg az, hogy a flip-flop váltó üzemmódban működjön. Ezen kívül a flip-flop lehet fel vagy lefutó órajelre triggerelt. Ez fontos szempont, mert ettől függően kell összekapcsolnunk egymáshoz a flip-flop-okat. A 39. ábrán a flip-flop-ok lefutó órajelre triggereltek, ezért a nem invertált kimenetet használtuk fel órajelként a következő flip-flop-nál. Ezzel szemben, ha felfutó órajelre triggerelt flip-flop-okból kell megépíteni a számlálót, akkor az invertált kimenetet kell használni a láncban soron következő flip-flop órajeleként.

Korábban már láttad azt, hogy egy N bites aszinkron számláló tartománya [0, 2^N-1] és a számláló ezen a tartományon számol végig. Úgy is tekinthetünk a számlálókra, mint **véges állapotgépek** (erről még később lesz szó), ahol az állapotok számát **modulusnak (MOD)** nevezzük.

A számláló teljes tartományának kihasználása helyett, arra is lehetőségünk van, hogy a számlálási folyamatot egy adott ponton (állapotban) megszakítsuk és nullázzuk a számlálót. Az ilyen számlálókat **csonkított számlálóknak** nevezzük, mivel a teljes tartománynak csak egy részén fog végig lépkedni. Az egyik leggyakoribb csonkított számláló a **tízes (decade) számláló**, amely a tízes számrendszer számjegyein halad végig (0-tól 9-ig binárisan). A tízes számlálót, egy 4-bites számlálóból tudjuk létrehozni, a számláló tartományának „megtörésével" 1001 után. Ezt úgy érjük el, hogy a 4-bites számlálót kiegészítjük egy NAND kapuval úgy, ahogy az a 40. ábrán is látható. A NAND kapu, a számláló 1. és 3. bitpozíciójának értékét kapja meg bemenetként. De miért? Ennek az az oka, hogy az 1001 még egy érvényes állapot és amikor a számláló 1010 (10)-re váltana, akkor kell a számlálót törölni. Tehát a NAND kapu azt vizsgálja, hogy mikor lépne át a számláló 1010-ra, mert ekkor az 1. és a 3. bitpozíció értéke is 1. Ha ez teljesül a NAND kapu 0-s értéket ad a kimenetén (vedd észre, hogy minden más esetben 1-et) és ezzel azt érjük el, hogy a flip-flop-ok CLR (clear) bemenetét aktiváljuk és így töröljük a flip-flop-ok tartalmát. Azaz nullázzuk a számláló értékét.



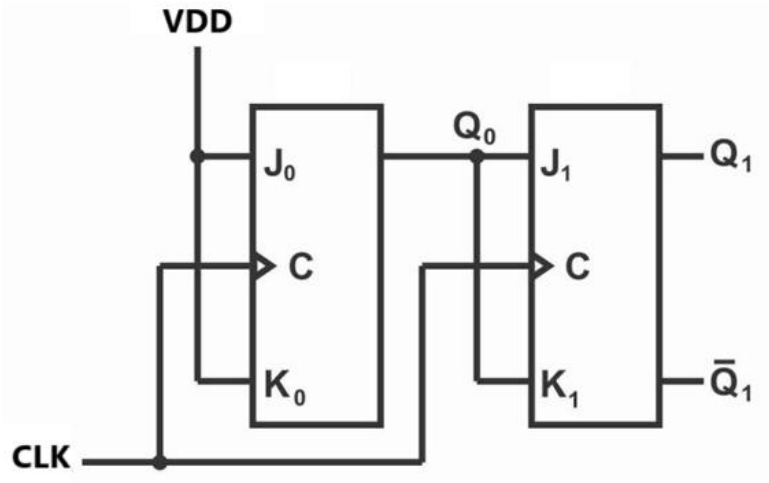
40. Ábra. Tízes számláló.

A 40. ábrán azt is figyeld meg, hogy a CLR bemenet negatív logikával működik! Fontos azt is látni, hogy valójában a számláló át is fordul 1010-ba, de csak egy nagyon rövid időre, ameddig a flip-flop-ok törlődnek. Végül, az ábrán minden flip-flop rendelkezik PR bemenettel is, ami fixen 1-es értékű. A PR bemenetek is negatív logikával működnek, így ezeknek nincs befolyása a számlálóra.

Ebben a fejezetben csak egyetlen csonkított számlálót néztünk meg, de látni kell azt, hogy az itt használt elképzeléssel tetszőleges csonkított számlálót létre tudunk hozni. Ehhez mindössze annyi kell, hogy képesek legyünk detektálni azt az állapotot, amikor nullázni (törölni) kell a számlálót és olyan filp-flop-okat kell használni a számlálóban, amelyek rendelkeznek törlő bemenettel.

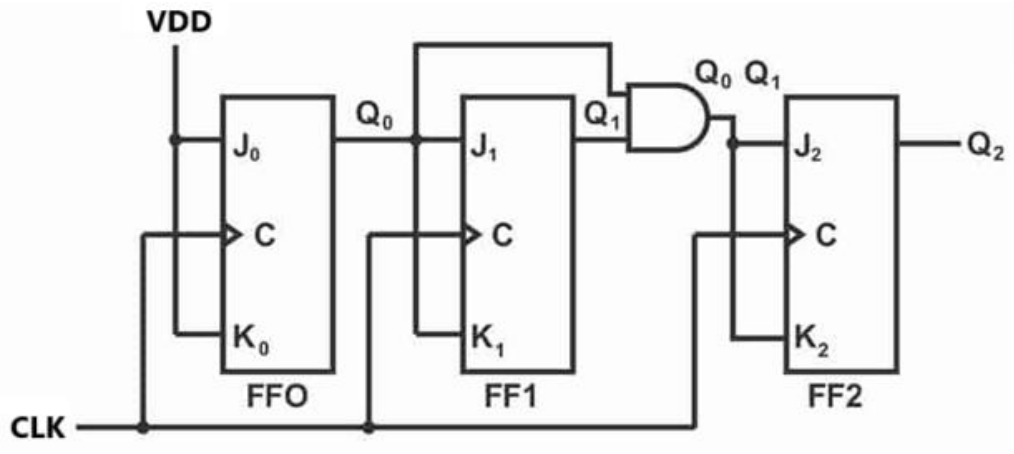
15.3 Szinkron számlálók

Az aszinkron szálálók mellett **szinkron** számlálókat is tudunk tervezni. Az aszinkron számlálókról bemutatásakor már láttad, hogy a számláló láncban az első flip-flop külső órajellel rendelkezik, míg a többi az előző flip-flop kimenetét kapja meg órajelként. A szinkron számlálókban minden flip-flop ugyan azt az órajelet használja, így azok szinkronban vannak egymással. Első példaként tekintsd meg a 41. ábrán látható 2-bites szinkron számlálót, amit JK flip-flop-ok felhasználásával hoztunk létre. Először vedd szemügyre a számláló lánc első flip-flop-ját, ami váltó üzemmódban van, tehát órajelenként invertálódik a benne tárolt érték. A második flip-flop JK bemenete az első flip-flop nem invertált kimenetét kapja meg.



41. Ábra. 2-bites szinkron számláló.

Tegyük fel, hogy kezdetben a flip-flop-ok törölve vannak (00). A következő órajelnél az első flip-flop 1-re vált, de az áramkör késleltetése miatt, a 2. flip-flop J és K bemenetén még 0-s érték detektálható az órajel felfutó élének bekövetkezésekor. Így a 2. flip-flop még 0-t tartalmaz (01). A következő felfutó élnél a 2. flip-flop J és K bemenetén még 1-es érték detektálható, így a 2. flip-flop 1-re vált át, de közben az első flip-flop ismét invertálja az értékét, ami most 0 lesz (10). Innentől a számláló ugyan úgy működik tovább, mint ahogy már a korábbi állapotokban láttuk. Az 1. flip-flop ismét invertálódik és most 1-es értékű lesz, míg a 2. flip-flop marad 1-es mivel az előző állapotban az 1. flip-flop kimenete 0-volt (11). Most nézzük meg, hogy hogyan hozható létre egy 3-bites szinkron számláló, amelynek az áramköri rajza a 42. ábrán látható.



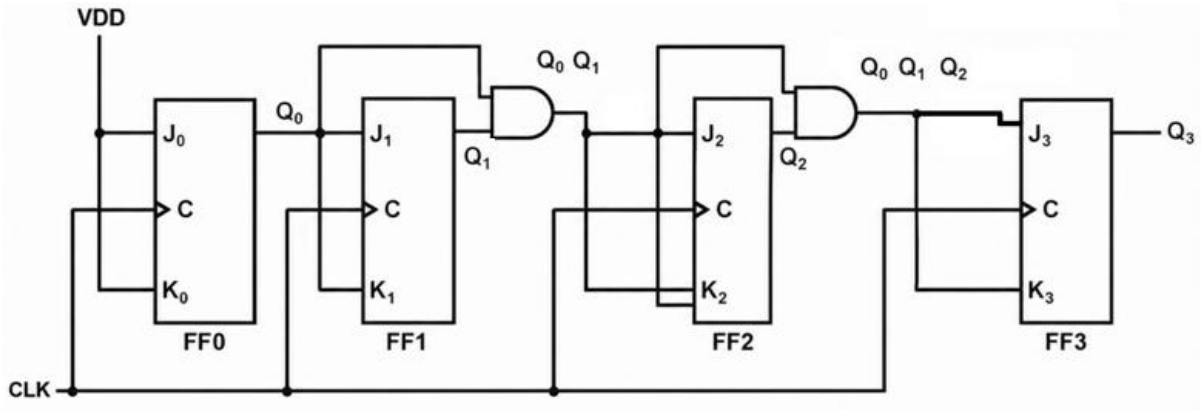
42. Ábra. 3-bites szinkron számláló.

Figyeld meg, hogy a láncban az 1. és a 2 flip-flop pontosan ugyan úgy van összekötve, mint ahogyan azt a 2-bites számlálónál láttuk. Az újdonság a 3. flip-flop bekötésében rejlik, amelynek a J és a K bemeneteit most egy AND kapunak a kimenetével vezérlünk. Az AND kapu bemenetként az első két flip-flop nem invertált kimenetét kapja meg. A kérdés ismét adott: mi ennek az értelme? Ahhoz, hogy az AND kapu funkciója világos legyen vess egy pillantást a 10. táblázatra, ahol a 3-bites számláló tartománya látható. A táblázatban kiemeltem, hogy mely állapotok után kell invertálni a 3 flip-flop értékét. Két ilyen állapot van: 011 és az 111. Mindkét esetben a számláló kimeneti értékének a 0. és az 1. bitje 1 értékű. Ez azt mutatja, hogy akkor kell invertálni a 3. flip-flop-ban letárolt értéket, ha a 0. és az 1. kimeneti bitpozíción 1-es érték van. Ezt a feltételt vizsgálja az AND kapu, ami 1-es értéket ad, ha a feltétel teljesül. Ez biztosítja, hogy a következő órajelnél a 3. flip-flop értéke invertálódjon, ezzel biztosítva a számláló helyes működését.

Kimenet			
Órajel ciklus	**Q_2**	**Q_1**	**Q_0**
0 (kezdő állapot)	0	0	0
1	0	0	1
2	0	1	0
3	**0**	**1**	**1**
4	1	0	0
5	1	0	1
6	1	1	0
7	**1**	**1**	**1**

10. Táblázat. 3-bites számláló tartománya.

Végül nézzük meg, hogy hogyan kell létrehozni egy 4-bites szinkron számlálót, aminek az áramköri rajza a 42. ábrán tekinthető meg.



42. Ábra. 4-bites szinkron számláló.

Ha megvizsgáljuk az ábrát, akkor itt is látható, hogy az első három flip-flop bekötése ugyan úgy történt, mint a 3-bites számlálónál. Az utolsó (MSB) flip-flop-nál ismét azt látjuk, hogy a J és K bemeneteket egy AND kapu vezérli, ami most a Q_2 értéket (FF2 kimenete) és az előző AND kapu kimenetét kapja meg bemenetként. Az első AND kapu kimenete Q_0Q_1. Ebből következik, hogy a második AND kapu kimenete: $Q_0Q_1Q_2$. Hasonlóan, mint ahogy azt a 3-bites szinkron számlálónál láttad, itt is az a kérdés, hogy mikor kell invertálni az utolsó flip-flop-ban tárolt értéket. Ehhez ismét a számláló tartományára kell gondolnunk

(ennek a felírását most rád bízom), amelyből jól látható, hogy az utolsó flip-flop-nak a 0111 és az 1111 állapotok után kell invertálni az értékét. Ez az oka annak, hogy a második AND kapuval azt vizsgáljuk, hogy mikor lesz az első három flip-flop kimenete 1-es értékű. Remélem ezekből a példákból már látható, hogy milyen logika mentén lehet tetszőleges bitszámú szinkron számlálót létrehozni. Egyszerűen csak meg kell vizsgálnunk a számláló tartományát és meg kell határozni, hogy mely állapotok után kell invertálni egy adott bitpozícióhoz tartozó flip-flop értékét.

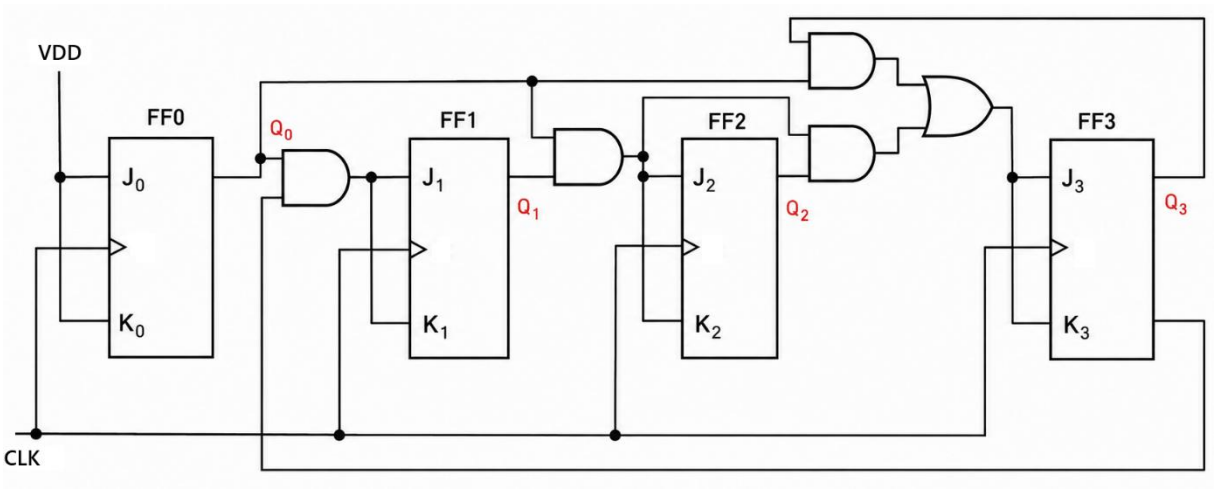
Ennek az alfejezetnek a lezárásaként még nézzük meg, hogy hogyan lehet szinkron tízes számlálót létrehozni. Ennek az aszinkron változatát már láttad korábban és már tudod, hogy ezt egy 4-bites számlóló tartományának a csonkításával lehet létrehozni. Mivel ennek a számlálónak 10 állapota van ([0000-1001]), így sok esetben **MOD10**-es számlálóként szoktak rá hivatkozni. Itt a kérdés az, hogy hogyan tudjuk a szinkron számláló számlálási folyamatát „megtörni”? Ehhez ismért érdemes megvizsgálni a MOD10-es számláló tartományát, ami a 11. táblázatban látható.

Kimenet				
Órajel ciklus	**Q_3**	**Q_2**	**Q_1**	**Q_0**
0 (kezdő állapot)	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

11. Táblázat. MOD10 számláló tartománya.

Vizsgáljuk meg a táblázatban látható mind a négy kimeneti bitet és határozzuk meg, hogy mikor kell az adott bitpozícióhoz tartozó flip-flop-nak invertáni az értékét. Az első flip-flop (0. bitpozíció) órajel periódusonként invertálódik. Ennek alapján az első flip-flop-nak egyszerűen csak váltó üzemmódban kell működni hasonlóan, mint a 4-bites szinkron számlálóban. A második flip-flop-nál az után kell invertálni a flip-flop állapotát, ha $Q_0 = 1$ (világos kékkel kiemelve). De itt észre kell venni, hogy az 1001-es állapotban is $Q_0 = 1$, viszont ezt követően nem kell invertálni a második flip-flop tartalmát mert a következő állapotban, a 0000-ban is $Q_1 = 0$ lesz. Ezt a két feltételt összekapcsolva azt mondhatjuk, hogy a második flip-flop-ot (1. bitpozíció) azt követően kell invertálni, ha $Q_0 = 1$ és $Q_3 = 0$. Ezt átalakíthatjuk úgy, hogy $Q_0 = 1$ és $\overline{Q_3} = 1$, ami lehetővé teszi a flip-flop J és K bemeneteinek vezérlését egy AND kapuval.

A harmadik flip-flop-nak ugyan úgy kell invertálnia az értékét, mint ahogyan azt a 4-bites szinkron számlálónál láttad. Azaz, azt követően, ha $Q_0 = 1$ és $Q_1 = 1$. A számláló láncban a legérdekesebb az utolsó flip-flop lesz, aminek két esetben kell invertálni az értékét a számláló tartományán belül (pirossal emeltem ki a táblázatban). Ezt a két esetet kell összekapcsolnunk. A 0111 állapotban a feltétel az, hogy $Q_0 = 1$, $Q_1 = 1$, $Q_2 = 1$, míg az 1001 állapotban $Q_0 = 1$ és $Q_3 = 1$. Tehát, az utolsó flip-flop-nak azt követően kell invertálni az értékét, ha $Q_0Q_1Q_2 = 1$ vagy $Q_0Q_3 = 1$. Miután meghatároztuk, hogy a számláló láncban az egyes flip-flop-oknak mikor kell invertálni az értékét, a számláló áramköri rajzának megszerkesztése már nem fog problémát jelenteni (43. ábra).



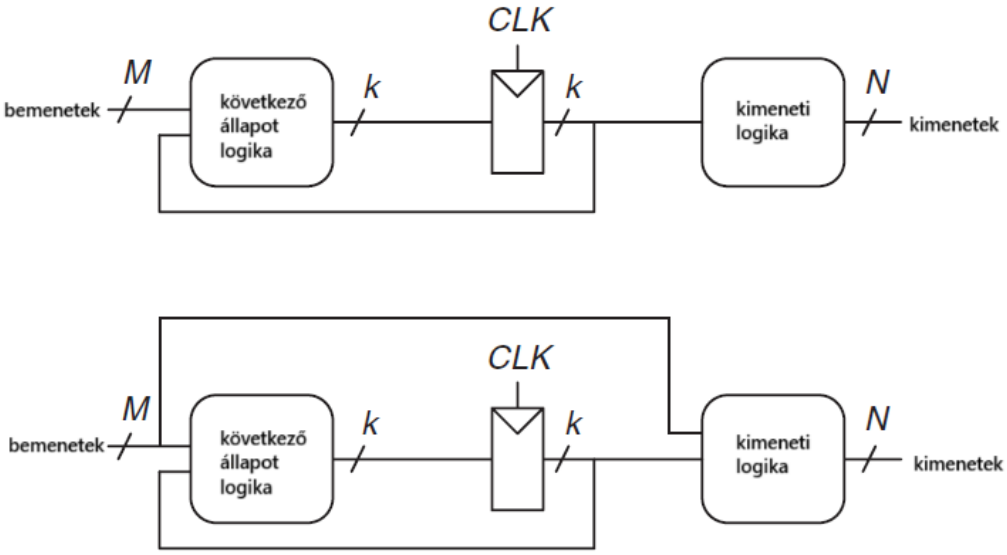
43. Ábra. MOD10 szinkron számláló.

15.4 Frekvenciaosztó

A flip-flop-ok egy további fontos alkalmazási területe a frekvencia osztás (csökkentés). Az elnevezésből könnyen kitalálható, hogy itt a cél, egy gyorsan váltakozó periodikus jelnek (mint például az órajel) a váltakozási sebességét (frekvenciáját) lecsökkenteni. Valójában erre a feladatra is aszinkron számlálókat használunk. Tehát azt mondhatjuk, hogy a számlálókat frekvencia osztásra is lehet alkalmazni. Ahhoz, hogy megértsd ennek az elméleti hátterét ismét tekintsd meg a 39. ábrán látható időzítési diagrammot. Elsőnek azt kell megfigyelni, hogy a láncban a flip-flop-ok kimenetei is egy periodikusan váltakozó négyszög jelet írnak le, hasonlóan, mint a CLK órajel. Ha összehasonlítsuk az A kimenetet a CLK-val, akkor az látható, hogy az A kimenet periódusideje a CLK periódus idejének a kétszerese. Ebből adódik, hogy az A jel frekvenciája a CLK frekvenciájának a fele! Ezt követően, ha összehasonlítjuk a B jelet az A-val, akkor itt is ugyan ez látható. A B jel periódus ideje az A periódus idejének a kétszerese, így a B jel frekvenciája az A frekvenciájának a fele. Ez a jelenség általánosan úgy fogalmazható meg, hogy a számlálóban elhelyezett flip-flop-ok kimenetének a frekvenciája a flip-flop órajel frekvenciájának fele. Mivel az aszinkron számláló láncban a második flip-flop-tól (1. bitpozíció) kezdve minden flip-flop órajelét az őt megelőző flip-flop kimenete adja, így az mondható el, hogy minden flip-flop a hozzá befutó órajel frekvenciáját a felére osztja le!

16. Véges állapotgépek

A **véges állapotgépek (FSM)** széles körben használt szekvenciális logikai áramkörök. A nevükből adódóan, a logikai áramkör több különböző állapotban lehet, ahol az állapotok száma véges (az automata elméletben végtelen állapotgépek is megjelennek). Az állapotgépeknek (a továbbiakban a véges jelzőt elhagyom) két fajtája van: **Moore és Mealy**. A Moore gépekben, a kimenetek csak az áramkör aktuális állapotától függenek. A Mealy gépekben a kimenetek, az aktuális állapoton túl az áramkör aktuális bemeneti értékeitől is függ. Mindkét állapotgép típusra elmondható az, hogy három részből tevődik össze (44. ábra): egy állapotregiszterből és két kombinációs logikai áramkörből.

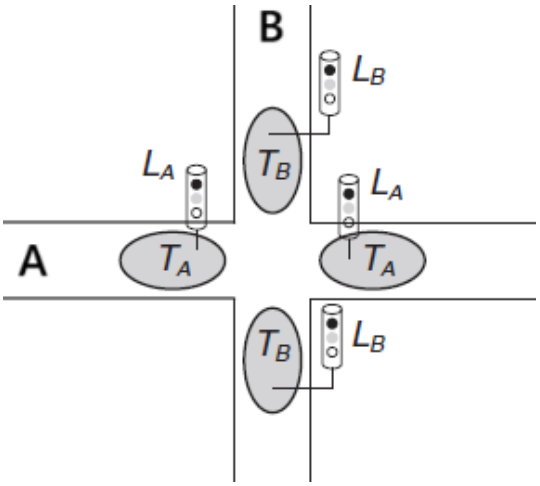


44. Ábra. A Moore (fent) és a Mealy (lent) állapotgép szerkezeti felépítése.

Ezek közül, az egyik kombinációs logikai áramkör, a következő állapot beállításáért felelős. A másik, az állapotgép kimeneti értékeit állítja be. A regiszter feladata az állapotgép aktuális állapotának tárolása, ami nem más, mint egy bináris kód. Mivel az áramkör regiszterrel is rendelkezik, ezért az állapotgép is igényel órajelet és a legtöbb esetben egy **reset** bemenettel is el van látva, ami az állapotgépet alapállapotba (0. állapot) állítja. Az állapotváltás az órajellel van szinkronban, ami függ az aktuális állapottól és az áramkör bementi értékeitől.

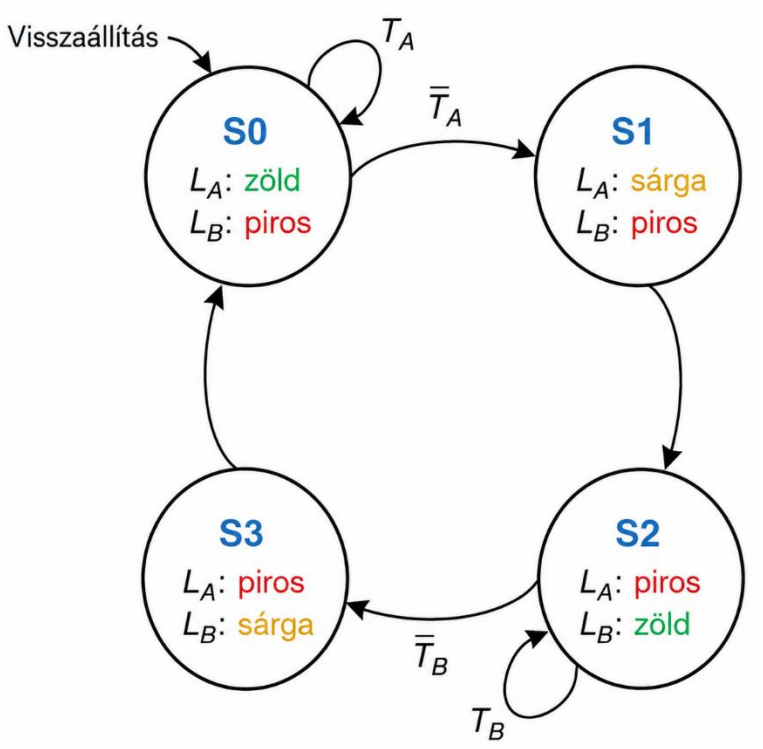
Valójában azt mondhatjuk, hogy a fent ismertetett számlálók is állapotgépek, mivel véges számú állapotban lehetnek és az állapotváltás az órajellel van szinkronban. A számlálóknál

az állapotok száma 2^N, ahol N a számláló szélességét jelöli bitben kifejezve. Egy másik széles körben használt példa az állapotgépek bemutatására a közlekedési jelzőlámpa, ahol az állapotok a jelzőlámba színei. Tegyük fel, hogy neked kell megtervezni az állapotgépet, amely a 45. ábrán látható útkereszteződésben fogja vezérelni az L_A és L_B-vel jelölt közlekedési lámpákat. Itt azt is feltételezzük, hogy az útburkolat alatt forgalomjelző szenzorok vannak beépítve, amik 1-es értéket adnak a kimeneten, ha van forgalom, különben 0-át.



45. Ábra. Útkereszteződés illusztráció állapotgép tervezéshez.

Az állapotgép tervezése során az első és legidőigényesebb lépés az állapot diagram megszerkesztése. Az állapot diagrammokban tipikusan körökkel jelölik az állapotokat, és nyilakkal az állapot átmeneteket. A nyilak felett látható az a feltétel, amelynek a teljesülése esetén fog bekövetkezni az állapot váltás. Ha a nyíl felet nincs semmi, akkor az egy feltétel nélküli állapotváltást jelöl, ami a következő órajel él megjelenésekor fog végrehajtódni. A Moore gépek esetén, a körök belsejében szokták megadni az aktuális állapot kimeneti értékeit. A jelzőlámpás kereszteződéshez készített állapotdiagramm a 46. ábrán látható. Az egyszerűség kedvéért azt feltételeztük, hogy mindösszesen négy állapot van (piros-sárga állapotokat kihagytuk). Ahogy az az állapot diagrammon is látható, egy adott útszakaszon a lámpa mindaddig zöld marad, ameddig van forgalom az útszakaszon. Ha a forgalom megszűnik, akkor a lámpa sárgára, majd pirosra vált. Amikor a lámpa pirosra váltott, azzal egyibőben a másik útszakaszon a lámba bezöldül és az állapotgép elkezdi mintavételezni a másik útszakaszhoz tartozó forgalomjelző szenzort.



46. Ábra. A jelzőlámpát vezérlő állapotgép állapot diagramja.

Miután elkészült az állapotdiagramm, a következő lépés az állapot átmenet táblázat meghatározása a diagramm alapján (12. táblázat). Ebben a táblázatban fel kell venni az állapotgép összes állapotát az állapotban előforduló összes bemeneti kombinációval és meg kell adni, hogy ezekben az esetekben mi lesz a következő állapot. Gyakorlatilag ez egy igazság táblázat, ahol a következő állapot az aktuális állapot és az aktuális bemenetek függvénye. Ezt arra kell felhasználni, hogy megtervezd az állapotgép következő állapotát meghatározó kombinációs logikai áramkört. A 12. táblázatban az X-el jelöl értékek azt jelentik, hogy az adott bemenetnek nincs hatása a következő állapot meghatározásában. A szemléltetés kedvéért, a diagrammon is használt címkéivel lett jelölve az aktuális és a következő állapotok a táblázatban. Ahhoz, hogy ennek a táblázatnak az alapján meg tudd tervezni a következő állapot beállítását végző áramkört, a címkéket le kell cserélni bináris kódokra. Ez a feladatot **állapotkódolásnak** nevezzük. Erre a legegyszerűbb megoldás, ha az állapotokra az indexeikhez létrehozott bináris értékekkel hivatkozunk. Mivel összesen négy állapot van (S0, S1, S2, S3), így két bit elég a **bináris kódoláshoz**. Az állapotok kódolása legyen a következő: S0 – 00, S1 – 01, S2 – 10, S3 – 11. Itt fontos azt kihangsúlyozni, hogy ahány bites kódokat használunk a kódoláshoz, akkora szélességűnek kell lenni az állapotgép regiszterének, mivel ebben tároljuk majd az állapot kódját. A bináris

kódolást használatát követően a 12. táblázatban használt címkéket lecseréltül bináris kódokra. Ennek eredményét szemlélteti a 13. Táblázat.

Aktuális állapot	Bemenetek		Következő állapot
S	T_A	T_B	S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

12. Táblázat. Állapotátmenet táblázat

A táblázatban S_i jelöli az aktuális állapot i. bitpozícióját, míg S'_i jelöli a következő állapot i. bitpozícióját. Most már az is jól látszik, hogy ennek az áramköri résznek két-bites kimenete van. Ehhez a két bitpozícióhoz egy-egy egyenletet lehet felírni, amivel majd meg lehet tervezni a következő állapot beállításáért felelős logikai áramkört. Természetese a cél itt is az, hogy minimalizált egyenleteket használjuk. Ehhez használhatnák Karnaugh táblát is, de vannak olyan esetek, amikor szemrevételezéssel (és egy kis tapasztalattal) is meg tudjuk ezt határozni. Kezdjük az S'_0-vel. Itt nem tudunk egyszerűsíteni, így itt felhasználjuk a kimenet SOP egyenletét változatlan formában. Az S'_1-nél már más a helyzet. Itt az eredeti SOP egyenletet tudtuk egyszerűsíteni és végül még kihasználhatjuk az XOR SOP alakját:

$$S'_1 = \overline{S_1} S_0 + S_1 \overline{S_0} = S_1 \oplus S_0$$

$$S'_0 = \overline{S_1}\ \overline{S_0}\ \overline{T_A} + S_1 \overline{S_0}\ \overline{T_B}$$

Aktuális állapot		Bemenetek		Következő állapot	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

13. Táblázat. Kódolt állapotátmenet táblázat

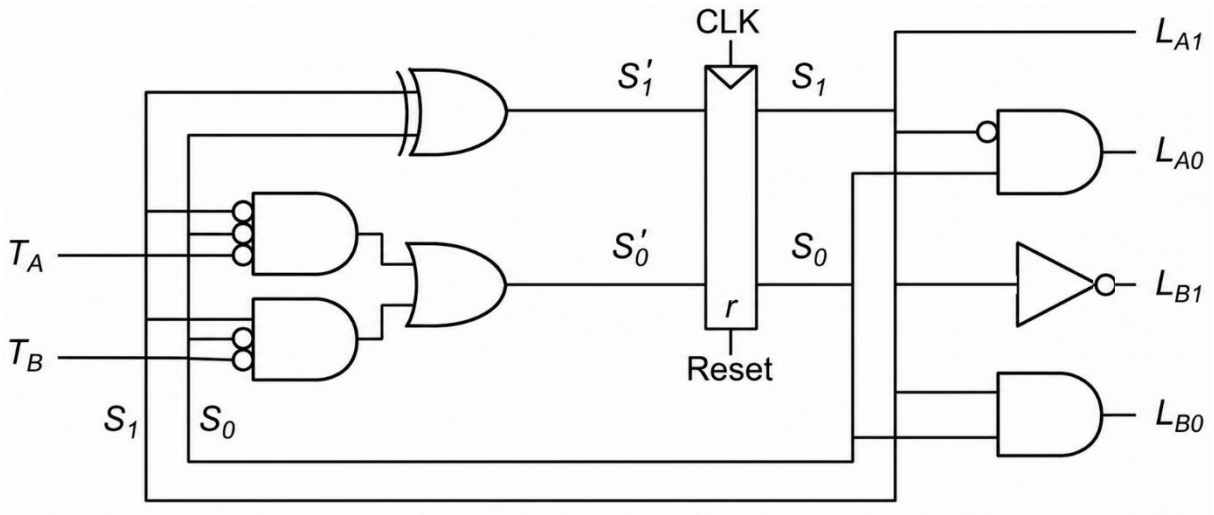
Miután megvannak a kódolt állapotátmenet táblázathoz a logikai egyenletek, a következő lépés, a kimenet igazságtáblájának meghatározása, ami az állapotgép minden állapotához megadja az áramkör kimeneti értékeit. Mielőtt nekiesnénk a kimeneti táblázat meghatározásához, először itt is kódolni kell a lehetséges kimeneteket. Most is célszerű bináris kódolást használni és azokkal hivatkozni a lámpa színeire: zöld – 00, sárga – 01, piros – 10. Mivel három színünk van, így itt is elég két bites kódokat használni. Arról se feledkezz meg, hogy mindkét útszakaszhoz be kell állítani a lámpák színét. Mivel a lámpák színének kódolásához két bites kódokat használtunk, így 4 kimeneti bitünk lesz a táblázatban (14. táblázat). A táblázatban L_{A1} és L_{A0} az egyik jelzőlámpa színét jelölő biteket, míg L_{B1} és L_{B0} jelöli a másik lámpa bitjeit. A feladat itt is az, hogy felírd a kimeneti bitek minimalizált egyenletét. Ezek közül az L_{B0} és L_{A0}-nál nem lehet egyszerűsíteni, így itt az eredeti SOP egyenleteket kell használni. L_{A1} egyenletének meghatározásához azt kell észrevenni, hogy S_1 is abban a két esetben 1-es, amikor L_{A1}. L_{B1} egyenletének felírásánál pedig azt kell meglátni, hogy S_1 is abban a két esetben 0-a, amikor L_{B1} 1-es értékű. Ezek alapján, a kimenetek egyszerűsített egyenletei a következőek:

$$L_{A1} = S_1$$
$$L_{A0} = \overline{S_1} S_0$$
$$L_{B1} = \overline{S_1}$$
$$L_{B0} = S_1 S_0$$

Aktuális állapot		Kimenet			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

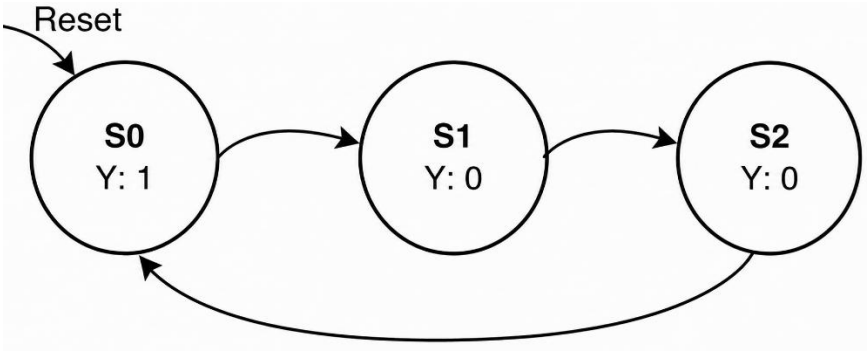
14. Táblázat. Kódolt kimenet táblázat.

Miután meg lett határozva a kimenetek beállításáért felelős áramkör logikai egyenletei, az utolsó lépés az állapotgép áramköri rajzának megtervezése a logikai egyenletek és a korábban megszerzett ismeretek alapján. Itt ismét gondolj az állapotgép három részére. Ezek közül az első az állapotregiszter. Mivel ebben a példában kétbites kódokat használtunk az állapotok lekódolására, így itt az áramkörben is kétbites regisztert kell használni az aktuális állapot kódjának tárolásához. Ez után, meg kell rajzolni a következő állapot meghatározását végző áramköri részt a korábban felírt S'_0 és S'_0 logikai egyenletek alapján. Végül a kimenet beállításért felelős áramköri részt kell megrajzolni az L_{A0}, L_{A1}, L_{B0}, és L_{B1} egyenletek alapján úgy, ahogy az a 47. ábrán látható.



47. Ábra. A jelzőlámpát vezérlő állapotgép áramköri rajza.

Az előző példafeladatban bináris számokkal kódoltuk az állapotokat. Ha másfajta kódolást használnánk, az más áramkört eredményezne. Itt felmerülhet a kérdés, hogy vajon milyen kódolást lenne érdemes használni, ami a legkevesebb logikai kaput és a leggyorsabb működést eredményezi? Sajnos nem létezik legoptimálisabb kódolás, ami mindig hatékonyabb a többinél. Egyszerűen annyit tudunk tenni, hogy kipróbálunk több kódolási lehetőséget és összehasonlítjuk őket. A bináris kódoláson túl, további elterjedt kódolási lehetőség a **one-hot kódolás** és a **one-cold** kódolások. Mind a one-hot, mind a one-cold kódolásban annyi bitre van szükség, ahány állapotunk van. Ez egyben azt is jelenti, hogy a regiszter mérete meg fog egyezni az állapotok számával. A one-hot elnevezése onnan származik, hogy minden kódban csak egy bitpozíción van 1-es érték, az összes többi nulla. Gyakorlatilag az 1-es érték helye jelöli az állapotot. Ebből könnyű kitalálni, hogy a one-cold kódolásban ez pont fordítva van, ott minden bitpozíció 1-es értékű egy kivételével. A one-cold és a one-hot kódolás hasonlósága miatt, mi itt csak a one-hot kódolásra fogunk fókuszálni. A bináris és a one-hot kódolás összehasonlításához vegyünk egy szemléletes példafeladatot, ahol egy olyan állapotgépet kell tervezni, amely három órajelenként egyetlen impulzust generál. Ezt egy órajel osztónak is tekinthetjük. Az első lépés ismét az állapotdiagramm megtervezése, amely a 48. ábrán látható. Ahogy az állapotdiagrammon is látszik, az órajelen (és a reset jelen) kívül most nincs más bemenet, így az állapot átmenetek feltétel nélküliek és az órajellel vannak szinkronban. A három állapot az egymást követő három órajelciklust reprezentálja, amelyek közül csak egy órajel periódusnyi ideig lesz 1-es az Y-al jelölt kimenet.



48. Ábra. Az órajel osztó állapot diagrammja.

Az állapotdiagramm azt is mutatja, hogy a bináris kódolás 2 bitet fog használni az állapotok kódolására (3 állapot) míg a one-hot kódolás 3-bitet (15. táblázat).

Állapot	One-hot kódolás			Bináris kódolás	
	S_2	S_1	S_0	S_1	S_0
S0	0	0	1	0	0
S1	0	1	0	0	1
S2	1	0	0	1	0

15. Táblázat. Az órajel osztó kódolt állapotai.

Ha tudjuk az állapotok kódolását, elkezdhetjük a kódolt állapot átmenet tábálaz megszerkesztését, hasonló módon, mint ahogy azt a korábbi feladatban is tettük. Itt annyi lesz a plusz feladatunk, hogy minkét kódoláshoz el kell készíteni egy kódolt állapot átmenet táblázatot (16. és 17. táblázat).

Aktuális állapot			Következő állapot		
S_2	S_1	S_0	S'_2	S'_1	S'_0
0	0	1	0	1	0

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

16. Táblázat. Kódolt állapot átmenet táblázat a one-hot kódoláshoz.

Aktuális állapot		**Következő állapot**	
S_1	**S_0**	**S'_1**	**S'_0**
0	0	0	1
0	1	1	0
1	0	0	0

17. Táblázat. Kódolt állapot átmenet táblázat a bináris kódoláshoz.

A kódolt állapotátmenet táblázatok ismeretében, meg lehet szerkeszteni a logikai egyenleteket az állapot frissítésért felelős áramkör megszerkesztéséhez. A one-hot állapotátmenet esetén, mivel három kimeneti bitérték van, így három egyenletet kell meghatározni. Ezek a következők:

$$S'_2 = S_1$$
$$S'_1 = S_0$$
$$S'_0 = S_2$$

A bináris kódolásnál csak két bites a kimeneti érték, így itt csak két egyenletre lesz szükség:

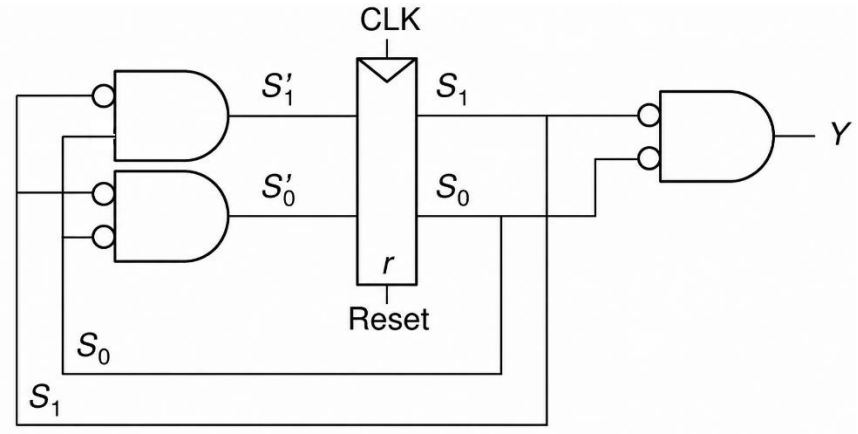
$$S'_1 = \overline{S_1} S_0$$
$$S'_0 = \overline{S_1}\,\overline{S_0}$$

A korábban ismertetett tervezési sorrendet követve, a következő lépés a kimenet táblázatának a megszerkesztése lenne. Viszont, itt egy bináris kimenet van, ami csak az

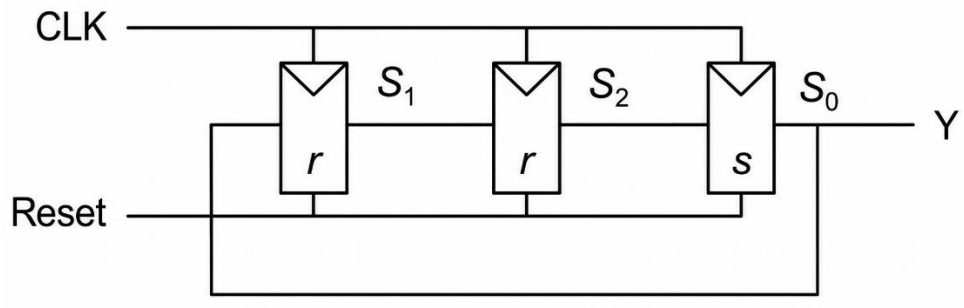
S0 állapotban 1-es. Ebből fakadóan, a kimenet egyenletének a felírása nem is igényel külön táblázatot, mert ez nagyon egyszerűen meghatározható mindkét kódolási módhoz:

$$Y_{one-hot} = \mathrm{S}_0$$
$$Y_{bináris} = \overline{\mathrm{S}_1}\,\overline{\mathrm{S}_0}$$

Ezt követően már minden adott az áramkörök megtervezéséhez a one-hot és a bináris kódolási formához. A bináris kódoláshoz létrehozott áramkör rajza a 49. ábrán, míg a one-hot kódoláshoz létrehozott áramkör az 50. ábrán tekinthető meg.



49. Ábra. Az órajel osztó áramköri rajza a bináris kódolással.

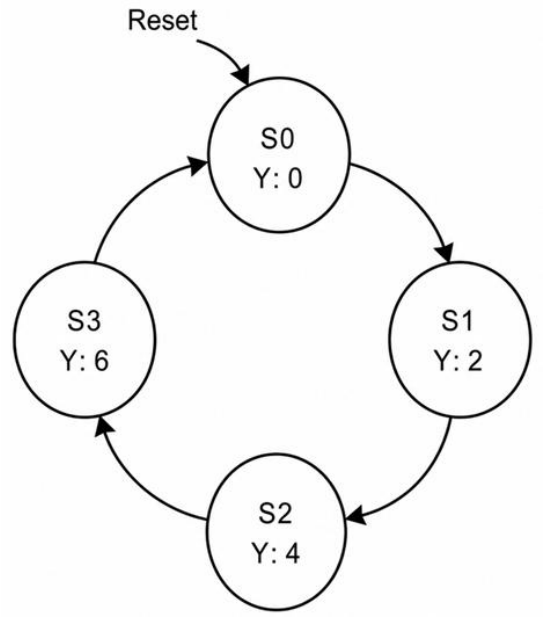


50. Ábra. Az órajel osztó áramköri rajza a one-hot kódolással.

Figyeld meg, hogy a one-hot kódoláshoz megszerkesztett logikai áramkör csak egy regiszterből áll és nem igényelt további logikai kapukat. Persze itt a regiszter 3-bites, míg a bináris kódolásnál csak 2-bites, de az áramkör koncepciója egyszerűbb, ezért ebben a példában a one-hot kódolást lenne célszerűbb használni. Viszont fontos kihangsúlyozni, hogy az ilyen esetekben mindig mérlegelni kell melyik a költségesebb: a memória mérete vagy a plusz logikai kapuk felhasználása.

16.1 Számlálótervezés

Korábban már említettem, hogy a számlálók is állapotgépeknek tekinthetők. Az állapotgépekről szerzett ismeretek felhasználásával lehetőség nyílik tetszőleges számlálási lánccal rendelkező számlálók tervezésére. Ez azt jelenti, hogy egy n-bites számlálónak nem kell az n-bit által biztosított tartományon egyesével végig menni, hanem helyette, a tartomány elemein tetszőleges sorrendben lehet haladni. Vegyünk egy példát. Tegyük fel, hogy egy olyan 3-bites számlálót szeretnénk létrehozni, ami a 3-bites tartomány páros értékein halad végig (kettesével lépked): 000, 010, 100, 110. Ezt is úgy kell megtervezni, mint egy állapotgépet, ahogyan azt a fejezet elején láttad. Az első lépés az állapotdiagram felrajzolása, ami a 51. ábrán látható.



51. Ábra. Saját tervezésű 3-bites számláló állapotgépének diagrammja.

Miután az állapotdiagram elkészült, jöhet az állapot átmenet táblázat felírása (18. táblázat). Annak ellenére, hogy csak 4 állapot van az állapotdiagrammon, egy 3-bites számlálót kell tervezni, mert legalább 3-bit kell ahhoz, hogy 0-tól elszámoljunk 6-ig kettesével. Tehát a számláló aktuális értéke 3 filp-flop-ban lesz tárolva, így az állapotkódok 3-bitesek lesznek. Jelöljük a flip-flop-ok kimeneteit Q_0, Q_1 és Q_2-vel és a következő állapot i. bitjét Q'_i-vel.
Az állapot átmenet tábla felírását követően azt is el kell dönteni, hogy milyen típusú flip-flop-okból fogjuk megépíteni a számlálót. Ebben a példában JK flip-flop-okat fogunk használni. Miután ez eldőlt, fel kell használni a flip-flop állapotátmenet táblázatát (19.

táblázat). Ebben az összes lehetséges jelenlegi (Q_n) és következő állapotot (Q_{n+1}) felsoroljuk. Minden kombinációhoz meg van adva a J és K bemenetek, amelyek az átmenetet (ha egyáltalán van átmenet) okozzák. Ebben a táblázatban is, az X-szel jelölt értékek tetszőlegesen lehetnek 0 vagy 1.

Állapotátmenet táblázat					
Q_2	Q_1	Q_0	Q'_2	Q'_1	Q'_0
0	0	0	0	1	0
0	1	0	1	0	0
1	0	0	1	1	0
1	1	0	0	0	0

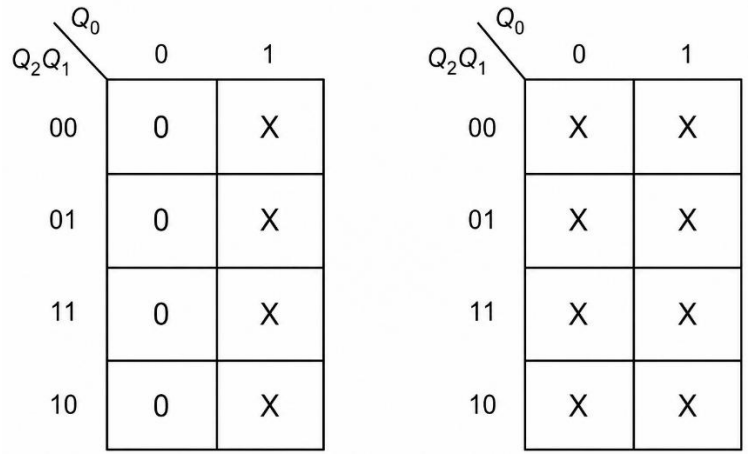
18. Táblázat. A saját tervezésű 3-bites számláló állapot átmenet táblázata.

Kimenet változása			Flip-flop bemenetek	
Q_n		Q_{n+1}	J	K
0	→	0	0	X
0	→	1	1	X
1	→	0	X	1
1	→	1	X	0

19. Táblázat. JK flip-flop állapot átmenet táblázata.

A következő lépésben, a 18. és 19 táblázatokra támaszkodva fel kell írni a következő állapot meghatározásáért felelős áramköri rész logikai egyenleteit. Az egyenletek felírásánál törekedni kell a lehető legegyszerűbb egyenletalak meghatározására. Ennek érdekében

Karnaugh táblát fogok használni az egyenletek felírásához. Továbbá, mivel JK flip-flop-okból lesz felépítve a számláló, ezért mind a J, mind a K bemenetek vezérléséhez fel kell írni 1-1 egyenletet mindhárom flip-flop-hoz. Példaként megadom az első flip-flop J és K bemenetéhez tartozó Karnaugh táblákat (52 ábra). A többi felrajzolását rád bízom.

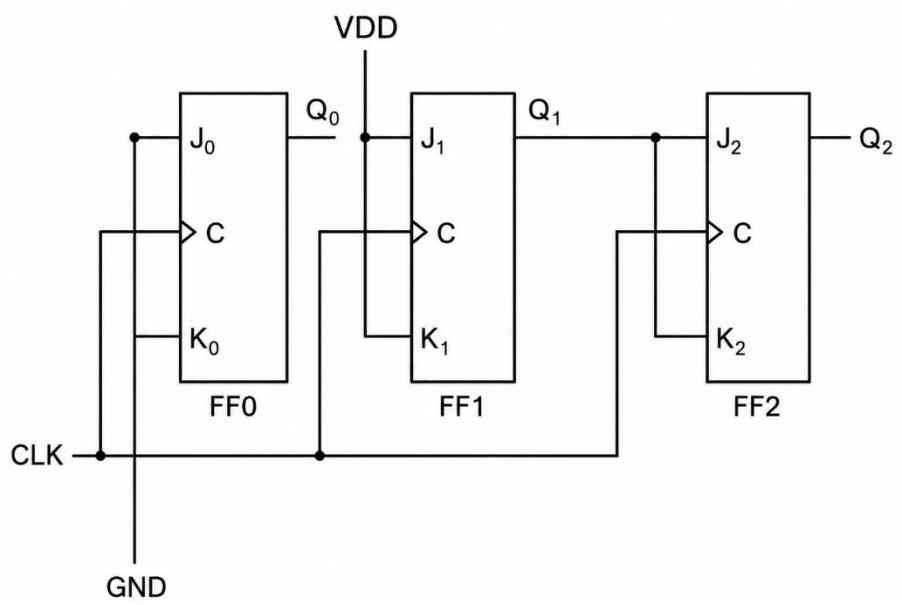


52. Ábra. Saját tervezésű 3-bites számláló első flip-flop-ja J és K bemeneteihez felírt Karnaugh táblák.

Miután felrajzoltad a táblázatokat, a már megszokott módon, meg kell határozni minden egyes táblázat egyenletét. Fontos kihangsúlyozni a X-eket tartalmazó cellák speciális kezelését. Az X-el jelölt cellákat lehet 0-nak és 1-es értékűnek is tekinteni, ha ezzel növelhető a táblázatban kialakított körök mérete. Ezt figyelembevéve, a 3 flip-flop J és K bemeneteinek vezérlését leíró egyenletek a következőek:

$$J_0 = 0$$
$$K_0 = 0$$
$$J_1 = 1$$
$$K_1 = 1$$
$$J_2 = Q_1$$
$$K_2 = Q_1$$

Az utolsó lépés a kombinációs logika megtervezése, ami a 3 darab JK flip-flop helyes összekötését jelenti, az imént meghatározott formulák felhasználásával (53. ábra).



53. Ábra. Saját tervezésű 3-bites számláló kapcsolási rajza.

Irodalomjegyzék

[1]. S. L. Harris, D. M. Harris, Digital design and computer architecture, ARM edition. Morgan Kaufmann, 2016.

[2]. T. L. Floyd, Digital Fundamentals. 11th Edition. Pearson, 2015.

[3]. M.M Mano, C.R. Kime, T. Martin, Logic and Computer Design Fundamentals, 5th Edition. Pearson, 2015.