Szabadkai Műszaki Főiskola

Burány Nándor
Divéki Szabolcs

Digitális elektronika

-jegyzet-

Szabadka, 2007.

Előszó

A jegyzet a Szabadkai Műszaki Főiskola másodéves hallgatóinak íródott, a *Digitális elektronika* tantárgy előadásainak anyagát tartalmazza. Ez a tárgy a *Digitális technika* és az *Analóg elektronika* tantárgyak közvetlen folytatása. A *Digitális technikában* a digitális tervezés logikai alapjaival ismerkedtek meg a hallgatók. Az *Analóg elektronika* logikai áramkörökről szóló fejezete a digitális berendezések alapját képező hardvermegoldásokba nyújtott betekintést.

A *Digitális elektronika* tantárgy tanmenetéhez igazodva, a jegyzet a digitális áramkörök ma jellemző megvalósítási módjait tárgyalja. Először (I. rész) a fizikai megvalósítás körüli általános kérdésekkel foglalkozunk. Ezt követi (II. rész) a kis- és közepes integráltságú (*SSI, MSI*) áramkörökre épülő tervezés, amely a XX. század hetvenes-nyolcvanas éveiben érte el a tetőfokát.

A XX. század hetvenes éveitől kezdve jelentek meg a szoftveres programozású áramkörök (mikroprocesszorok, mikrovezérlők) majd később a hardveres programozású áramkörök (*programmable logic device – PLD*). A szoftveres programozású megoldásokkal más tantárgyakban ismerkednek meg a hallgatók. A *PLD*-kre épülő digitális tervezést viszont ebben a tantárgyban tárgyaljuk részletesen, így a jegyzet III. része a hardveres programozás korszerű eszközét, a *Verilog* hardver leíró nyelvet ismerteti. Itt a feladat leírásáig és számítógépes szimulációjáig fogunk eljutni. Az áramkör gépi szintézisét végző szoftverekkel és magával a programozási eljárással az Elektronikai készülékek tervezése nevű tantárgyban ismerkednek meg a hallgatók.

A mai tervezésre jellemző, hogy a digitális rendszerekben a szoftveres vezérlésű mikroprocesszorok és mikrovezérlők, valamint a hardveres programozású *PLD*-k végzik a feladat oroszlánrészét, az SSI és MSI áramköröknek kisegítő szerepük van (meghajtás, szintillesztés stb.).

A szoftveres eszközök szélesebb körben elterjedtek és kiválóan alkalmasak összetett vezérlési-, mérési-, kijelzési- stb. feladatok elvégzésére. Léteznek viszont olyan gyors jelfeldolgozási- vezérlési- és egyéb feladatok, amelyek nem oldhatók meg mikroprocesszorokkal, illetve mikrovezérlőkkel. Ebben az esetben a hardveres programozású eszközök jelentik a kiutat.

Bizonyos mértékben a szoftveres programozású és a hardveres programozású eszközök egymásnak konkurrenciát jelentenek, de sok helyen ki is egészítik egymást. Számos készülékben a mikrovezérlő mellett ott van egy vagy több *PLD* is.

A progamozható logikai áramkörök ma olyan bonyolultságot értek el, hogy bennük akár mikrovezérlő is megvalósítható. Így a felhasználó a saját igényei szerint alakíthat ki szoftveres programozású eszközt, megfelelő hardverlehetőségekkel és tetszés szerinti utasításkészlettel. Mivel az egyes tervezők kapacitását egy ilyen feladat rendszerint meghaladja, a *PLD* gyártók mindinkább támogatják ezt az irányzatot kész tervekkel.

A szerzők

Bevezető

Ma a digitális megoldások jellemzőek a műszaki élet számos területén. Elsősorban az ipari folyamatok irányítását, a számítástechnikát és a híradástechnikát emelnénk ki.

Kezdetben a digitális berendezések megvalósítására csupán mechanikai üzemeltetésű kapcsolók és mágneskapcsolók (jelfogó, relé) álltak rendelkezésre. Az analóg jelfeldolgozásban sokáig egyeduralkodó elektroncsövekkel is történtek próbálkozások digitális berendezések megvalósítására: az első elektronikus számítógépek elektroncsövekkel épültek. Sajnos a nagy méretek, a nem kielégítő megbízhatóság és a jelentős fogyasztás megakadályozta ezeknek a megoldásoknak a szélesebbkörű elterjedését.

A tranzisztor felfedezésével szinte egyidejűleg kezdődtek el az analóg és a digitális alkalmazások. Tranzisztorok, diódák és ellenállások összekapcsolásával viszonylag kis méretű, kis fogyasztású és gyors digitális áramköröket lehetett kialakítani. A XX. század 50-es éveinek végétől a digitális elektronika alapját egyértelműen az integrált megoldások képezik. A félvezető alkatrészek gyártástechnológiájának fejlesztésével lehetőség nyílt, hogy mind bonyolultabb áramköröket építsenek (integráljanak) egyetlen kristály lapocska felületére.

Az áramkörök integrálásának kezdetén néhány logikai kaput sikerült egy lapocskára integrálni, ezt nevezzük kis fokú integrálásnak (*small scale integration – SSI*). Ezt követően a közepes fokú integrálás (*medium scale integration – MSI*) már lehetővé tette, hogy bonyolultabb egységek (multiplexerek, számlálók stb.) kerüljenek egy lapocskára illetve egy áramköri tokozásba. A digitális berendezéseket ezeknek az SSI és MSI áramköröknek az ügyes kombinációival alakították ki.

Akkor úgy tünt, hogy a fejlődés iránya a mind bonyolultabb áramkörök kialakítása, amelyek teljes egészében integrálják az egy berendezéshez szükséges digitális elektronikai megoldásokat. Ilyen áramkörök születtek is, pl. digitális órákhoz, mérőműszerekhez stb., azonban hamarosan kiderült, hogy az alkalmazások annyira sokrétűek, hogy nem gazdaságos minden alkalmazáshoz külön integrált áramkört (*application specific integrated circuit – ASIC*) kifejleszteni. Az integrált áramkörök fejlesztése és gyártása csak nagy sorozatok esetén gazdaságos.

Ezeket a korlátozásokat felismerve a fejlődés két irányban haladt tovább. Egyrészt megjelentek a mikroprocesszorok, amelyek önmagukban semmi konkrét célra nem használhatók, de megfelelő segédáramkörökkel (memóriák, illesztőegységek stb.) összekapcsolva és szoftver hozzáadásával úgyszólván tetszőleges feladat megoldására tehetők alkalmassá. Másrészt megjelentek a hardveres programozású digitális alkatrészek (*programmable logic device – PLD*), amelyek nagyszámú előregyártott logikai blokkot tartalmaznak, ezeket hardveres programozással olyan módon köthetjünk össze, hogy a kívánt feladatot végezzék. Mindezek az áramkörök a nagyintegráltságú (*large scale integration – LSI*) illetve a nagyon nagy integráltságú (*very large scale integration – VLSI*) kategóriákba tartoznak.

A szerzők

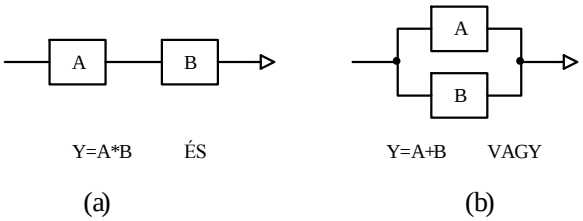
I. A digitális áramkörök fizikai megvalósításának kérdései

1. A digitális áramkörök fizikai jellemzői

Ebben a fejezetben csak a digitális áramkörök fizikai jellegzetességeire összpontosítunk. Az itt tárgyalt jellemzők egyaránt vonatkoznak az egyszerűbb felépítésű *SSI* és *MSI* áraköröke és bonyolultabb (*LSI, VLSI*), szoftveres és hardveres programozású rendszerekre. Ennek oka, hogy hasonló alapanyagokat és hasonló gyártástechnológiát alkalmaznak mindezeknél a digitális áramköröknél, sőt az egyes áramköri megoldások is nagyon hasonlóak.

1.1. Áramlogika – feszültséglogika

A digitális berendezések a kezdeti szakaszban kapcsolókból és jelfogókból épültek fel. Ezeknél a logikai funkció megvalósítása úgynevezett áramlogikával történt: ha a kapcsolók megfelelő állapotainak a köszönhetően áram jutott a vezérelt berendezésre, azt tekintették a logikai egyes állapotnak. Ha ugyanez nem történt meg, valamelyik kapcsoló bontott állapota miatt, az a logikai nulla állapotnak felelt meg. Az 1-1 ábra az *ÉS* illetve a *VAGY* logikai függvény áramlogikás megvalósítását szemlélteti.

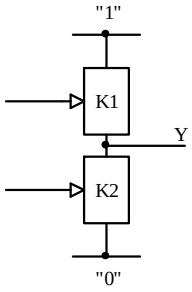


1-1 ábra: Az ÉS (a) és a VAGY (b) logikai függvény megvalósítása áramlogikás technikával.

Mára az ilyen áramlogikás kapcsolások úgyszólván teljesen kihaltak, viszont az utóbbi években az egyes integrált áramköri megoldásoknál újra kezdenek előtérbe kerülni. Általánosságban véve ma a digitális áramköröknél a feszültséglogika a jellemző. Itt is kapcsolók nyitásával és zárásával hozzuk létre a kívánt logikai állapotokat, de a cél nem az áram áthaladása a bementi pont(ok) felől a kimeneti pont(ok) felé, hanem bizonyos szabványos feszültségszintek létrehozása a logikai nulla és a logikai egyes reprezentálására.

Az 1-2a ábra egy feszültséglogikás digitális (logikai) elem elvi rajzát adja. Két kapcsoló (*K1, K2*) van egymással sorba kötve (tekinthetjük ezt feszültségosztónak is), a szabad kivezetések viszont a logikai egyesnek (“1”) valamint a a logikai nullának (“0”) megfelelő feszültségszinteket biztosító forrásokhoz kapcsolódnak. Rendszerint egy forrásról van szó, amelynek egyik végét a földpontra kötjük és a földpont potenciálja körüli értékeket tekintjük logikai nullának, a forrás másik végén levő potenciál a logikai egyes állapotnak felel meg. A kapcsolók bekapcsolása elvileg négy módon variálható, amit az 1-2b ábrán megadott táblázat szemléltet.

A mai digitális berendezések rendszerint két feszültségszinttel, két logikai értékkel működnek. A táblázat első sorának megfelelő esetben az áramkör nem egy határozott feszültségszintet valósít meg, ezzel eltértünk a kétértékű logika definíciójától. Mivel mindkét kapcsoló ki van kapcsolva az *Y* kimenet nagyimpedanciás-, más néven harmadik állapotba (angolul *tri-state* vagy 3-*state*) kerül. A feszültségszintet nem ez a logikai elem határozza meg, hanem az ugyanerre a pontra kapcsolt más elemek. Ezt a lehetőséget olyan digitális berendezéseknél szokták igénybe venni, amelyeknél egy közös vezetéken valósul meg több logikai elem között a kommunikáció. A különböző időintervallumokban különböző logikai elemeket aktivizálunk. A pillanatnyilag aktív elem vezérli a vezetéket, a többi elem a nagyimpedanciás állapotnak köszönhetően érdemben nem befolyásolja a logikai szinteket.



(a)

K1	K2	Y
0	0	Harmadik állapot
0	1	0
1	0	1
1	1	Tilos állapot

(b)

1-2 ábra: Feszültséglogikás logikai elem elvi rajza (a) és a kapcsolók állapotának variációi (b).

A táblázat második sorában *K2* vezet, *K1* nem vezet, így az *Y* kimenet rövidzárban van a földponttal, ami által logikai nullát valósítunk meg. A harmadik sorban a fordított eset áll elő, *K1* vezet, *K2* nem vezet, így a kimenetre a logikai egyesnek megfelelő potenciál jut. A valós kapcsolók nem képeznek teljes rövidzárat bekapcsoláskor. Ebből kifolyólag a logikai elem kimenetének terhelésekor a feszültség változik, a logikai szintek eltérnek az ideális értékektől. Hasonlóképpen, a kapcsolók ki- és bekapcsolási idejei végesek, így a vezérlőjelek hatása csak késleltetve érvényesül, ami késleltetést okoz a logikai elem működésében. Ezekkel a gondokkal a következő szakaszban fogunk szembesülni.

A táblázat utolsó sorába azt jegyeztük be, hogy ez a kapcsoló kombináció tilos: nem engedhető meg, hogy a *K1* és *K2* kapcsolók egyidőben vezessenek, mert ez által a tápfeszültséget rövidre zárnánk. Rövidzár esetén a kialakuló nagy áram tönkretenné a kapcsolókat.

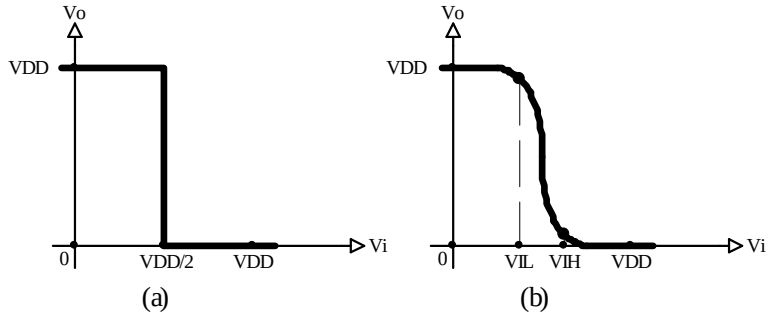
1.2. Fizikai jellemzők

A digitális áramkörök adatlapjai a logikai funkció mellett megadják az áramkör fizikai jellemzőit. Ilyen fizikai jellemzők az átviteli jelleggörbe, a névleges logikai szintek, a zajtűrés, a különböző késések, a kimenetek terhelhetősége, a fogyasztás, a hőmérsékleti tartományok, tokozás stb. Most ezeket a jellemzőket definiáljuk és elemezzük hatásukat az áramkör működésére.

1.2.1. Átviteli jelleggörbe

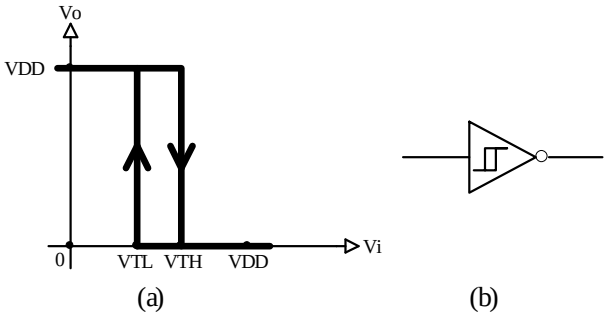
Ha az 1-2a ábrán bemutatott logikai elem kapcsolóinak vezérlését úgy oldjuk meg, hogy egy közös vezérlőjel hatására ellenfázisban működjenek, az 1-3a ábrán bemutatott idealizált átviteli jelleggörbét kapjuk. Amíg a bemeneti vezérlőjel a tápfeszültség fele ($V_{CC}/2$) alatt van, a felső kapcsoló vezet és magas logikai szintet tart a kimeneten. A tápfeszültség felénél nagyobb értékekre az alsó kapcsoló vezet, ami a kimenetet alacsony logikai szintre húzza. Mivel alacsony bemeneti logikai szint magas kimeneti logikai szintet eredményez és viszont, ez a jellegörbe logikai *NEM* függvénynek (inverter) felel meg.

Valós esetben a kapcsolók ellenállása a bemeneti jel változásának hatására nem ugrásszerűen, hanem folyamatosan változik. Ennek eredménye a valós logikai inverterekre jellemző, az 1-3b ábrán megrajzolt átviteli jelleggörbe. A diagramon megjelöltük azokat a pontokat, amelyeknél a görbe meredeksége -1 értékű. Ezek a bemeneti feszültségértékek (V_{IL}, V_{IH}) között a görbe meredeksége nagy, kis bemeneti változásokra a kimeneti logikai szint nulláról egyesre válthat és fordítva, ami miatt a kapcsolás működtetése ebben a tartományban nem kívánatos (tiltott zóna).



1-3 ábra: Logikai inverter átviteli jelleggörbéi: (a) ideális eset, (b) valós eset.

Az 1-3 ábrán bemutatott egyértelmű átviteli jellegörbék mellett alkalmaznak kétértelmű, hiszterézises (Schmitt féle) jelleggörbét is (1-4a ábra). A megfelelő áramkörökben megvalósított belső pozitív visszacsatolásnak köszönhetően az átmenet az egyik logikai szintről a másikra nem fokozatos, hanem ugrásszerű. Ez által a hosszú felfutási- és lefutási idejű digitális jelek négyszögesíthetők és javítható a zajtűrés (1.2.3 pont). A Schmitt féle jelleggörbét az áramkörök rajzjelein is fel szokták tüntetni: az 1-4b ábra egy Schmitt féle inverter rajzjelét mutatja. Bármely más digitális áramkör bemenetére is alkalmazható hiszterézis.



1-4 ábra: Schmitt féle logikai inverter átviteli jelleggörbéje (a), és rajzjele (b).

1.2.2. Logikai szintek

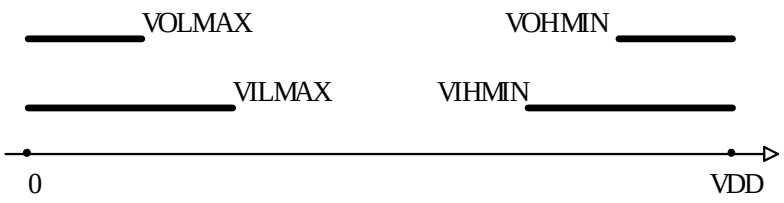
A tiltott zóna alatt és felett a kimeneti logikai szintek (V_{OL}, V_{OH}) megközelítőleg stabilak, nem függenek jelentősen a bemeneti jeltől, a terhelőáramra is csak enyhén reagálnak, a tápfeszültséget viszont stabilnak tekintjük. A két szint közötti különbséget logikai amplitúdónak nevezzük.

Szabályosan megszerkesztett digitális áramköröknél a kimeneti alacsony logikai szint minden esetben lényegesen alacsonyabb a bemeneten alacsony logikai szintként elfogadható értéknél ($V_{OL}<V_{IL}$). Hasonlóképpen, a kimeneti magas logikai szint jelentősen meghaladja a bementi magas szintű jel alsó határértékét ($V_{OH}>V_{IH}$).

Ez a két feltétel teljesítése szükséges ahhoz, hogy a digitális áramkörök elemeinél kaszkád kötést alkalmazhassunk, akár az integrált áramkörök között, akár azokon belül. Mivel a digitális rendszerekben a jelfeldolgozás általában több egymást követő fokozatban történik, az elemek kaszkád kötése elkerülhetetlen.

1.2.3. Zajtűrés

Mind a bemeneti, mind a kimeneti logikai szintekre az adatlapok nem konkrét értékeket, hanem értéktartományokat adnak meg, tekintettel a tápfeszültség esetleges változásaira, a terhelés következményeire, a technológiai eltérésekre és egyéb tényezőkre. Ha ezeket az értéktartományokat egymás feletti tengelyeken ábrázoljuk (1-5 ábra), megállapíthatjuk, hogy mekkora zavarokat viselhet el adott áramkör bemenete, hogy a kimeneten ne jelentkezzék téves logikai szint.



1-5 ábra: A digitális áramkörök zajtűrésének értelmezése.

Az ábra szerint alacsony bemeneti logikai szint esetén a zajtűrés a:

$$NM_0 = V_{IL\max} - V_{OL\max}$$

képlettel adott, míg magas logikai szint esetén:

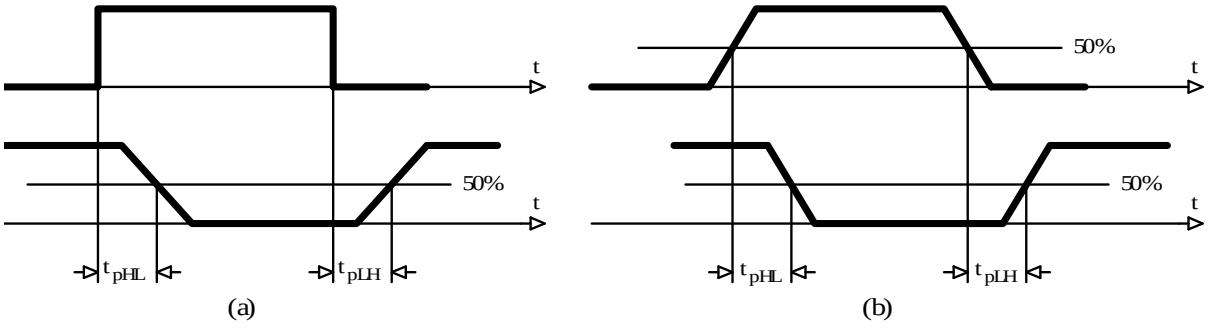
$$NM_1 = V_{OH\min} - V_{IH\min}\,.$$

Mind később látni fogjuk, a nagy zajtűrés előnyös tulajdonság ugyan, de nagy tápfeszültséget feltételez, ami nagy fogyasztással is jár együtt. Nagy zajtűrésű áramkörök alkalmazása főleg a zajos ipari környezetekben ajánlatos.

1.2.4. Késések

A digitális áramkörök kapcsolótranzisztorainak ki-be kapcsolása véges időt vesz igénybe. Az áramkör késése a bementi vezérlőjel kialakulásától a kimeneti logikai szint létrejöttéig eltelt idő. A késésre a véges kapcsolási idők mellett kihatással vannak a parazita kapcitások is. A rendelkezésre álló véges töltő-ürítő áramok a logikai szintek megváltoztatását véges idő alatt végzik.

Az 1-6a ábra egy logikai inverter késéseit szemlélteti idealizált bemenő jelre (négyszögjel). A bemenő jel ugrásait követően a kimeneti logikai szintek megváltozásai t_{pHL} illetve t_{pLH} késési idők elmúltával következnek be. Valós esetben a (1-6b ábra) az áramkör bemenetén nem tudunk idealizált négyszögjelet biztosítani, helyette a jelet egy, a vizsgált inverterhez hasonló, előző fokozatból kapjuk. Ez esetben a késéseket rendszerint a logikai amplitúdó vagy a tápfeszültség felénél (az 1-6b ábrán 50%-kal jelöltük) meghúzott egyenesnek a jellel alkotott metszéspontjai között számoljuk. A lefutó él késése (t_{pHL}) rendszerint különbözik a felfutó él késésétől (t_{pLH}).

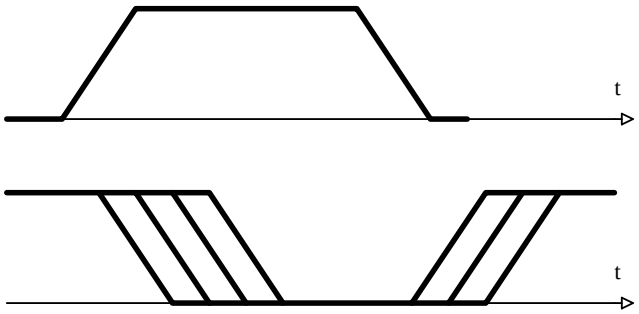


1-6 ábra: Digitális áramkör késései: (a) idealizált bemenő jel mellett, (b) valós bemenő jel esetére.

Az integrált áramkörök belső késései mellett meg kell említeni, hogy késések jelentkeznek az összekötő vezetékeken is. Ez különösen kifejezett a nyomtatott áramköri vezetékeken, de *LSI*, *VLSI* áramköröknél az egyes tömbök közötti belső összekötések okozta késéseket is figyelembe kell venni.

A késések nem tekinthetők állandónak: a sorozatgyártásban jelentkező különbségek, a tápfeszültség-, a terhelés- a hőmérséklet változásai és egyebek miatt a gyártói adatlapok a késésekre

nem meghatározott értékeket, hanem értéktartományokat adnak meg. Az értéktartományokat a diagramokon az 1-7 ábrán bemutatott módon szokták ábrázolni. A késések hatásainak elemzésekor a lehető legkedvezőtlenebb esetet kell figyelembe venni.



1-7 ábra: A késési tartományok grafikus ábrázolása a digitális áramkörök adatlapjain.

1.2.5. A kimenetek terhelhetősége

A digitális árakörök kapcsolóinak ellenállása bekapcsolt állapotban nem nulla, kikapcsolt állapotban nem végtelen, ugyanakkor a kapcsolók vezérlésére véges áram szükséges. A digitális rendszerek kiépítésekor egy logikai elem egy vagy több másik, hozzá hasonló logikai elemet hajt meg. Ritkán adódik olyan eset, hogy nem logikai áramkört hanem más fogyasztót (LED, átviteli vonal stb.) kell meghajtani.

A véges kimeneti és bemeneti ellenállások miatt a meghajtható elemek száma véges. Túlterhelés esetén a logikai szintek eltolódnak, ami a következő fokozatoknál téves reakciókat válthat ki. A késések növekedésével is számolni kell a terhelés következtében.

A kimenet terhelhetőségét nem a konkrét terhelési árammal szokták megadni, hanem a kimenetre kapcsolható logikai bemenetek számával (egy vagy egynél nagyobb természetes szám). Ugyanazon az áramkörcsaládon belül is a bemeneti áramok változóak, ezért a gyártók úgynevezett szabványos bemeneti áramokkal szoktak számolni.

Rendszerint logikai nulla illetve logikai egyes esetén a bemenet nem egyenlő mértékben terheli a kimenetet. A terhelhetőség számításakor a kedvezőtlenebb esetet kell figyelembe venni.

1.2.6. Fogyasztás

Működés közben a digitális áramkörök bizonyos áramot vesznek fel a tápforrásból. Az áramfelvétel függ a kimenetek logikai szintjétől, ezért az adatlapokon rendszerint átlagértéket tüntetnek fel. Az áramkörök fogyasztása többé-kevésbé függ a logikai szintek változásának gyakoriságától is (működési frekvencia). Ennek egyik oka, hogy az 1-2a ábrán bemutatott logikai elemnél az alsó és a felső kapcsoló vezetése között (a logikai szintek változtatásakor) némi átfedést alkalmaznak a logikai késések minimalizálása érdekében. Mint említettük, a kapcsolók ellenállása vezetési állapotban véges így az átfedés során nem alakul ki végtelen áram, de a fogyasztás jelentősen megnől a statikus állapothoz képest. Hasonlóan a fogyasztás növekedését eredményezi a parazita kapacitások töltése-ürítése is. Az átmeneti áramimpulzusok az elsődleges oka, hogy a digitális áramkörök tápfeszültségét jó minőségű kondenzátorokkal szűrni kell, minél közelebb magához az áramkörhöz.

Mivel a fogyasztás rendszerint összefüggésben van az áramkörre alkalmazható legnagyobb üzemi frekvenciával, a különböző áramkör családok összehasonlításakor a fogyasztás és a késés szorzatát szokásos alapul venni. Ezt a szorzatot *PDP*-vel jelöljük (*power-delay product*) és az energia mértékegységével (*J* illetve *pJ*) mérjük:

$$PDP = P_D t_p .$$

A *pJ* azért alkalmas nagyságrend, mert a késések rendszerin *ns* nagyságrendűek, a fogyasztás pedig *mW* nagyságrendű. Egy áramkörcsalád akkor számít jobbnak, ha az adott technológiával megvalósított logikai elemek *PDP*-je kisebb.

1.2.7. Hőmérsékleti tartományok

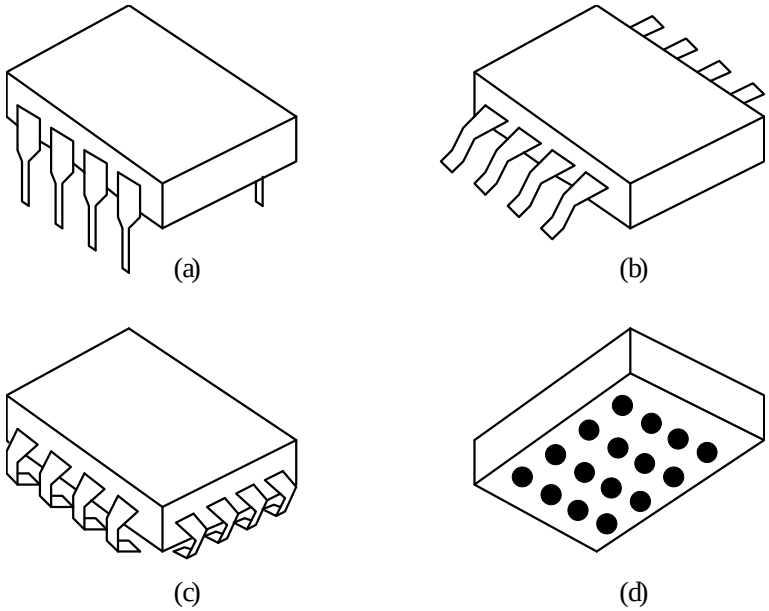
Az integrált áramkörök üzemi és tárolási hőmérsékleti tartományait a gyártó az adatlapokon definiálja. Általában három hőmérsékleti tartományt említenek: kereskedelmi, ipari és katonai tartományok. Rendszerint ugyanaz az áramkör beszerezhető mind a három hőmérsékleti tartományra. A kereskedelmi hőmérsékleti tartomány 0^oC-tól $+70^oC$ környezeti hőmérsékletig terjedő működést engedélyez, az ipari tartomány -25^oC-tól $+85^oC$-ig, a katonai tartomány -55^oC-tól $+125^oC$-ig. A tárolási hőmérsékleti tartomány rendszerint egységes, pl. -65^oC-tól $+150^oC$-ig terjed.

Az áramkörök tényleges belső hőmérséklete a környezeti hőmérséklet és a veszteségek (fogyasztás) függvénye. A legnagyobb megengedhető belső hőmérséklet az adatlapok szerint általában 150^oC. Minél alacsonyabb a környezeti hőmérséklet, annál nagyobb veszteségeket engedhetünk meg az áramkörben anélkül, hogy túllépnénk a 150^oC-os határt. Ezeket a viszonyokat táblázatosan vagy diagram formájában adják meg. A működési hőmérsékleti tartományok enyhe túllépése rendszerint nem okozza az áramkör azonnali meghibásodását, viszont az áramkör jellemzői jelentősen változnak.

Az integrált áramkörök beépítése rendszerint forrasztással történik. Ez rendkívüli hőterhelést jelent az áramkörre. A gyártók rendszerint 250^oC – 300^oC közötti forrasztási hőmérsékletet engedélyeznek *10s* - *60s* időtartamban. Gépi forrasztás esetére definiálják a hőmérsékleti profilt, amit az áramkör el tud viselni.

1.2.8. Tokozás

A gyártó a félvezető lapocskát, amelyben az áramkört integrálta, megfelelő tokozásba építi a kellő mechanikai szilárdság és a megbízható csatlakoztatás érdekében. Sokáig az integrált áramkörök úgynevezett DIL (dual in line – kétsoros) tokozásban láttak napvilágot (1-8a ábra). A szerelés a kivezetések lyukakba illesztésével és forrasztásával történt. Az egy sorban levő lábak egymás közötti távolsága szabványosan *0,1hüvelyk* (*2,54mm*). A tokozás anyaga valamilyen jó szigetelőanyag: műgyanta vagy kerámia.



1-8 ábra: Integrált áramköri tokozások: (a) DIL tokozás, (b) SO típusú SMD tokzás, (c) PLCC típusú SMD tokzás, (d) BGA típusú SMD tokozás.

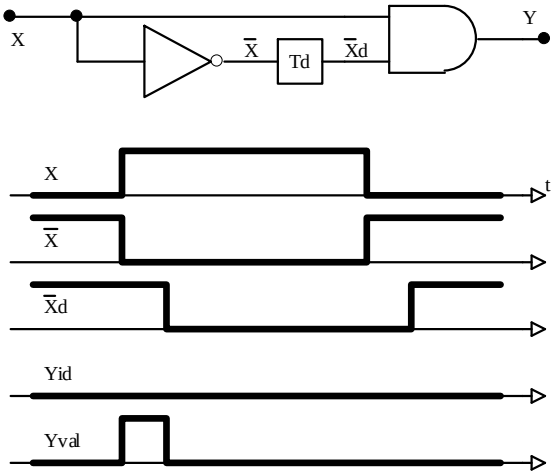
A mind nagyobb integráltsági fok elérése törvényszerűen nagy számú kivezetés létrehozását tette szükségessé. A furatokba történő szerelés nem tette lehetővé az integrált áramkörök lábtávolságának radikális csökkentését. Így jelentek meg a különböző felületszerelt tokozások (*surface mounted device – SMD*). Az *SO* tokozás (1-8b ábra) hasonló a DIL tokozáshoz azzal, hogy a kivezetések sirályszárny alakban vannak meghajlítva, hogy felfekvést biztosítsanak a nyomtatott áramkörre. Kezdetben az *SO* tokozások lábtávolsága *0,05hüvelyk* (*1,27mm*) volt, később tovább csökkentették a méreteket. A PLCC tokozásnál (1-8c ábra) mind a négy oldalon képeztek ki kivezetéseket, ráadásul a lábakat a tokozás alá hajtották a méretek csökkentése érdekében. A *BGA* tokozás (1-8d ábra) a hagyományos értelemben vett kivezetések nélkül készül. A tokozás alsó felén a szigetelőanyaggal egy síkban fém szigetek vannak kialakítva. Forrasztáskor ezeket a szigeteket a nyomtatott áramkörön elhelyezett forraszanyag-gömbökre helyezik, a forrasztás az egész tokozás átmelegítésével történik.

Kezdetben a felületszerelt tokozás drágábbnak számított, maga a beültetés is körülményes volt. 1990 körül az árkülönbség már átbillent a felületszerelt technológia javára, a beültetést viszont szinte kizárólag robotok végzik (az egyedi gyártástól és a javításoktól eltekintve). Az *SMD* alkatrészek alkalmazása lehetővé teszi, hogy a nyomtatott áramkör mindkét oldalára alkatrészeket építsünk és így csökkentsük a berendezés méreteit.

1.3. A késések hatása: hazárdok

Az 1.2.4 pontban tárgyalt késések nem csak egyszerűen késleltetik a kimeneti jelek kialakulását, hanem átmenetileg vagy tartósan az áramkör téves működését okozhatják. Mivel a késésekből eredő tévedések egyrészt kiszámíthatatlanok, másrészt általában káros következményekkel járnak, hazárdjelenségeknek nevezzük őket.

Ha megvizsgáljuk az 1-9 ábrán megadott egyszerű kapcsolást, megállapíthatjuk, hogy ideális esetben a a kimeneten mindig logikai nullát kellene kapnunk, mert az *ÉS* kapu bemeneteire sohasem érkezik egyszerre két egyes az inverter jelenléte miatt. Valós esetben azonban figyelembe kell vennünk az inverter késését, amit a T_d tömb jelképez. A késésből eredően az *X* változó felfutó élét követően rövid időre az *ÉS* kapu mindkét bemenetére logikai egyes érkezik, ami a kimeneten átmenetileg logikai egyest eredményez.



1-9 ábra: Késésből eredő hazárdjelenség egy egyszerű áramkörben.

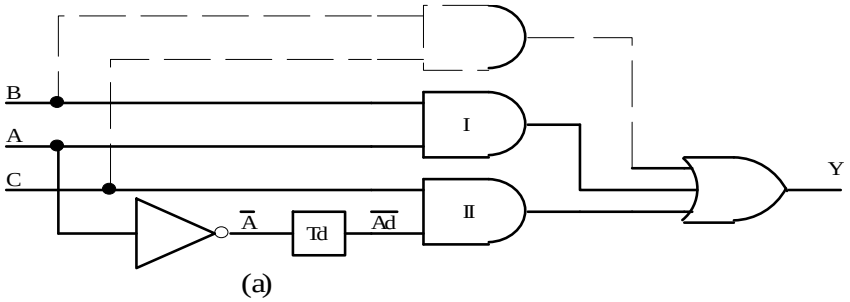
A bemutatott jelenség esetenként hasznos is lehet: ezen a módon egy rövid ideig tartó impulzust tudunk előállítani adott jel felfutó éle közelében, anélkül, hogy passzív differenciáló tagot, vagy valamilyen monostabil áramkört használnánk. A továbbiakban a késésekből eredő

különböző hazárdjelenségeket tanulmányozzuk, és lehetőségeket keresünk a hazárdok kiküszöbölésére.

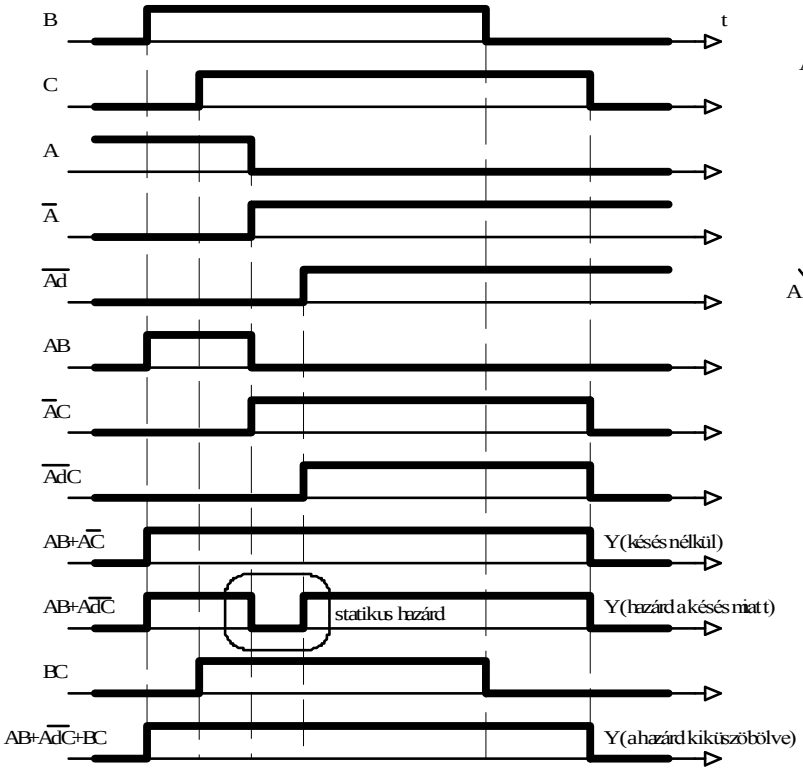
1.3.1. Statikus hazárd

Statikus hazárdnak az olyan jelenségeket nevezzük, amikor az áramkör (a logikai funkcióból megítélve) egy állandó logikai szintet kellene, hogy produkáljon, viszont a feszültség egy rövid időre az ellenkező logikai szintre vált. Az 1-9 ábra esete is statikus hazárd.

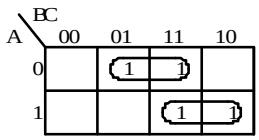
Most elemezzünk egy kissé összetettebb kapcsolást (1-10a ábra) és keressünk megoldást a hazárd ellen. Az áramkör egy logikai szorzatok összegeként felírt függvényt valósít meg kétlépcsős, *ÉS-VAGY* hálózattal. Az egyszerűség kedvéért egyetlen késést, az inverter késését vesszük csak figyelembe.



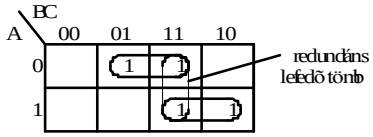
(a)



(b)



(c)



(d)

1-10 ábra: Statikus hazárd és kiküszöbölése: (a) a minimalizált hálózat, (b) idődiagramok, (c) Karnaugh tábla a logikai szorzatok bejelölésével, (d) a kiegészített, hazárdmentes hálózat Karnaugh táblája.

Késés nélkül a *VAGY* kapu bemenetére vagy az egyik vagy a másik *ÉS* kapu felől logikai egyes érkezik és a kimenetet magas logikai szinten tartja (1-10b ábra). A késést figyelembe véve az *AB* szorzat logikai nullára esik még mielőtt a $(\Delta\overline{A})C$ szorzat elérné a logikai egyes értéket, így a

kimeneten átmenetileg logikai nullát érzékelünk. Ez hazárdjelenség, ami a következő fokozatok működésében komoly zavarokat okozhat.

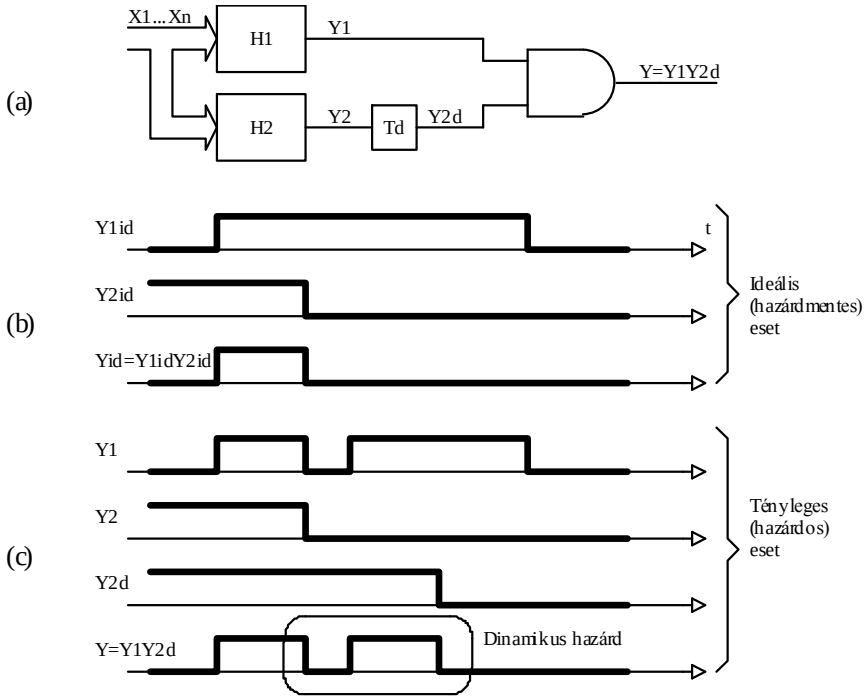
A hazárd kiküszöbölése érdekében tanulmányozzuk a 1-10c ábrán megadott *Karnaugh* táblát. Statikus hazárd azért lép fel, mert a függvényt felépítő prímimplikánsok élszomszédosak és diszjunktak. Ha a bemeneti változók megváltozásakor át kell térni az egyik prímimplikánsból a másikba, megeshet, hogy az nem a szomszédos élen keresztül történik, hanem a tábla egy olyan mezőjén kereszül, amelyen a függvény értéke logikai nulla. Amíg ez az átmenet tart, hazárd jelentkezik.

A hazárd kiküszöbölésének egyik módja az áramkör logikai átalakítása. Ha az átmenet helyét egy redundáns tömbbel (szagatott vonallal jelöltük a 1-10d ábrán) lefedjük, a hazárd megszünik. A redundáns tömböt egy szaggatott vonallal megrajzolt *ÉS* kapuval valósítottuk meg a 1-10a ábrán. Kielemezhetjük, hogy a hozzáadott kapu olyan logikai szorzatot valósít meg, amely a minimalizáció során ki lett hagyva, mert (a hazárdtól eltekintve) nem szükséges a függvény megvalósításához.

Léteznek más módszerek is a statikus hazárd megszüntetésére. Az egyik elgondolás szerint az áramkör egyik ágában jelentkező késést az áramkör másik ágába céltudatosan beépített késéssel kompenzáljuk. A másik igen gyakran alkalmazott módszer az órajellel történő szinkronizáció. A szinkronizáció lehetővé teszi, hogy az áramkör kimenetét csak a hazárd elmúltával vegyük figyelembe.

1.3.2. Dinamikus hazárd

A dinamikus hazárd olyan esetekben jelentkezik, amikor a digitális hálózat kimenetén változik a logikai szint. Ha ez a változás nem szabályosan játszódik le, hanem az új állapot beállta előtt többször is megváltozik a logikai szint, dinamikus hazárdról beszélünk.



1-11 ábra: Dinamikus hazárd keletkezése

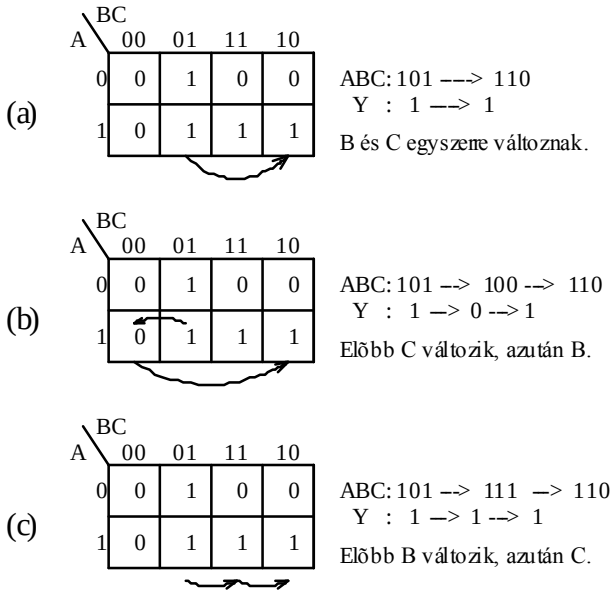
Példaként tekintsük az 1-11a ábrán megadott kapcsolást. A teljes kapcsolás két részhálózatot (*H1* és *H2*) foglal magába. Tételezzük fel, hogy a *H1* hálózatra olyan statikus hazárd jellemző, amelynél a hálózat kimenetén a tartós logikai egyes helyett átmenetileg logikai nulla lép fel. Ha ugyanakkor a *H2* hálózat kimenete T_d idővel késleltetve van, a diagramokon (1-11b ábra)

végigkövethető, hogy az *Y* kimenet nem esik le végérvényesen logikai nullára ugyanakkor, amikor az *Y2* kimenet leesik (ahogyan az az ideális esetben történne), hanem egy *1-0-1-0* logikai értéksorozatot kapunk.

Ez a dinamikus hazárd a digitális berendezés következő fokozataiban komoly tévedéseket okozhat. A dinamikus hazárd a részhálózatban jelentkező statikus hazárd következménye. Ha a statikus hazárdokat kiküszöböljük, a dinamikus hazárd is megszünik.

1.3.3. Funkcionális hazárd

A funkcionális hazárd egy újabb nemkívánatos jelenség a digitális hálózatokban. Akkor jelentkezik, amikor két vagy több bemeneti változó értékváltozása megközlítőleg egyidőben történik meg. Az 1-12a ábrán bemutatott *Karnaugh* táblának megfelelő logikai hálózatnál feltételezzük, hogy adott pillanatban az ABC változók értékkombinációja 101-ről 110-ra változik. Elvileg a B és a C változók logikai értékei egyidőben változnak és ez a kimeneten tartósan logikai egyest kellene, hogy eredményezzen. A valóságban azonban pontosan egy időben történő változások nem jellemzőek.



1-12 ábra: Funkcionális hazárd keletkezése.

Az 1-12b ábrán elemezzük azt az esetet, ahol először a *C* változó esik le logikai nullára, majd kis késsel a *B* változó logikai egyesre ugrik. A *Karnaugh* táblán nyilakkal jelöltük ezt az átmenetet, amelynél a logikai függvény kis időre logikai nulla értéket kap.

Ha a bemeneti változók értéke a fordított sorrendben változik (B változik először, C utána), akkor az 1-12c ábra tanulsága szerint nem lép fel hazárdjelenség.

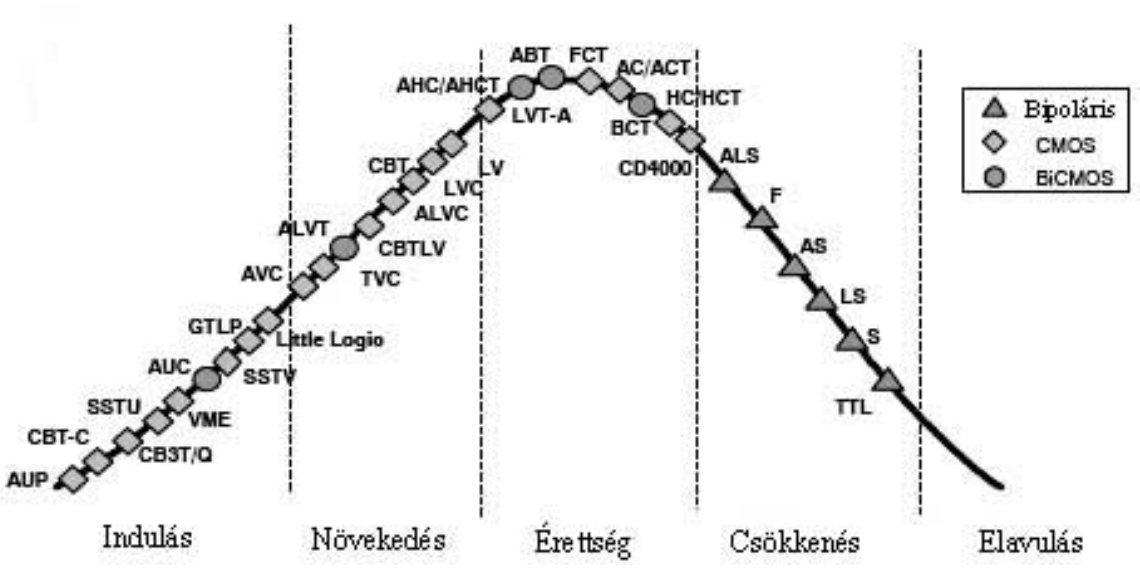
A funkcionális hazárd megszüntetését mekísérelhetjük szándékos késésekkel megoldani, de a rendszerbeli megoldás a bemeneti jelek órajellel történő szinkronizálása.

1.4. Integrált áramköri technológiák

Ugyanazon szerepet ellátó digitális áramkörök megvalósíthatók különböző technológiai eljárásokkal, s így az egyes paraméterek (más jellemzők kisebb-nagyobb kárára) optimalizálhatók. Általában a késések csökkentése a veszteségek növekedéséhez vezet, a tápfeszültség- és vele együtt a fogyasztás csökkentése törvényszerűen rontja a zajtűrést, stb.

1.4.1. Az áramkörcsaládok népszerűsége és életciklusa

Kezdetben a digitális áramkörök kapcsolóelemei bipoláris tranzisztorok voltak. Később jelentek meg a *MOSFET* kapcsolók, melyekkel kedvezőbb átviteli jelleggörbét sikerült elérni, ugyanakkor a fogyasztás is csökkent. Ma a *MOSFET* alapú digitális áramkörök túlsúlyban vannak, de a a bipoláris megoldások sem szorultak ki. Az 1-13 ábrán a *Texas Instruments* által gyártott digitális áramkörcsaládok népszerűségi diagramját láthatjuk. Háromszög jelöli a bipoláris-, négyszög a *CMOS*- és kör a vegyes technológiájú családokat.



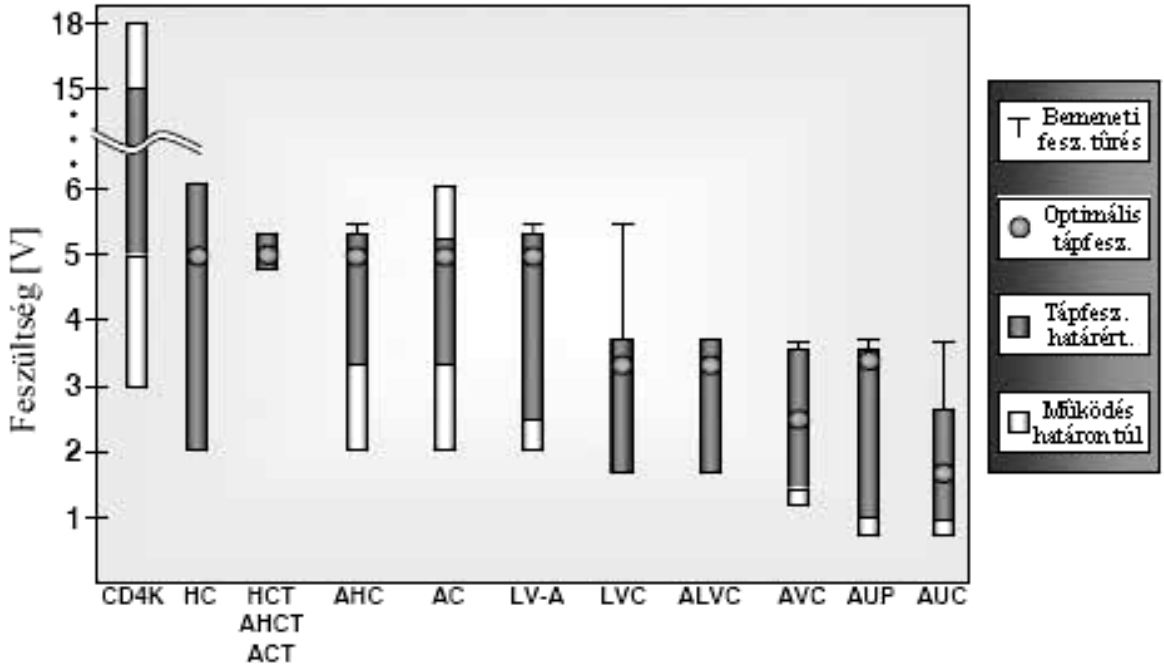
1-13 ábra: A Texas Instruments digitális alkatrészeinek népszerűségi diagramja.

A diagram jobboldali részén azok az áramkörcsaládok vannak feltüntetve (a szokásos betűjelekkel: *TTL, S, LS*...), amelyeket a XX. század hatvanas és hetvenes éveiben kezdtek el gyártani, s mára már népszerűségük csökkent. A diagram csúcsa körül a nyolcvanas években fejlesztett áramkörök vannak, ezek ma az iparban a legelfogadottabb termékek. A bal oldalon az új, ígéretes technológiájú áramkörcsaládokat tüntettük fel, amelyek csak most vannak feltörőben. Az áramkörcsaládok többsége *CMOS* technológiával készül a kisebb fogyasztás és az ideálisabb átviteli jelleggörbe miatt. A bipoláris technológia ma elsősorban ott kap szerepet, ahol nagy árammal történő meghajtásra van igény.

1.4.2. Tápfeszültség szerinti megoszlás

Jellemző, hogy az újabb fejlesztéseknél egyre kisebb tápfeszültséget alkalmaznak, ugyanakkor nem kötelező a tápfeszültséget pontos értéken tartani, bizonyos szélesebb működési tartomány lehetséges. A különböző áramkörcsaládok tápfeszültség szerinti megoszlását az 1-14 ábrán láthatjuk.

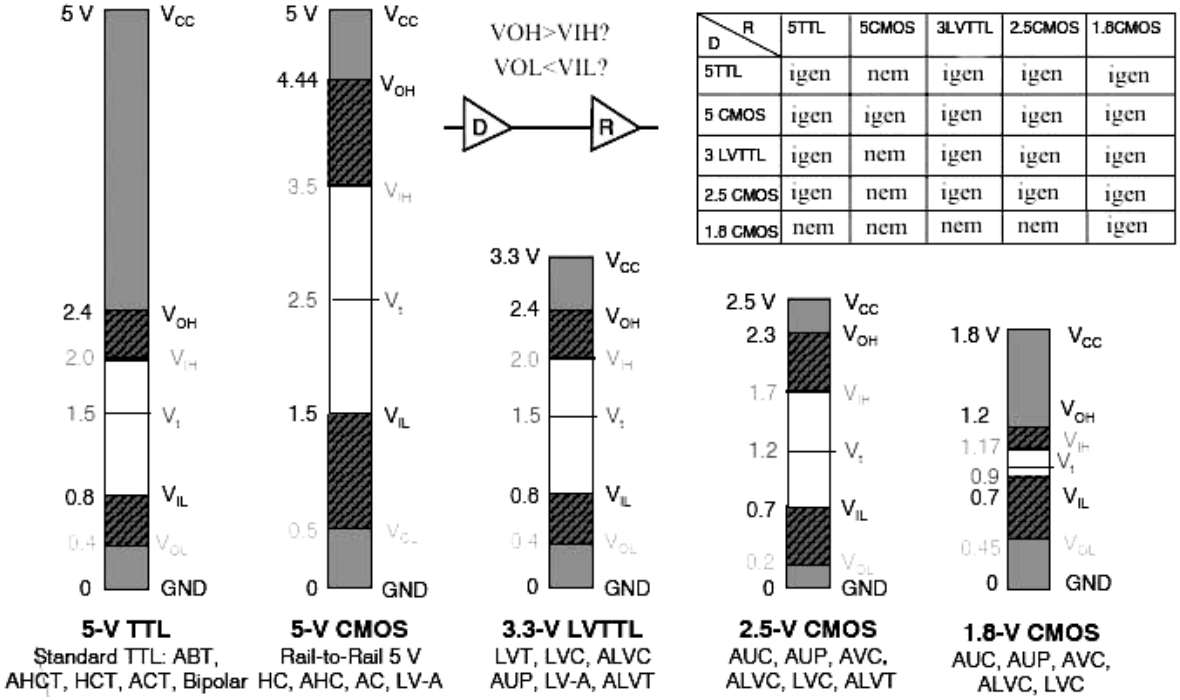
Körrel van jelölve a tápfeszültség névleges értéke adott áramkörcsaládra, amely mellett optimális működés várható. A központi, szürke szakasz az adatlapokon megjelölt működési tartomány, amelyben az áramkör jellemzői a deklarált értékeken belül vannak. A világossal jelölt tartományban az áramkör még működőképes, de a paraméterekben eltérés lehet a deklarált értékektől. A fekete vonallal (T) jelölt határig az áramkör károsodás nélkül eltűri a bemeneti és kimeneti túlfeszültségeket.



1-14 ábra: Az áramkörcsaládok tápfeszültség tartomány szerinti megoszlása.

1.4.3. A logikai szintek kompatibilitása

Az egyes áramkörcsaládokra jellemző logikai szintekről az 1-15 ábra tájékoztat. A régebbi fejlesztésű, 5V-os tápfeszültségű bipoláris áramköröknél a logikai szintek nem szimmetrikusak a 0V-os és az 5V-os szinthez képest, míg a későbbi fejlesztéseknél, különösen a *CMOS* kiviteleknél már jellemző a szimmetria.



D \ R	5TTL	5CMOS	3LVTTL	2.5CMOS	1.8CMOS
5TTL	igen	nem	igen	igen	igen
5 CMOS	igen	igen	igen	igen	igen
3 LVTTL	igen	nem	igen	igen	igen
2.5 CMOS	igen	nem	igen	igen	igen
1.8 CMOS	nem	nem	nem	nem	igen

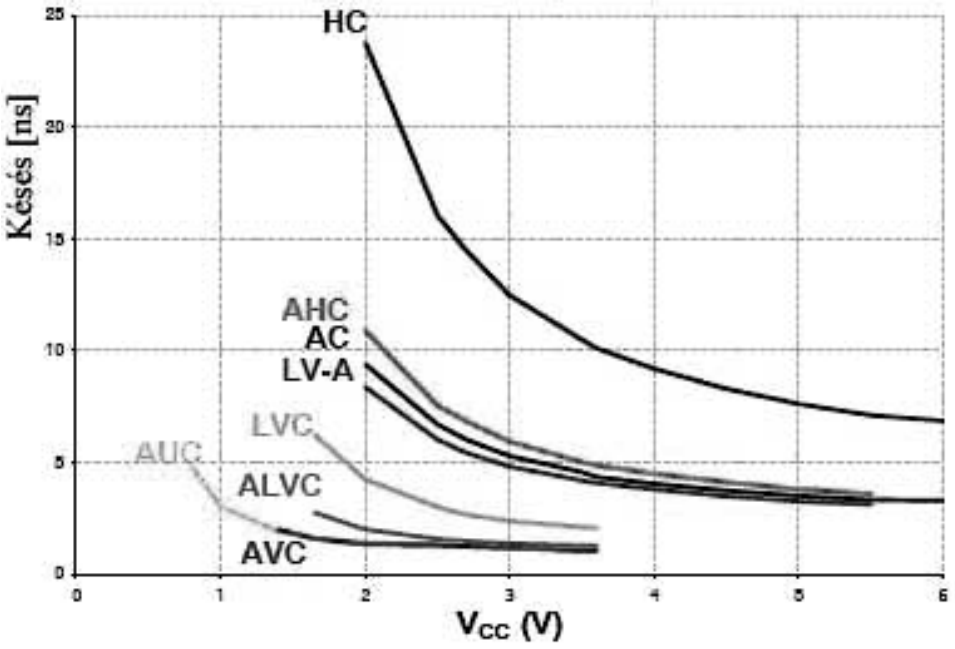
1-15 ábra: Az egyes áramkörcsaládok logikai szintjei és családok közötti kompatibilitás.

Az ábra jobb felső sarkában közölt táblázat az egyes áramkörcsaládok közötti kompatibilitást mutatja. Egy családon belül az áramkörök logikai szint szempontjából mindig kompatibilisek. Két család áramkörei csak akkor tudnak együttműködni, ha érvényesek a korábban tárgyalt $V_{OL}<V_{IL}$ és $V_{OH}>V_{IH}$ feltételek. Megtörténik, hogy a keresett logikai funkciót nem találjuk meg az egyébként alkalmazott áramkörcsaládban, ilyenkor másik családból vagyunk kénytelenek választani és felmerül a kompatibilitás kérdése. A kompatibilitás elérése céljából alkalmazhatunk különleges illesztő áramköröket is.

A táblázatban olyan családokat is kompatibiliseknek jelöltünk be, amelyeknél a meghajtó áramkör (*D*) kimeneti magas logikai szintje (V_{OH}) nagyobb, mint a meghajtott áramkör (*R*) tápfeszültsége (V_{CC}). Ilyen kötést csak akkor alkalmazhatunk, ha a gyártó engedélyezi a tápfeszültségnél nagyobb feszültséget a bemeneten az illető áramkörcsaládra.

1.4.4. A késések függése a tápfeszültségtől

A fejlesztés iránya évtizedeken keresztül a mind kisebb tápfeszültség a veszteségek és a késések csökkentése végett. Meg kell azonban jegyezni, hogy azok az áramkörök, amelyeket szélesebb tápfeszültség tartományra terveztek kevesebb késést mutatnak, ha nagyobb feszültséggel tápláljuk őket (1-16 ábra). Értelemszerűen minden egyes családra a diagramot az adatlapon deklarált tápfeszültség tartományra rajzoltuk fel.

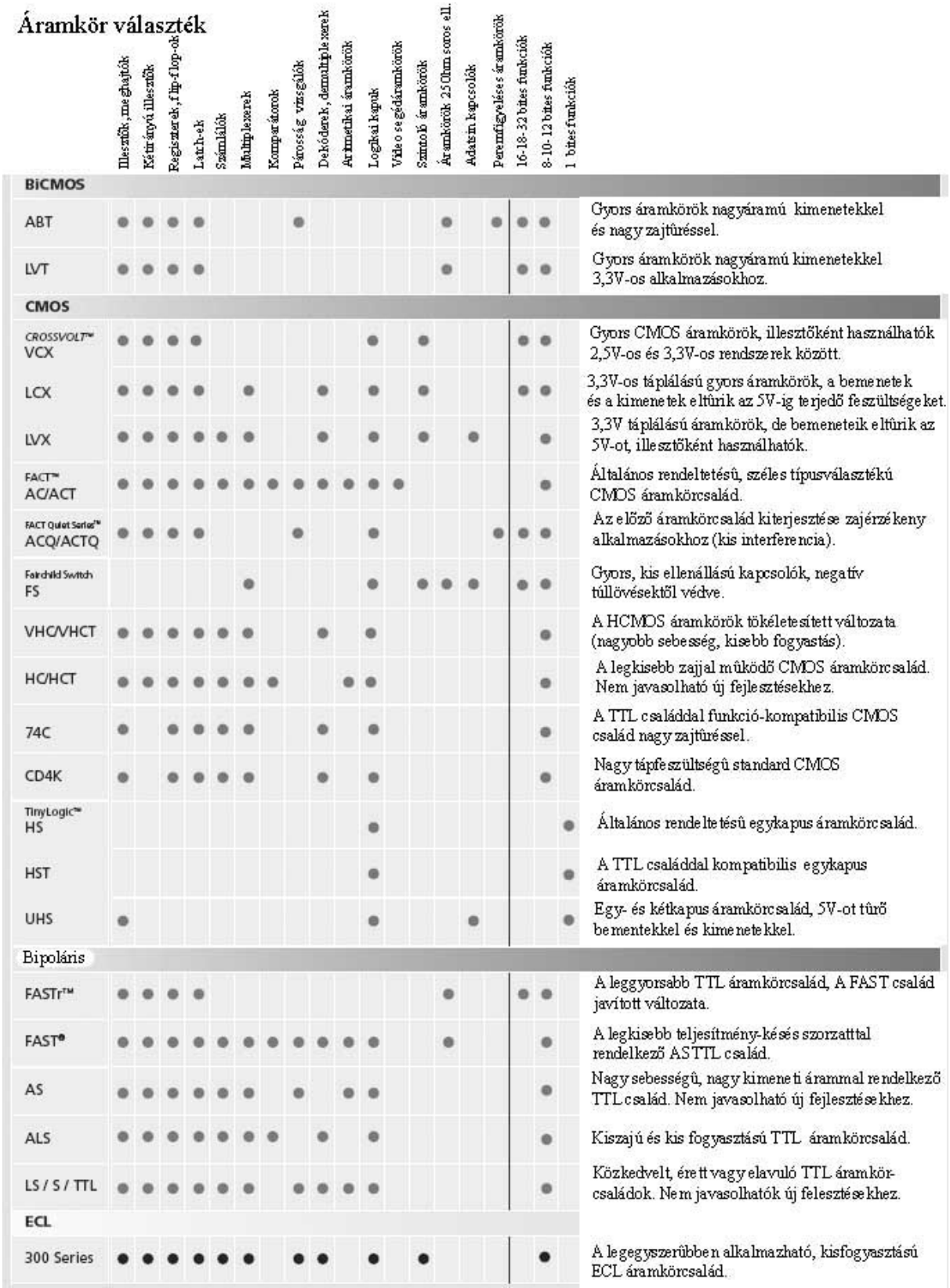


1-16 ábra: A késések függése az alkalmazott tápfeszültségtől különböző áramkörcsaládokra.

1.4.5. Az áramkörcsaládokban fellelhető logikai funkciók

Az 1-17 ábrán közölt táblázat példaként a *Fairchild* cég egyes áramkörcsaládjaiban fellelhető logikai funkciókról ad rövid összefoglalót. Részletesebb tájékoztatást a konkrét funkciókról a gyártók terméklistáiban találhatunk. A táblázat jobb oldalán némi utalást találunk az alkalmazási területekre és meggondolásokra vonatkozóan. A bipoláris technológiát illetően legteljesebb a választék a *TTL*, *S* , *LS* és *F* jelzésű áramkörökből. A *CMOS* családok között legteljesebb az érettnek számító *AC* és *ACT* sorozat.

Az itt közölt termékválaszték csak *SSI* és *MSI* áramköröket említ, de a felsorolt gyártástechnológiákat alkalmazzák a szoftveres és hardveres programozású *LSI* és *VLSI* áramköröknél is



1-17 ábra: Logikai funkciók választéka az egyes technológiákban gyártott áramkörcsaládokban.

II. Digitális tervezés
SSI és MSI funkcionális egységekkel

2. Kombinációs hálózatok

Kombinációs hálózat alatt olyan digitális áramköröket értünk, amelyeknél a kimenet(ek)en megjelenő logikai értékek kizárólag a pillanatnyilag érvényes bementi logikai állapotokkal vannak meghatározva. Természetesen a bemenet(ek) megváltozásakor el kell telnie bizonyos időnek (késések, 1.3 szakasz), amíg az érvényes kimeneti logikai szintek kialakulnak, de ezen az időn túl az áramkör kimeneti logikai szintjei már nem függenek a korábbi bemeneti értékektől. Ezt azért hangsúlyozzuk ki, mert a 3. fejezetben tárgyalásra kerülő sorrendi hálózatok viselkedése eltér ettől.

A kombinációs hálózatok itt tárgyalt fajtái a XX. század hatvanas évei óta mint SSI és MSI funkcionális egységek kerülnek forgalomba. Alkalmazásuk azonban nem korlátozódik a digitális berendezések SSI és MSI áramkörökből való építésére. A mikroprocesszorok és mikrovezérlők valamint a *PLD*-k belső szerkezetében úgyszintén találkozunk ezekkel a kapcsolásokkal. A *PLD*-k programozására szolgáló hardverleíró nyelvekben is megtalálhatjuk ezeknek a funkcióknak a megfogalmazását.

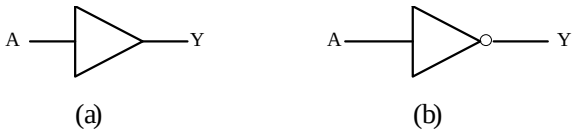
2.1. Illesztő áramkörök

A különböző funkcionális egységek összekapcsolása gyakran nem közvetlenül történik, hanem bizonyos illesztő áramkörökkel. Ezek az áramkörök végezhetik a logikai szintek változtatását, impedanciaillesztést, ugyanakkor a jeleket átengedhetik direkt vagy invertált formában. Jórészt itt nem is logikai funkciókról van szó, csak bizonyos erősítő áramkörökről, amelyeket kapcsolóüzemben használunk. Az illesztő áramkörökből általában nagyobb számút (pl. nyolcat) építenek be egy *DIL* vagy *SO* tokozásba és az *SSI* kategóriába sorolhatók.

2.1.1. Neminvertáló és invertáló illesztők

A neminvertáló illesztőket (angolul: *buffer*) és az invertáló illesztőket (inverter) úgy képezik ki, hogy nagy legyen a bemeneti ellenállásuk és kicsi legyen a kimeneti ellenállásuk. Ezzel biztosítható, hogy a bemenet nem terheli az előző funkcionális egységet, ugyanakkor a kimenet nagyszámú következő fokozattal terhelhető, anélkül, hogy a logikai szintek torzulnának. A megfelelő rajzjeleket a 2-1 ábrán láthatjuk.

2-1 ábra: Az illesztő áramkörök rajzjelei: a) neminvertáló illesztő, b) invertáló illesztő.

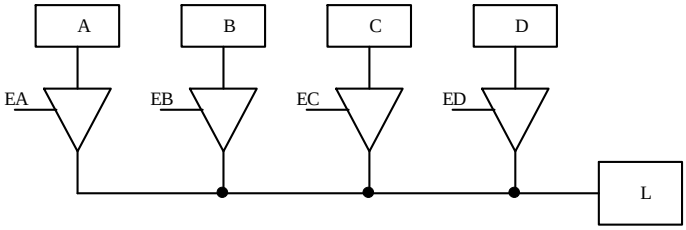


Egyszerűbb esetekben az illesztők egy tápfeszültséget igényelnek. Különleges kivitelnél, amikor az illesztő két különböző tápfeszültségen működő logikai egység között teremt kapcsolatot, az áramkör bemeneti és kimeneti oldalát külön tápforrásra kötjük.

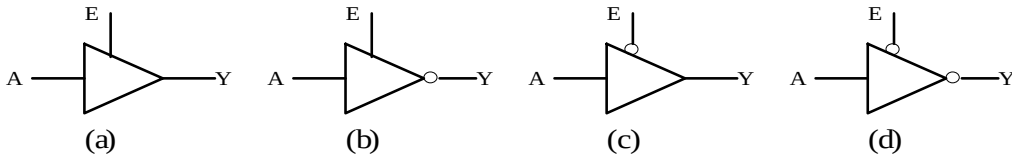
2.1.2. Háromállapotú illesztők

Ha a 2-2 ábrán bemutatott módon több forrásból (*A, B, C, D*) kell jeleket közvetíteni egy közös vezetéken egy felhasználó (L) felé, háromállapotú illesztőket alkalmazunk. A háromállapotú illesztők közül egyszerre csak egyet aktivizálunk, míg a többit a harmadik (nagyimpedanciás állapotban tartjuk. Az aktivizálást az E_A, E_B, E_C, E_D engedélyező jelek (angolul: *enable*) segítségével végezzük.

2-2 ábra: Jelek továbbítása közös vezetéken háromállapotú illesztők segítségével.



A háromállapotú illesztők közül négy elvi megoldás van forgalomban (2-3 ábra). Az illesztők működhetnek a továbbított jel invertálása nélkül (a és c ábra), vagy invertálásával (b és d ábra) ugyanakkor az engedélyezés történhet logikai egyessel (a és b ábra) vagy logikai nullával (c és d ábra). A háromállapotú illesztőkből is nagyobb számút szoktak beépíteni egy közös tokozásba.

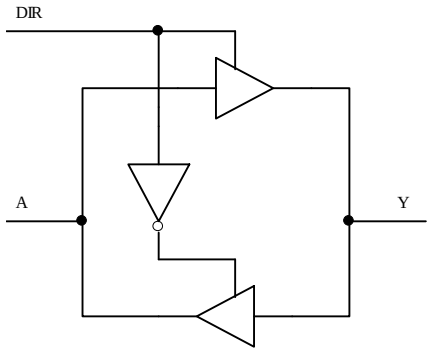


2-3 ábra: Különböző háromállapotú illesztők.

2.1.3. Kétirányú illesztők

Ha a különböző funkcionális egységek között kétirányú jelátvitelt kell megvalósítani, azt kétirányú illesztőkkel tehetjük meg. Egy logikai jel átvitelére alkalmas kétirányú illesztő belső szerkezetét a 2-4 ábrán láthatjuk. Az irányt meghatározó jel (*DIR*) belső invertálásának köszönhetően hol az egyik, hol a másik illesztő áramkör működik, viszi át a jelet adott irányba. A pillanatnyilag kikapcsolt illesztő a nagyimpedanciás állapotnak köszönhetően nincs kihatással a logikai szintre.

A kétirányú illesztőkből szintén nagyobb számút szoktak integrálni egy tokozásban. Gyártanak olyan változatot is, amely különböző tápfeszültségen és különböző logikai szintekkel működő hálózatok között teremt kapcsolatot.



2-4 ábra: Kétirányú illesztő belső szerkezete.

2.2. Dekóder

A dekóderek több bementű és több kimenetű kombinációs hálózatok, amelyeknél minden egyes bemeneti variációra külön kimenet aktivizálódik.

2.2.1. Teljes dekóder

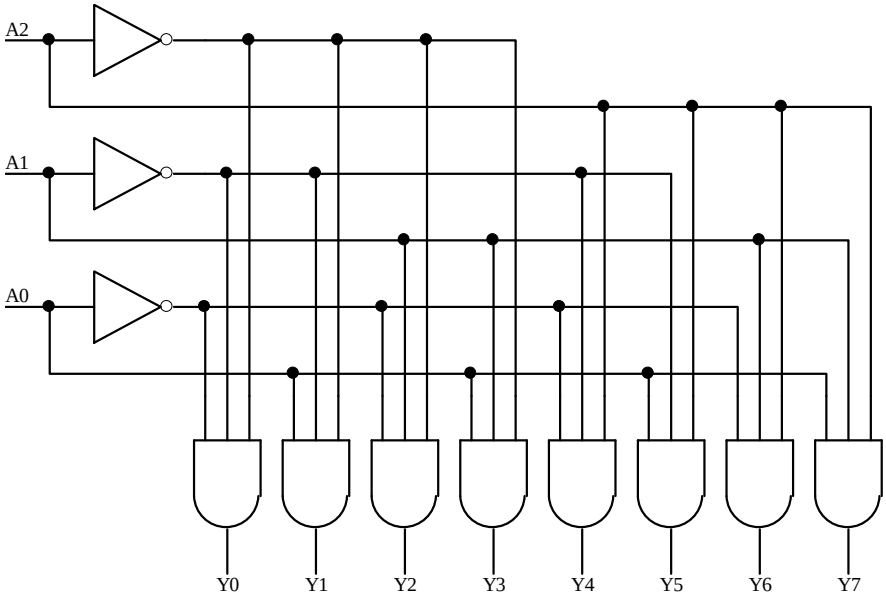
A teljes dekóder bemenetei binárisan kódolt számok. Mivel n bemeneti vonalon a logikai szintek variációinak száma 2^n, a teljes dekóder kimeneti vonalainak száma 2^n. Adott bemeneti variációra egy és csak egy kimeneti vonal aktivizálódik (jelenik meg logikai egyes). A három bemenetű, nyolc kimenetű (3/8-as) teljes dekóder kombinációs táblázatát a 2-1 táblázatban láthatjuk.

A2	A1	A0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

$$Y0 = \overline{A2} \cdot \overline{A1} \cdot \overline{A0}$$
$$Y1 = \overline{A2} \cdot \overline{A1} \cdot A0$$
$$Y2 = \overline{A2} \cdot A1 \cdot \overline{A0}$$
$$Y3 = \overline{A2} \cdot A1 \cdot A0$$
$$Y4 = A2 \cdot \overline{A1} \cdot \overline{A0}$$
$$Y5 = A2 \cdot \overline{A1} \cdot A0$$
$$Y6 = A2 \cdot A1 \cdot \overline{A0}$$
$$Y7 = A2 \cdot A1 \cdot A0$$

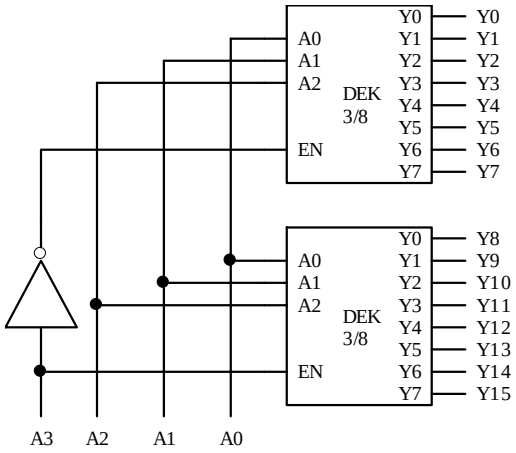
2-1 táblázat: A 3/8-as dekóder kombinációs táblázata és logikai függvényei.

Az *Y0...Y7* kimeneti logikai függvények csak egy logikai szorzatot tartalmaznak, így a dekóder megvalósítására kellő számú bemenettel rendelkező *ÉS* kapukat használunk (2-5 ábra).



2-5 ábra: A teljes dekóder megvalósítása ÉS kapukkal.

Gyakran a dekódolási célokra gyártott integrált áramkörök az *A, B, C* bemenetek mellett még egy engedélyező bementettel (*EN – enable*) is rendelkeznek. Ezzel a dekóder kapacitása növelhető: két *3/8*-as dekóder összekapcsolásával építhető *4/16*-os dekóder stb. (2-6 ábra).



2-6 ábra: 4/16-os dekóder megvalósítása két darab 3/8-as dekóder felhasználásával.

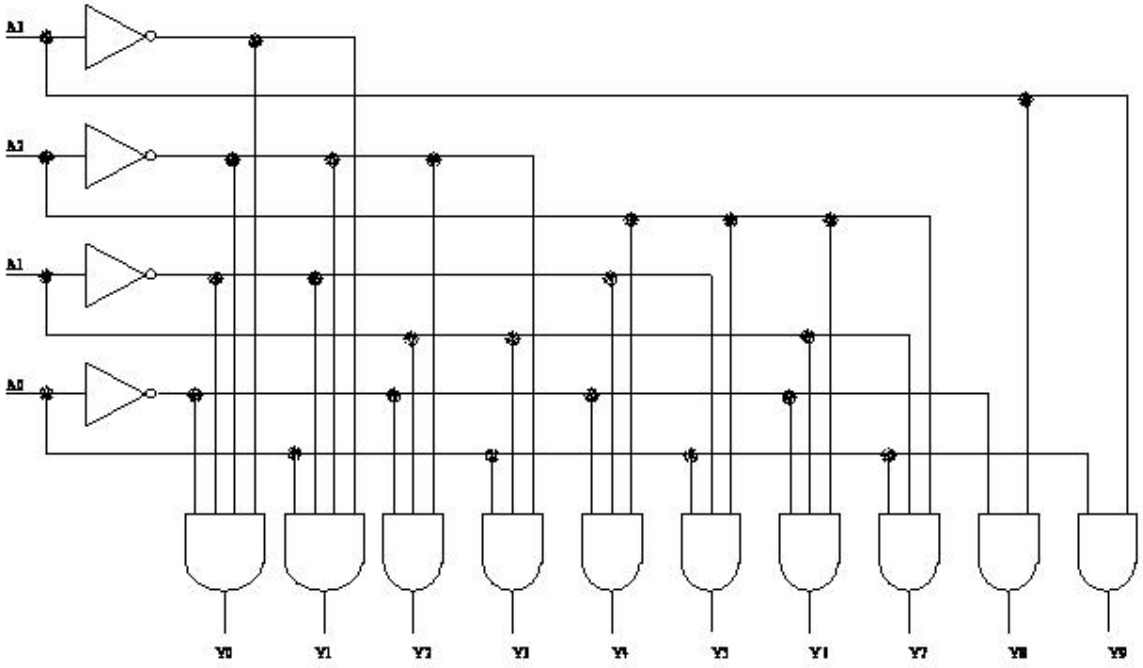
2.2.2. Nem teljes dekóder

Ha tudjuk, hogy bizonyos bemeneti variációk sohasem fognak megjelenni a dekóder bemenetén, a teljes dekóder leegyszerűsíthető, elsősorban úgy, hogy kihagyjuk azokat az *ÉS* kapukat, amelyek olyan logikai szorzatokat képeznek, amelyek értéke sohasem lehet egyes. Az így kapott nem teljes dekóder helyesen fog működni, de szerkezete nem minimális.

A nem teljes dekóderek tipikus példája a *BCD* dekóder, amely binárisan kódolt decimális számjegyek dekódolását végzi. A *BCD* dekóder kombinációs táblázatát a *4/16*-os kóder táblázatából kapjuk, az utolsó hat sor elhagyásával, mivel a *10...15* számok kódjanak megjelenésére nem számítunk (2-2 táblázat). Az így definiált *Y0...Y9* logikai függvények minimalizációja a 2-7 ábrán megrajzolt logikai hálózatot adja.

A3	A2	A1	A0	Y9	Y8	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	1	0	0	0	0	0	0
0	1	1	1	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0	0	0	0	0	0	0

2-2 táblázat: A BCD dekóder kombinációs táblázata.



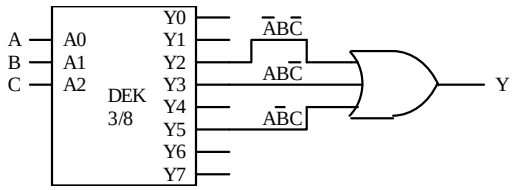
2-7 ábra: A BCD dekódert megvalósító minimalizált logikai hálózat.

2.2.3. Logikai függvények megvalósítása dekóderrel

Mivel a dekóderek a bemeneti változók logikai szorzatait hozzák létre, *VAGY* kapu hozzáadásával a szorzatok összegeként felírt logikai függvények minden további nélkül megvalósíthatók. Az:

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

függvény például a 2-8 ábrán bemutatott módon valósítható meg.



2-8 ábra: Logikai függvény megvalósítása dekóder felhasználásával.

2.3. Kóder

A digitális jelek feldolgozása kódolt formában történik. Pl. a számítógép egy billentyűjét lenyomva egy logikai nullákból és egyesekből álló számsor (kód) állítódik elő. A kódolás lényege, hogy minden bemeneti értékhez másik kód tartozik.

2.3.1. Teljes kóder

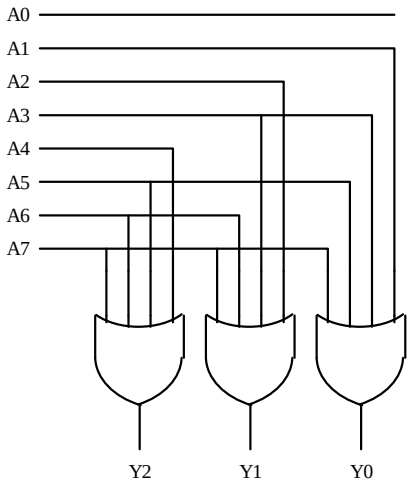
A teljes kóder 2^n bemeneti változó n bit hosszúságú kódját alakítja ki. A 2-3 táblázatban a nyolc bemenetű és három kimenetű (8/3-as) kóder kombinációs táblázatát adtuk meg. A táblázatban a lehetséges $2^8=256$ sorból csak azt a nyolc sort tüntettük fel, amelyekben egyszerre csak egy bemeneti változó értéke logikai egyes. Az ábrán adtuk meg a minimalizált logikai függvényeknek megfelelő kifejezéseket is. A kóder függvényeinek felírása nem szorzatok összegeként célszerű, hanem összeg formájában.

A7	A6	A5	A4	A3	A2	A1	A0	Y2	Y1	Y0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Y0=A1+A3+A5+A7
Y1=A2+A3+A6+A7
Y2=A4+A5+A6+A7

2-3 táblázat: A 8/3-as kóder kombinációs táblázatai és a kimenetek logikai függvényei.

Az egyenletek alapján a 8/3-as kóderre a 2-9 ábrán bemutatott kapcsolást kapjuk. Természetesen ez a logikai hálózat nem használható kódokat ad, ha a hálózat bemenetén egyszerre több mint egy logikai egyes jelenik meg. Erre az esetre a kódert kiegészíthetjük egy áramkörrel, amely a nemkívánatos esetre figyelmeztet.



2-9 ábra: A 8/3-as teljes kódert megvalósító logikai hálózat.

2.3.2. Nem teljes kóder

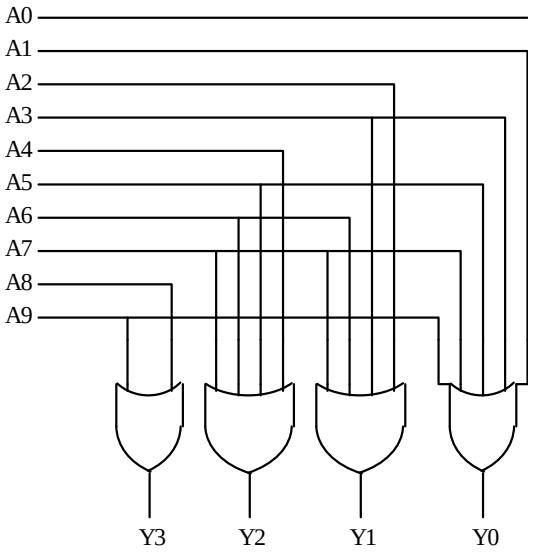
A kóder nem teljes, ha az n kimeneti vonallal kevesebb mint 2^n bemeneti vonal állapotát kódoljuk. Tipikus eset, amikor a decimális számrendszer tíz számjegyét kívánjuk négy kimeneti vonalra (bit) kódolni. Ezt a kódert *DC-BCD* kódernek nevezzük. A megfelelő kombinációs táblázatot a 2-4 táblázatban láthatjuk, ugyanott adtuk meg a minimalizált logikai függvényeket.

Ai	Y3	Y2	Y1	Y0
0	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

Y0=A1+A3+A5+A7+A9
Y1=A2+A3+A6+A7
Y2=A4+A5+A6+A7
Y3=A8+A9

2-4 táblázat: DC/BCD kóder kombinációs táblázata és logikai függvényei.

A DC/BCD kódert megvalósító logikai hálózatot a 2-10 ábrán adtuk meg. Itt is hansúlyozni szeretnénk, hogy a felhasználónak kell gondoskodni arról, hogy a kóder bemenetén ne jelenjen meg egyszerre több mint egy logikai egyes.



2-10 ábra: DC/BCD kódert megvalósító hálózat.

2.3.3. Prioritásos kóder

Említettük, hogy az eddig ismertetett kóderek téves illetve nem használható kódot adnak, ha a bemeneti vonalakon egyszerre több mint egy logikai egyes jelenik meg. Tipikus példája ennek a mikrovezérlők megszakításainak kezelése. Mivel a megszakító jelek aszinkron módon érkeznek a külső áramkörből, várható, hogy egyszerre több jel logikai egyesen fog lenni. Ilyenkor a mikrovezérlőnek el kell dönteni, hogy melyik megszakító jelre fog reagálni. Ezt a feladatot úgynevezett prioritásos kóderrel lehet kezelni.

A prioritásos kóder úgy tekinti, hogy a bemenetek fontossági (prioritási) sorrendbe vannak állítva. Ha egyszerre több logikai egyes jelenik meg, annak a bementnek a kódját kell kialakítani, amelyik a legnagyobb prioritású.

A 2-5 táblázatban a 8/3-as prioritásos kóder kombinációs táblázatát adtuk meg. *X*-szel jelöltük azokat a bementi értékeket, amelyek nincsennek kihatással a prioritásos kóder működésére. Meg kell említeni, hogy a táblázat felírásakor nem vettük figyelembe azt az esetet, amikor minden bemenet logikai nullán van. Ha ez szükséges (jelezni kell ezt az esetet), arra egy külön áramkört kell szerkeszteni.

Bemenet	Kimenet (kód)
1xxxxxxx	000
01xxxxxx	001
001xxxxx	010
0001xxxx	011
00001xxx	100
000001xx	101
0000001x	110
00000001	111

2-5 táblázat: 8/3-as prioritásos kóder kombinációs táblázata.

2.4. Kódátalakító

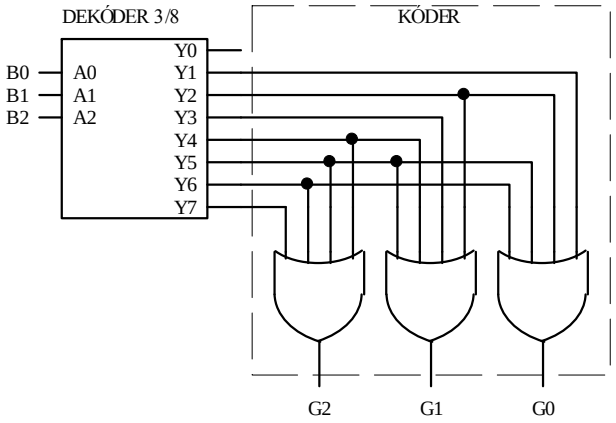
Az olyan kombinációs hálózatokat, amelyek a digitális információt egyik kódrendszerből a másikba alakítják, kódátalakítóknak nevezzük. Elvileg minden kódátalakító megépíthető egy dekóder és egy kóder kaszkád kötésével, tény viszont, hogy nagy esély van a hálózat minimalizálására.

2.4.1. Bináris/Gray kódátalakító

Három bites természetes bináris kód *Gray* kóddá való átalakítását a 2-6 táblázat szerint kell végezni. A dekóder és kóder kaszkád kötésével kapott megoldást a 2-11 ábra mutatja.

Természetes bináris kód	Gray féle kód
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

2-6 táblázat: Természetes bináris kód és Gra-féley kód párhuzamos bemutatása három bites kódok esetére.



2-11 ábra: Kódátalakító dekóder és kóder kaszkád kötésével.

A táblázat alapján a Gray kód egyes bitjeire a következő egyenletek írhatók fel:

$$G_2 = B_2\overline{B_1}\,\overline{B_0} + B_2\overline{B_1}B_0 + B_2B_1\overline{B_0} + B_2B_1B_0$$

$$G_1 = \overline{B_2}B_1\overline{B_0} + \overline{B_2}B_1B_0 + B_2\overline{B_1}\,\overline{B_0} + B_2\overline{B_1}B_0$$

$$G_0 = \overline{B_2}\,\overline{B_1}B_0 + \overline{B_2}B_1\overline{B_0} + B_2\overline{B_1}B_0 + B_2B_1\overline{B_0}$$

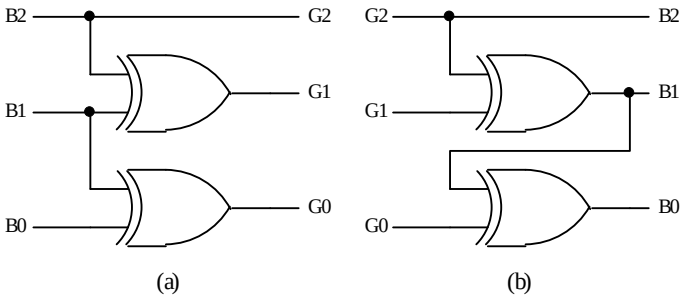
A logikai függvények minimalizációjával a következő, sokkal egyszerűbb kifejezésekhez jutunk:

$$G_2 = B_2$$

$$G_1 = B_2 \oplus B_1$$

$$G_0 = B_1 \oplus B_0$$

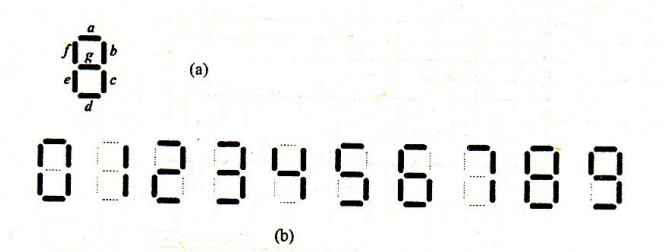
A kódátalakító egyszerűsített rajzát a 2-12a ábrán láthatjuk. A 2-12b ábrán bemutatott, hasonlóképpen minimalizált hálózat, a fordított műveletet végzi.



2-12 ábra: Minimalizált kódátalakítók: (a) bináris kódból Gray kódba és (b) Gray kódból bináris kódba.

2.4.2. BCD/7 szegmenses kódátalakító

A leggyakrabban alkalmazott kódátalakító a binárisan kódolt decimális számjegyek megjelenítését végzi hétszegmenses kijelzőn. A gyártói adatlapokon ezt a kódátalakítót gyakran (de tévesen) dekódernek nevezik. Az egy decimális számjegy megjelenítésére szolgáló LCD vagy LED kijelző szegmensei a közismert nyolcas alakban vannak elrendezve. A szegmensek betűjeleit a 2-13a ábrán láthatjuk. A 2-13b ábrán mutattuk be, hogy melyik szegmenseket kell aktivizálni az egyes számjegyek megjelenítéséhez.



2-13 ábra: A hétszegmenses kijelző szegmenseinek jelölése (a), és a decimális számok megjelenítése hét szegmenssel (b).

A kódátalakítónak megfelelő kombinációs táblázatot a 2-7 táblázatban láthatjuk.

Számjegy	D C B A	a b c d e f g
0	0 0 0 0	1 1 1 1 1 1 0
1	0 0 0 1	0 1 1 0 0 0 0
2	0 0 1 0	1 1 0 1 1 0 1
3	0 0 1 1	1 1 1 1 0 0 1
4	0 1 0 0	0 1 1 0 0 1 1
5	0 1 0 1	1 0 1 1 0 1 1
6	0 1 1 0	1 0 1 1 1 1 1
7	0 1 1 1	1 1 1 0 0 0 0
8	1 0 0 0	1 1 1 1 1 1 1
9	1 0 0 1	1 1 1 1 0 1 1

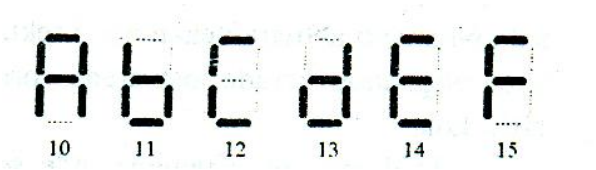
2-7 táblázat: BCD/7 szegmenses kódátalakító kombinációs táblázata.

A hét szegmens mindegyikére fel kell írni a megfelelő egyenletet. Minimalizáció után a következő eredményre jutunk:

$$a = B + D + AC + \overline{A}\,\overline{C}$$
$$b = \overline{C} + AB + \overline{A}\,\overline{B}$$
$$c = A + \overline{B} + C$$
$$d = D + \overline{A}B + A\overline{B}C + \overline{A}\,\overline{C} + B\overline{C}$$
$$e = \overline{A}B + \overline{A}\,\overline{C}$$
$$f = D + \overline{A}\,\overline{B} + \overline{A}C + \overline{B}C$$
$$g = D + \overline{A}B + B\overline{C} + \overline{B}C$$

Az így kapott logikai függvényeket kétlépcsős *ÉS-VAGY* logikai hálózattal valósíthatjuk meg. Az egész hálózatot rendszerint egy *MSI* áramkörbe integrálják. Ezek az áramkörök tartalmaznak még egy vezérlőbemenetet, amellyel minden szegmens egyszerre kikapcsoható. A vezérlőbemenet a többszámjegyes kijelzők időmultiplexben történő működtetésénél tesz nagy szolgálatot.

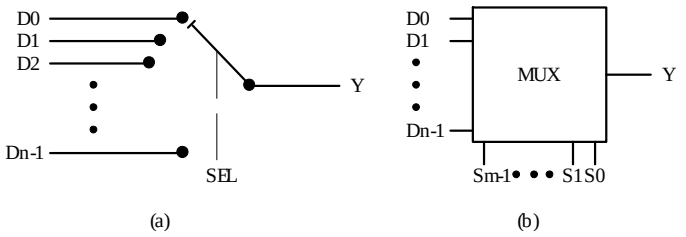
A fenti minimalizált megoldás értelmetlen kijelzést ad, ha a bemenetre nemlétező kódot vezetünk. Némi változtatással megoldható, hogy a hexadecimális számrendszer 10-től 15-ig terjedő számkódjainál a kijelző a 2-14 ábrán bemutatott szimbólumokat írja ki.



2-14 ábra: A hexadecimális számrendszer kilenc feletti számjegyeinek kijelzése hétszegmenses kijelzővel.

2.5. Multiplexer

A multiplexer egy többállású kapcsoló szerepét látja el (2-15 ábra). Digitális jeleket (D_0...D_{n-1}) továbbít a megfelelő számú bemenet egyikétől a kimenet felé. A továbbítandó bemeneti jel megválasztása a multiplexer választó bementeivel (S_0...S_{m-1}) történik.



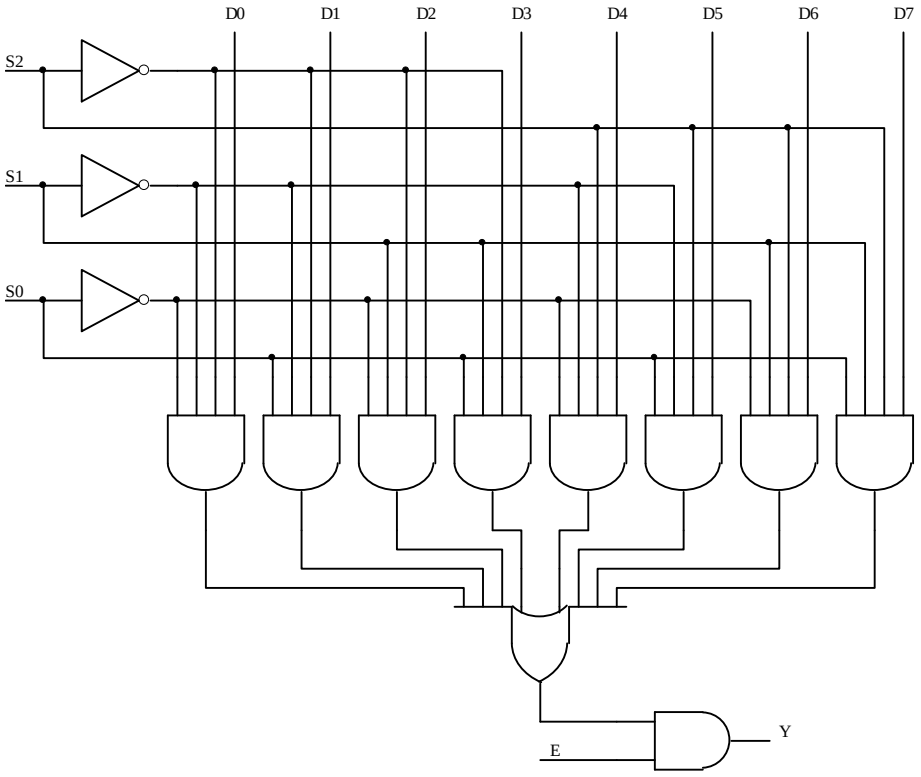
2-15 ábra: A multiplexer értelmezése (a) és rajzjele (b).

2.5.1. Digitális multiplexer szerkesztése

A gyártók rendszerint *2, 4, 8* és *16* bementű multiplexereket integrálnak egy MSI áramkörbe. A választóbementek száma rendre 1, 2, 3 és 4. A nyolc adatbemenettel (D0...D7) és három választóbemenettel (S0...S2) rendelkező multiplexer kimeneti logikai függvénye a következő:

$$Y = D_0 \overline{S}_2 \overline{S}_1 \overline{S}_0 + D_1 \overline{S}_2 \overline{S}_1 S_0 + ... + D_7 S_2 S_1 S_0 .$$

Az áramkör megvalósítását nyolc darab négybemenetű *ÉS* kapuval és egy nyolcbementű *VAGY* kapuval végzik. A 2-16 ábrán bemutatott logikai hálózat ezt a multiplexert valósítja meg, azzal, hogy létezik egy engedélyező bemenet (*E*) is, amellyel az áramkör kimenete logikai nulla állapotba hozható, függetlenül az adatbementektől és a választóbemenetektől. Az engedélyező bemenet a multiplexer bővítésére használható ki. Ez akkor jelentős, ha pl. tizenhatnál több adatvonalat szeretnénk átképezni egy kimeneti adatvonalra és nem áll rendelkezésünkre megfelelő számú bemenettel rendelkező integrált multiplexer.

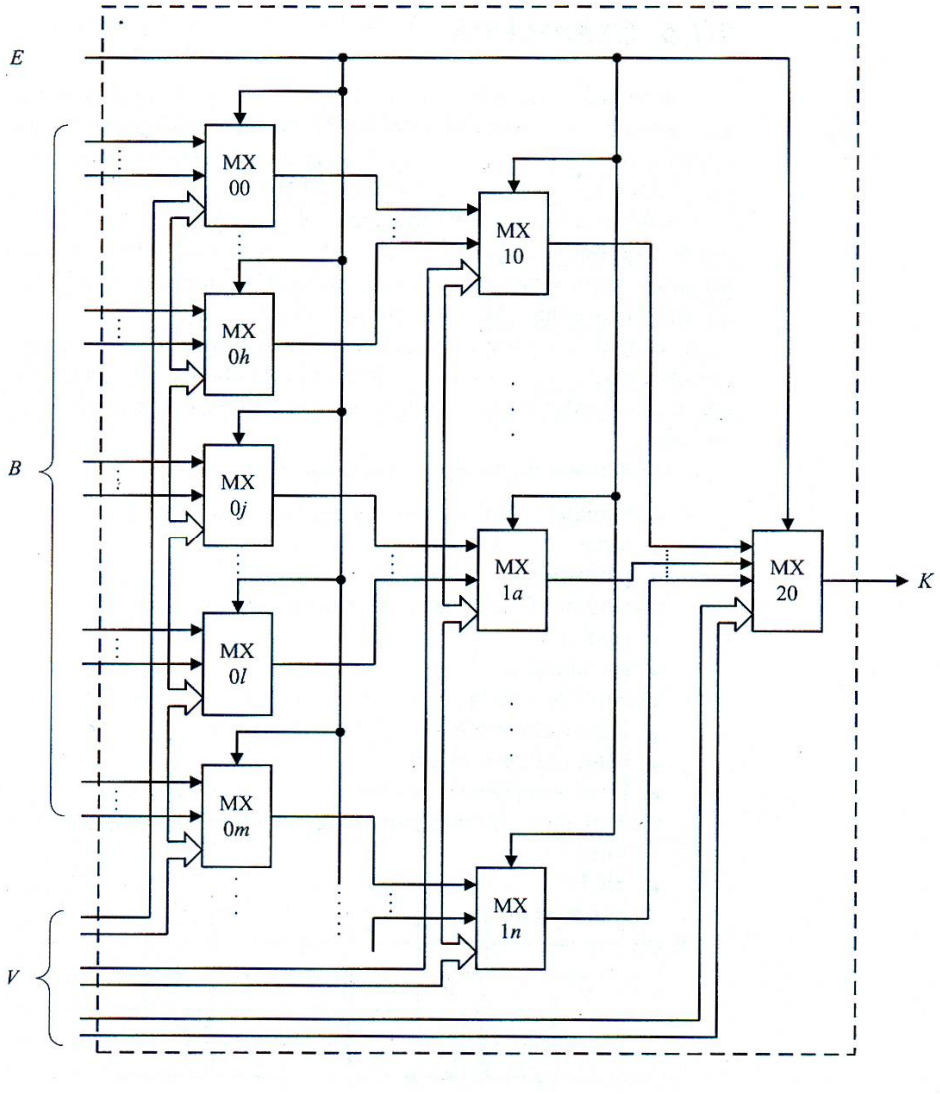


2-16 ábra: Nyolc adatbemenettel rendelkező multiplexert megvalósító logikai hálózat.

2.5.2. Multiplexer bővítése

A bővítés módját a 2-17 ábrán mutatjuk be. Az adatokat megfelelő számú kisebb multiplexer bemeneteire vezetjük. Ezeknek a multiplexereknek a választó bemeneteire ugyanazokat a jeleket vezetjük. Az első fokozat multiplexereinek kimenő adatait újabb multiplexer-

re vezetjük, így döntjük el, hogy az előző fokozat kimenetei közül melyik adatot továbbítjuk. A bővítés történhet több fokozatban is.



2-17 ábra: Multiplexer bővítésének elvi vázlata.

2.5.3. Logikai függvények megvalósítása multiplexerrel

Az alapfunkció mellett (adatok választása) a multiplexerek hatékonyan alkalmazhatók logikai függvények megvalósítására is. A megvalósítás feltétele, hogy a függvény szorzatok összegeként legyen kifejezve és a szorzatok tartalmazzák minden bemeneti változót. Ha például az

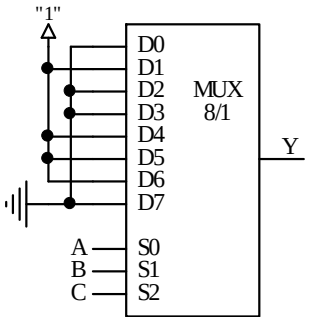
$$Y = \overline{C}\,\overline{B}A + C\overline{B} + CB\overline{A}$$

függvényt szeretnénk megvalósítani, először létrehozzuk a normál alakot:

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

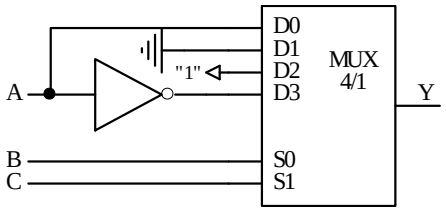
A megvalósítás úgy történik, hogy az A, B, C változókat a választó bemenetekre kötjük, az adatbemenekre pedig vagy logikai egyesnek vagy logikai nullának megfelelő feszültségszintet

hozunk, attól függően, hogy az illető adatbemenetnek megfelelő logikai szorzat szerepel-e a függvényben (2-18 ábra).



2-18 ábra: Háromváltozós logikai függvény megvalósítása nyolc adatbemenettel rendelkező multiplexerrel.

A multiplexer valamennyivel hatékonyabban is használható logikai függvények megvalósítására, mint azt az előző példában láttuk. Nyolc adatbenettel rendelkező multiplexerrel megoldható négyváltozós logikai függvény vagy az előző példában szereplő háromváltozós logikai függvény megvalósítható mindössze négy adatbemenettel rendelkező multiplexerrel (2-19 ábra).

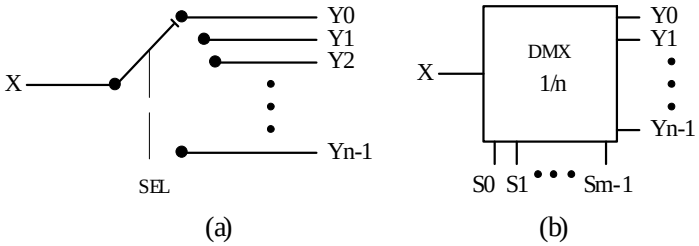


2-19 ábra: Háromváltozós logikai függvény megvalósítása négy adatbemenettel rendelkező multiplexerrel.

A függvény *B* és *C* változóit a szokott módon a választó bementekre kötöttük. Az egyes adatbementekre akkor kötünk logikai nullát vagy egyest, ha az *A* változótól függetlenül az *B* és *C* változók adott értékeinél a függvény értéke nulla vagy egyes. Ha a függvény értéke az A változótól is függ, az illető bemenetekre az *A* változó direkt vagy invertált értékét kötjük, úgy hogy a multiplexer kimenete a helyes függvényértéket adja.

2.6. Demultiplexer

A demultiplexer a multiplexerrel ellentétes szerepet tölt be: egy adatvonalról több kimeneti adatvonal egyike felé továbbítja az információt, az alapelv egy többállású kapcsolóval szemléltethető (2-20a ábra). A kimeneti adatvonal kiválasztása úgy történik, hogy megfelelő kódot vezetünk a választó bemenetekre (2-20b ábra, $S_0 ... S_{m-1}$ bemenetek).

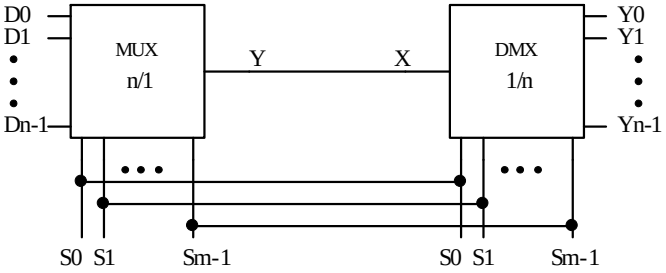


2-20 ábra: Demultiplexer szerepének szemléltetése többállású kapcsolóval (a) és a demultiplexer rajzjele.

A demultiplexer dekóderrel helyettesíthető, ha a dekóder engedélyező bementére vezetjük a bemeneti adatot. A dekóder bemeneteit választó bemenetként használjuk. A kimeneten egyes jelenik meg, ha az alkatrész engedélyezve van az engedélyező változóként bekötött adat által, ilyenkor a logikai egyes átkerül a bemenetről a választott kimenetre. Ha az adat pillanatnyilag logikai nullának felel meg, a kimeneten, mivel az alkatrész nincs engedélyezve logikai nulla jelenik meg. Így tehát mindkét esetben a helyes érték kerül a kimeneti csatornára. A gyártók általában ugyanazokat az alkatrészeket kínálják mint dekódert és mint demultiplexert.

2.6.1. Több adat átvitele közös csatornán

Multiplexer és demultiplexer összekapcsolásával digitális információkat vihetünk át csökkentett számú vezetéken. Ha a bemeneti adatvonalak száma $n=2^m$, az adatátvitel ilyen módon megoldható mindössze $m+1$ vezetékkel. A megfelelő kapcsolási rajzot a 2-21 ábrán láthatjuk.. A választó bemeneteket azonos módon kötöttük be mindkét áramkörnél, így a bemeneti és kimeneti adatvonalak sorrendje azonos.



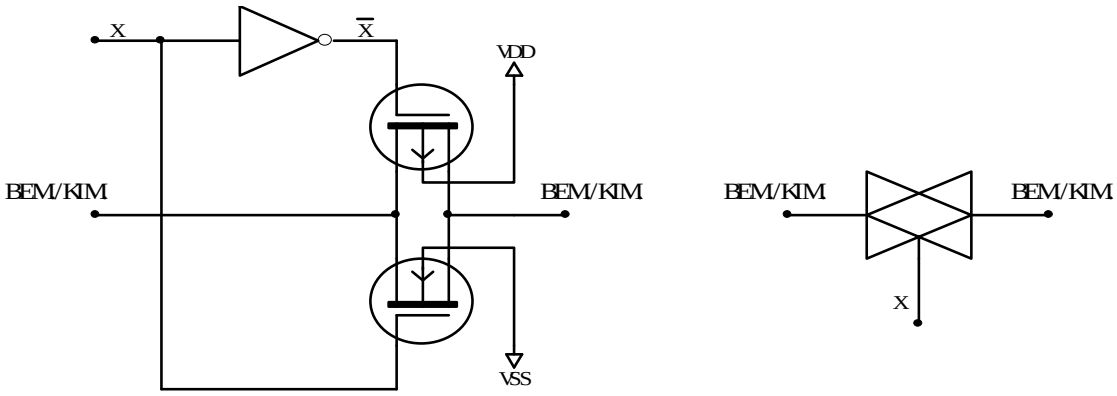
2-21 ábra: Adatátvitel csökkentett számú vezetéken multiplexer és demultiplexer felhasználásával.

A módszer hiányossága, hogy az adatátvitel az egyes bemeneti csatornákról az egyes kimeneti csatornákra nem egyidőben, hanem egymás után történik. Ezt nevezik idő-multiplexnek. A közös átviteli csatorna kapacitása megoszlik az egyes adatvonalak között így lényegesen korlátozódik az átvitel sebessége.

2.6.2. Analóg multiplexer/demultiplexer

A digitális multiplexerek és demultiplexerek csak logikai szintek átvitelére alkalmasak, ellentétben a 2-15 és 2-20 ábrákon bemutatott többállású kapcsolóval, amely mindkét irányba vihet át digitális és analóg jeleket. Analóg jelek multiplexálására/demultiplexálására pl. többcsatornás mérőberendezéseknél van szükség.

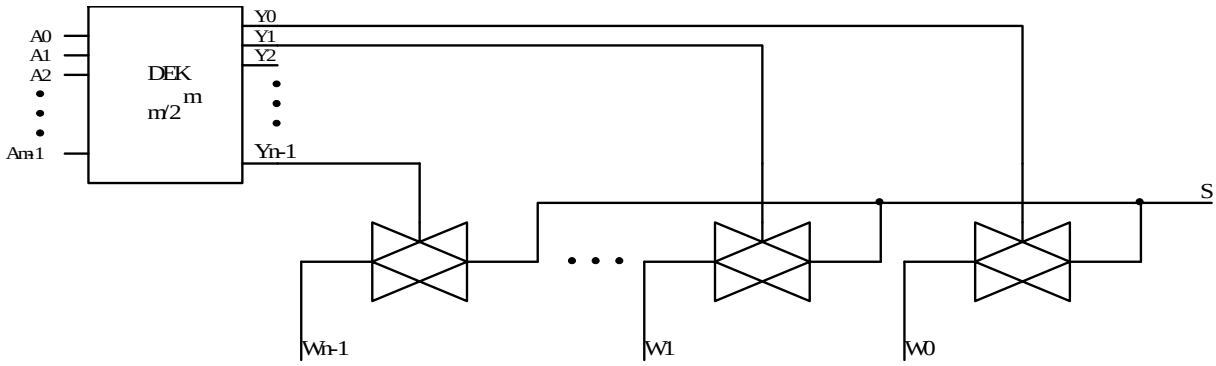
Az integrált technikában is megoldható az analóg jelek multiplexálása/demultiplexálása. Az alapot az úgynevezett analóg kapcsoló képezi, amely vezérlőjel hatására egy félvezető csatornát nyit illetve zár. Az analóg kapcsolót általában egy N és egy P csatornás MOSFET összekapcsolásával kapják (2-22 ábra), de léteznek más megoldások is. A lényeg, hogy bekapcsolt állapotban a csatorna ellenállása nagyon kicsi legyen, kikapcsolt állapotban pedig nagyon nagy.



2-22 ábra: Analóg kapcsoló felépítése N és P csatornás MOSFET-tel (a) és a kapcsoló rajzjele (b).

Kellő számú analóg kapcsolót felsorakoztatva és dekóderrel vezérelve kapjuk az analóg multiplexert/demultiplexert (2-23 ábra). A kapcsolók egyik oldalára hozzuk a megfelelő analóg jeleket, a kapcsolók másik végét viszont egy pontba kötjük össze és a kimeneti vonalra csatlakoztatjuk. A dekóder gondoskodik róla, hogy a vezérlőjelek bármely variációja esetén mindig egy és csak egy analóg kapcsoló kapjon vezérlést. Ilyen módon valósul meg az analóg multiplexer.

Ugyanez az áramkör használható demultiplexerként is, mivel az analóg kapcsoló mindkét irányba átviszi a jelet, amikor be van kapcsolva és egyik irányban sem vezet, amikor ki van kapcsolva.



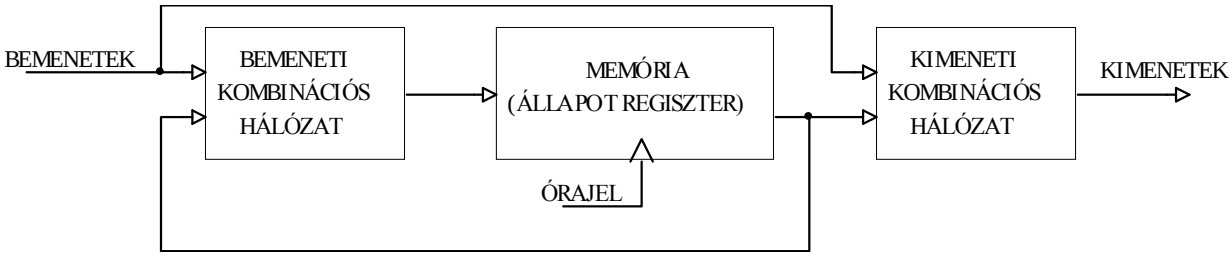
2-23 ábra: Analóg multiplexer/demultiplexer megvalósítása analóg kapcsolókkal és dekóderrel.

3. Sorrendi hálózatok

A sorrendi (szekvenciális) hálózatok olyan digitális áramkörök, amelyek emlékezetnek (memória) nevezhető tulajdonsággal rendelkeznek. Bizonyos információkat az áramkör a korábban alkalmazott vezérlőjelekkel kapcsolatosan tárol. Az áramkör kimenetén kapott jelek részben a tárolt információtól (a korábbi vezérlőjelektől) függenek, részben a pillanatnyilag érvényes bemeneti logikai szintektől. A sorrendi hálózatokat nevezik logikai automatáknak is, mivel gyakran automatikus vezérlési célokra alkalmazzák őket.

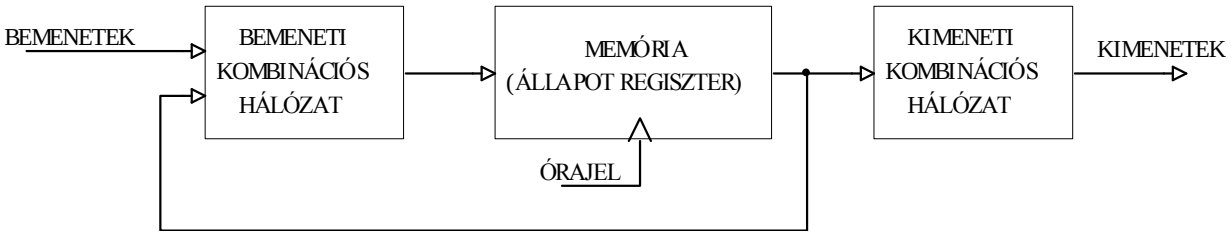
A sorrendi hálózatok információ tároló képessége általában kicsi, mindössze néhány bitről van szó. Az n bit memóriával rendelkező hálózatnál 2^n állapotot különböztethetünk meg. A véges számú állapot miatt a sorrendi hálózatokat nevezik véges állapotú automatáknak (angolul: *finite-state machine*) is.

A memória mellett a sorrendi hálózatok kombinációs elemeket (hálózatot) is tartalmaznak. A belső szerkezetet tekintve két fajta logikai automatát különböztetünk meg. A *Mealy* féle automata tömbvázlatát a 3-1 ábrán mutatjuk be. A memória mellett ez a szerkezet két kombinációs hálózatot is tartalmaz: egyet a bemeneten, egyet a kimeneten. A bemeneti hálózat a pillanatnyi bemeneti logikai szintek és a memóriában tárolt állapot alapján eldönti, hogy mi legyen az automata új állapota, ez alapján beírást végez a memóriába. Ugyanakkor a kimeneti logikai hálózat a pillanatnyilag érvényes bemeneti jelek és a tárolt információ alapján definiálja a kimeneti digitális jeleket.



3-1 ábra: Mealy féle sorrendi hálózat tömbvázlata.

Némileg egyszerűbb a *Moore* féle automata (3-2 ábra). Itt a kimeneti kombinációs hálózat csak a tárolt állapot alapján képezi a kimeneti logikai szinteket. A *Moore* féle megoldásnak egy egyszerűsített esete az, amikor kimeneti kombinációs hálózatot nem használunk, maguk a memóriaelemek kimenetei (mind vagy csak némelyik) képezik a kimeneti változókat.



3-2 ábra: Moore féle sorrendi hálózat tömbvázlata.

A sorrendi hálózatok szerkesztésénél nem élvez abszolút előnyt sem a *Mealy* féle- sem a *Moore* féle automata. Az előző megoldás általában gyorsabb, az utóbbi egyszerűbb.

A sorrendi hálózatok többségénél az állapotváltozás pillanatát órajellel szinkronizáljuk. Az órajel rendszerint periódikusan ismétlődő négyszögimpulzusokból áll, habár a periódikusság nem feltétel. Az órajelet csak a memória állapotváltozásainak időzítésére használjuk, a kombinációs hálózatok szinkronizáció nélkül működnek. Aszinkron sorrendi hálózatok építhetők, de működésük

sokkal kevésbé áttekinthető, mint a szinkron hálózatoké, ezért tervezésük sokkal nagyobb körültekintést igényel.

Ebben a fejezetben először a memóriaelemekkel foglakozunk, azután tekintjük át a sorrendi hálózatok leírásának és szerkesztésének módjait. A végén az SSI illetve MSI áramkörként beszerezhető, gyakran használatos sorrendi hálózatokat tekintjük át.

Az itt bevezetett fogalmak és módszerek nem csak az SSI és MSI áramkörökből történő hagyományos tervezésnél használatosak, hanem a programozható logikai áramkörökkel történő tervezésnél is.

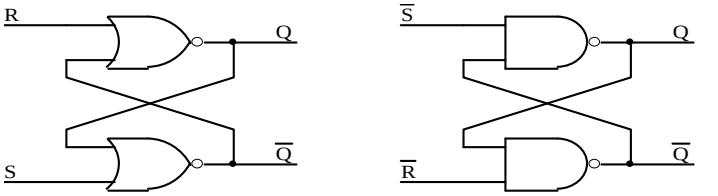
3.1. Elemi memóriák

A sorrendi hálózatok memóriáját *latch*-ek és *flip-flop*-ok alkotják Ezek az áramkörök egy bit információ tárolására alkalmasak. Maguk is elemi sorrendi hálózatoknak tekinthetők. Az információ rögzítését visszacsatolás útján végzik. A beírt információt addig jegyzik meg, amíg tápfeszültség alatt vannak. Egyes memória elemeknél szintvezérlést, másoknál élvezérlést alkalmazunk.

3.1.1. *Latch*-ek

A szintvezérléssel működő egybites memória áramköröket *latch*-eknek nevezzük. A két alapáramkört, amelyek ilyen szerepet képesek ellátni, logikai kapuk keresztbe csatolásával kapjuk és *SR latch*-eknek nevezzük őket (3-3 ábra).

A 3-3a ábrán a *NEM-VAGY* kapukból megépített *SR latch* kapcsolási rajza látható. A keresztcsatolásnak köszönhetően az áramkörnek két stabil állapota van: az egyik esetben a kimenetek $Q=1$, $\overline{Q}=0$ logikai szintekre állnak be, ezt hívjuk szetelt állapotnak, ha a kimeneteken a fordított a helyzet, a reszetelt állpotról van szó. Hasonló a helyzet a *NEM-ÉS* kapukból megépített *SR latch*-nél (4-3b ábra). Az áramkörök viselkedését az *S* (szet) és *R* (reszet) bemenetektől függően a megfelelő áramkörök alatti táblázatokban foglaltuk össze.



S	R	Q_{n+1}	$\overline{Q_{n+1}}$	
0	0	Q_n	$\overline{Q_n}$	
0	1	0	1	
1	0	1	0	
1	1	0	0	TILOS

S	R	Q_{n+1}	$\overline{Q_{n+1}}$	
0	0	1	1	TILOS
0	1	0	1	
1	0	1	0	
1	1	Q_n	$\overline{Q_n}$	

3-3 ábra: SR latch áramkörök keresztbe csatolt logikai kapukból: (a) NEM-VAGY kapukkal, (b) NEM-ÉS kapukkal.

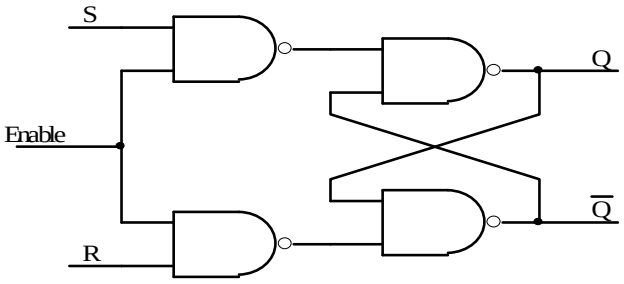
A *NEM-VAGY* kapukból megépített *SR latch-et* az *S* bemenetre hozott logikai egyessel szeteljük és az *R* bemenetre hozott logikai egyessel reszeteljük. Időközben a nem használt bemenet logikai nullán kell hogy legyen. Ha egyik műveletet sem kívánjuk végezni, mindkét bementet logikai nullán kell tartani, ilyenkor az áramkör tartja a korábban beállított állapotot.

A táblázatban nem megengedett (TILOS) esetnek neveztük azt, ha mindkét bemenetre logikai egyest hozunk. Ilyen esetben mindkét kimenet átmenetileg logikai nullán van, de nem lehet biztosan tudni, melyik állapot fog kialakulni miután a bemenetek visszaesnek nullára. Ha a visszaesés nem egyidőben történik, akkor az a bemenet fog érvényesülni, amely később esett visza, ha viszont egyszerre történik a visszaesés, akkor a kapuk késései közötti különbségek fognak

dönteni. Az ilyen bizonytalanságokat a digitális hálózatokban kerüljük, ezért neveztük ezt a bemeneti kombinációt nem megengedettnek.

A *NEM-ÉS* kapukkal megépített *SR latch*-et alacsony logikai szintekkel vezéreljük. Itt az egyszerre megjelenő bemeneti nullák okoznak bizonytalanságot, ezért itt ez az eset kerülendő.

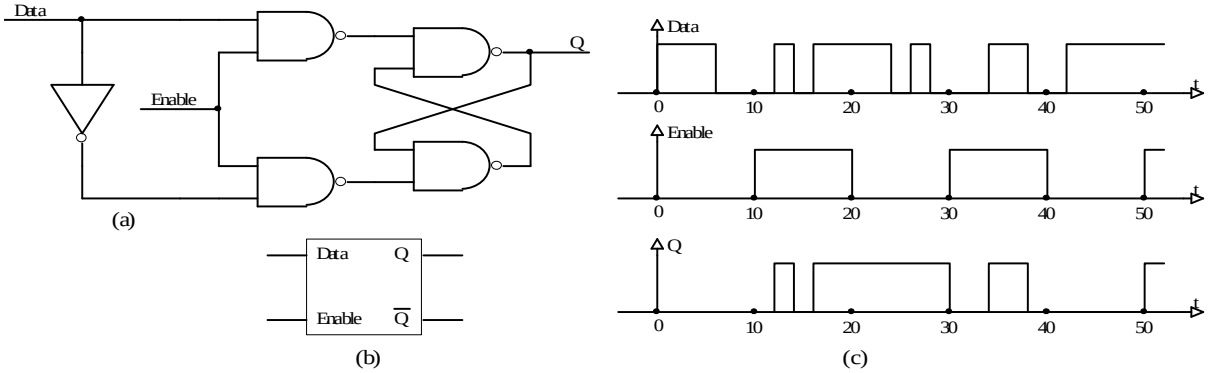
Az *SR latch* állapotváltozásai szükség szerint egy vezérlőjellel (*Enable*) szinkronizálhatók a 3-4 ábrán bemutatott módon. Amíg a vezérlőjel (engedélyező jel) logikai nullán van, a bemeneti *NEM-ÉS* kapuk kimenetei logikai egyesen vannak, a *latch*-be beírt érték nem változhat. Amint az *Enable* jel magas logikai szintre emelkedik, az *ÉS* kapuk átengedik a bemeneti *S* és *R* jeleket, amelyek hatására megtörténik a megfelelő állapotváltozás.



3-4 ábra: Engedélyező (Enable) jellel szinkronizált SR latch.

Ha az *S* és *R* bemeneteken változás történik, ez mindaddig kihatással van a *latch* állapotára, amíg az *Enable* jel magas szinten van. Az ilyen fajta szinkronizációt szintvezérlésnek nevezzük. Az így megvalósított áramkört transzparens *latch*-nek nevezzük, mivel az áramkör mintegy "átlátszó" a szet és reszet jelek számára, amíg a vezérlőjel aktív.

A bemeneti jelek nem megengedett kombinációja elkerülhető, ha a korábbi áramkörhöz még egy logikai invertert adunk (3-5a ábra). Az inverter biztosítja, hogy a bemeneteken nem jelenik meg egyszerre szet és reszet jel. Ezt az áramkört *D latch*-nek nevezzük mivel egy adat (*data*) bemenete van. A *D latch* szokásos rajzjelét a 3-5b ábrán láthatjuk, működési módját viszont a 3-5c ábrán megadott diagramok szemléltetik. A transzparens jelleg erre a kapcsolásra is érvényes. Az engedélyező jel hiányában a *Data* bemenet nincs kihatással a kimenetre, ellenkező esetben a bemenet kis késéssel átíródik a kimenetre. Az engedélyező jel lefutó élénél reteszelődik (rögzítődik) a pillanatnyilag érvényes állapot.

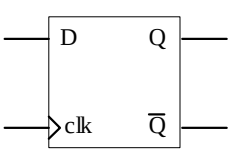


3-5 ábra: Szinkronizált D latch szerkezete (a), rajzjele (b) és a működését szemléltető diagramok (c).

3.1.2. *Flip-flop*-ok

A *flip-flop*-ok élvezérelt elemi memóriák. Az adat beírása a vezérlőjel felfutó vagy lefutó élénél történik, a konkrét megvalósítástól függően. A beírandó adatot a bemenet illetve a bemenetek határozzák meg. Az adatbemeneteket illetően többféle *flip-flop* van használatban. A *D latch*-hez némileg hasonlító *D flip-flop* rajzjelét, a működését leíró táblázatot és a jeldiagramokat a 3-6 ábrán adtuk meg. A működés lényeg, hogy a *D* változó értékét a kapcsolás csak az órajel (*clk*) felfutó élénél (a diagramon ezt szaggatott vonallal jelöltük) veszi figyelembe és írja át a *Q* kimenetre. A

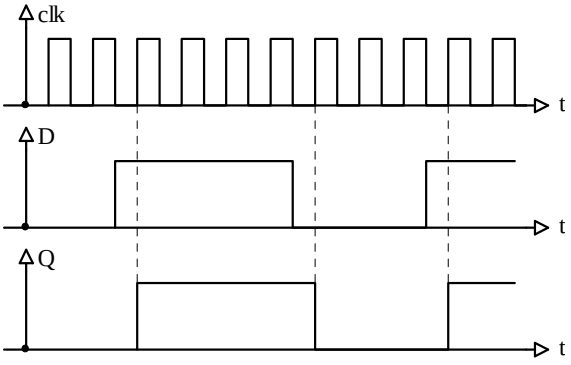
flip-flop rajzjelén a nyil mindig az órajel bemenetet jelzi. A táblázatban a Q_n érték az órajel felfutó éle előtt érvényes állapot, a Q_{n+1} érték viszont az órajel felfutó élét követő új állapot.



(a)

D	Qn	Qn+1
0	0	0
0	1	0
1	0	1
1	1	1

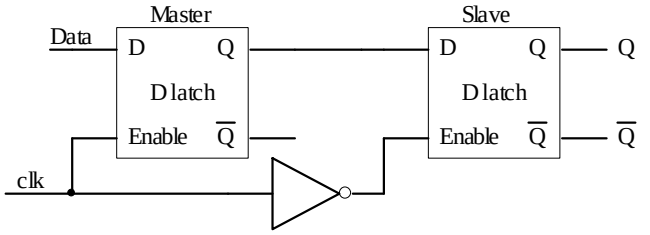
(b)



(c)

3-6 ábra: D flip-flop: (a) rajzjel, (b) vezérlési táblázat, (c) jeldiagramok.

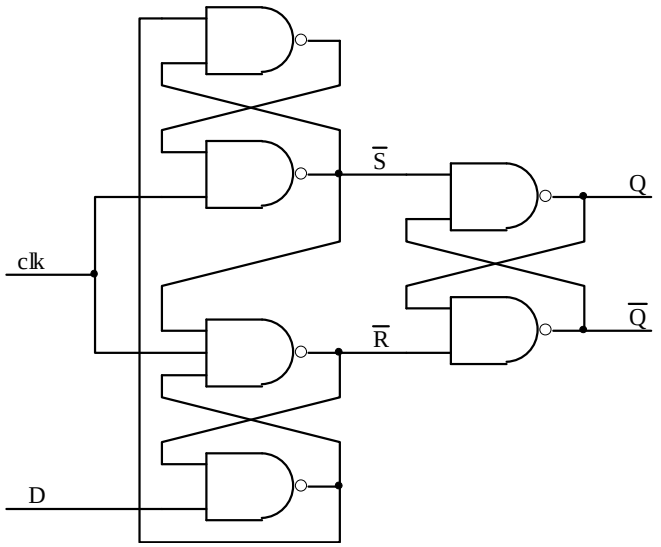
A *D flip-flop* megvalósítható két *D latch* kaszkád kötésével a 3-7 ábrán bemutatott módon (*master-slave* kapcsolás). Amíg az órajel magas logikai szinten tartózkodik, az első *latch* (*master*) transzparens viselkedéséből eredően a *Q* kimenet követi a *D* bemenetet, de a második *latch* ez idő alatt reteszelve van, mivel inverteren keresztül kapja az engedélyező jelet.



3-7 ábra: D flip-flop megvalósítása D latch-ek kaszkád kötésével (master-slave kapcsolás).

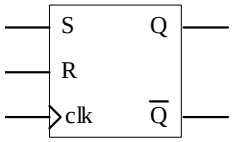
Az órajel lefutó élét követően az első *latch* reteszelődik (megtartja a lefutó élnél érvényes állapotot) a második (*slave*) *latch* viszont transzparenssé válik: a *master latch* kimenetét átírja a saját kimenentére. Kívülről szemlélve úgy tekinthető, hogy a *flip-flop*-ba a lefutó élnél történik a beírás.

A *D flip-flop* megvalósítására létezik más megoldás is, amely a kapuk számát és a sebességet illetően is előnyösebb. Egy ilyen, felfutó éllel aktivizálható kapcsolást a 3-8 ábrán láthatunk. Amíg az órajel alacsony logikai szinten van, az $\overline{S}$ és $\overline{R}$ változók magas logikai szinten vannak, a kimeneti *latch*-be nem történhet beírás. A bemeneteken jelenlevő két-két logikai kapu is egy-egy *latch* kapcsolásnak tekinthetők, amelyek mintegy előkészítik a beírandó értéket a kimeneti *latch* számára. Amikor az órajel magas szintre emelkedik, megtörténik az előkészített érték beírása.



3-8 ábra: D flip-flop hatékonyabb megvalósítása kevesebb számú logikai kapuval és kisebb késéssel.

Léteznek más típusú *flip-flop*-ok is, amelyek a bemenetek száma- vagy a vezérlési táblázat szempontjából különböznek a *D flip-flop*-tól. Ezek az *RS*, a *T* és a *JK flip-flop*-ok. Az *RS flip-flop* az *RS latch* (3-4 ábra) élvezérléssel működő változatának tekinthető, rajzjelét és vezérlési táblázatát a 3-9 ábrán adtuk meg.

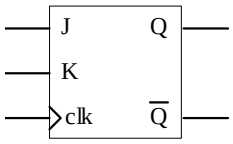


S	R	Qn	Qn+1
0	0	Q	Q
0	1	Q	0
1	0	Q	1
1	1	Q	nem def.

(a) (b)

3-9 ábra: Az RS flip-flop rajzjele (a) és vezérlési táblázata (b).

Az *RS flip-flop* a megfelelő *latch*-hez hasonlóan nem tudja megfelelően kezelni azt az esetet, amikor mindkét adatbeneten logikai egyes jelenkezik. Ennek a gondnak a megoldására fejlesztették ki a *JK flip-flop*-ot, amelynek rajzjele és vezérlési táblázata a 3-10 ábrán látható. A bemeneti két logikai egyes esetén ez a flip flop invertálja az órajel felfutó éle előtt érvényes kimeneneteket.

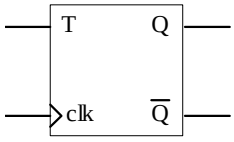


J	K	Qn	Qn+1
0	0	Q	Q
0	1	Q	0
1	0	Q	1
1	1	Q	$\overline{Q}$

(a) (b)

3-10 ábra: JK flip-flop rajzjele (a) és vezérlési táblázata (b).

A *T flip-flop*-nak csak egy adatbemenete van (3-11a ábra) így a vezérlési táblázata (3-11b ábra) is csak két sorból áll. Az órajel felfutó élénél a kimenetek invertálódnak, ha a *T* bemeneten logikai egyest tartunk, más esetekben nem történik változás.

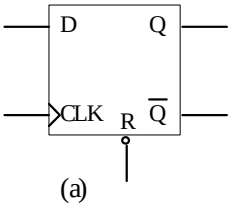


T	Qn	Qn+1
0	Q	Q
1	Q	$\overline{Q}$

(a) (b)

3-11 ábra: A T flip-flop rajzjele (a) és vezérlési táblázata (b).

Az *MSI* áramkörökként gyártott *flip-flop*-okból rendszerint kettő vagy négy darabot építenek egy tokozásba. Ezek a *flip flop*-ok az órajellel szinkronizált bemenetek mellett rendszerint aszinkron szet és/vagy reszet bemeneteket is tartalmaznak. A 3-12 ábra egy aszinkron reszet bementtel ellátott *D flip-flop* rajzjelét és vezérlési táblázatát mutatja. A reszet bemenet független az órajeltől. A bemutatott esetben a kis kör a rajzjelen azt jelenti, hogy a reszetelés alacsony logikai szintnél történik. A táblázatba a *CLK* oszlopban felfelé mutató nyillal jelöltük, hogy a *flip-flop*-ba a beírás az órajel felfutó élénél történik. Az *X* tetszőleges logikai értéket jelöl.

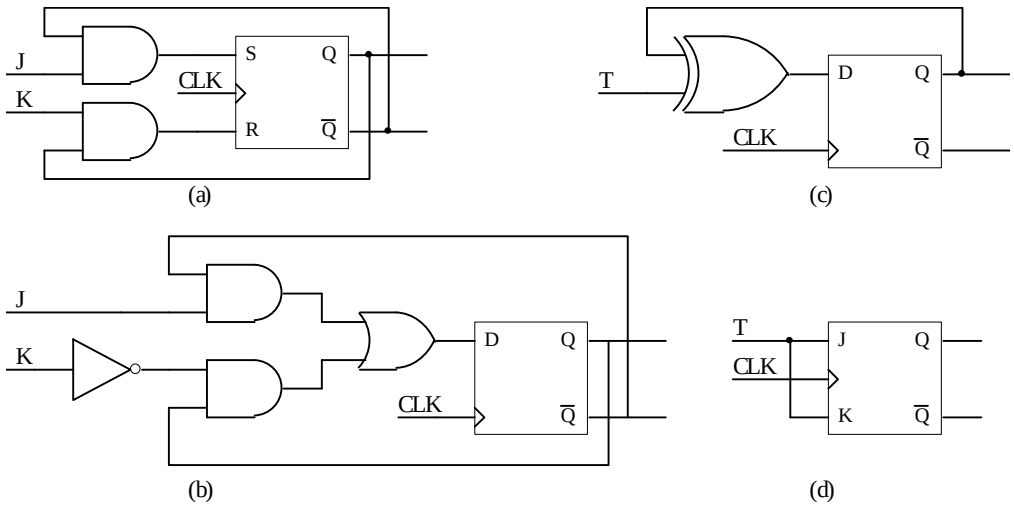


R	CLK	D	Qn	Qn+1
0	X	X	0	1
1	↑	0	0	1
1	↑	1	1	0

(a) (b)

3-12 ábra: D flip-flop aszinkron reszet bemenettel: (a) rajzjel, (b) vezérlési táblázat.

Bármely sorrendi hálózat megépíthető bármelyik típusú *flip-flop* segítségével, esetleg vegyes alkalmazás is számításba jöhet. Az alkalmazott *flip-flop*-ok típusa kihatással van a sorrendi hálózat kombinációs részeinek megvalósítására. Jó választás esetén a kombinációs hálózatok lényegesen leegyszerűsödhetnek. Sajnos nincs egyértelmű szabály a *flip-flop*-ok megválasztására, csak a sorrendi hálózat szintézisének megismétlésével dönthető el, hogy melyik választás az optimális. Szükség szerint az egyes *flip-flop*-ok átalakíthatók más *flip-flop*-okká, külső kötések ill. logikai kapuk alkalmazásával. A 3-13 ábra ilyen átalakításokra ad példákat.



3-13 ábra: Flip-flopok átalakításai: (a) RS flip-flop átalakítása JK flip-flop-pá, (b) D flip-flop átalakítása J-K flip-flop-pá, (c) D-flip-flop átalakítása T flip-flop- pá, (d) JK flip-flop átalakítása T flip-flop-pá.

A mai VLSI technológiában nem igazán döntő, hogy hány logikai kaput fogunk felhasználni a sorrendi hálózat kombinációs részeinek megvalósításához, sőt az sem feltétel, hogy okvetlenül minimalizáljuk a felhasznált *flip-flop*-ok számát. Sokkal fontosabb, hogy a megvalósítás illeszkedjen a programozgható integrált áramkör (*PLD*) belső szerkezetéhez, valamint, hogy minél áttekinthetőbb és megbízhatóbb megoldáshoz jussunk.

3.2. Logikai automaták leírása és szerkesztése

Valamely automatizációs vagy jelfeldolgozási célra megszerkesztett sorrendi hálózatokat logikai automatáknak nevezzük. Az automaták tervezése rendszerint valamilyen szóbeli leírásból indul ki. Ez alapján alkotjuk meg az állapotgráfot, vagy az ezzel egyenértékű állapottáblázatot, amely pontos képet ad az automata viselkedéséről. A táblázatból nyerhető egyenletek segítségével jutunk el az automata kombinációs hálózatának megszerkesztéséhez, majd ezt összekapcsoljuk a megfelelő *flip-flop*-okkal és létrehozzuk a teljes hálózatot.

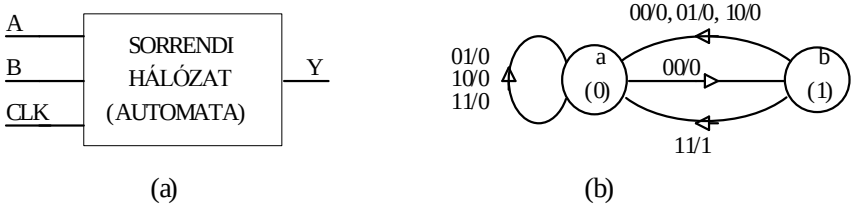
Ugyanezen a módon tervezik a gyártók az általános rendeltetésű regisztereket és számlálókat (sorozatgyártásban levő *MSI* áramkörök), amelyekről a 3.3 és 3.4 szakaszokban lesz szó. A programozható digitális áramkörökkel (*PLD*) végzett szintézist támogató szoftverek is hasonló módon szerkesztik meg a logikai automatákat.

3.2.1. Az állapotgráf

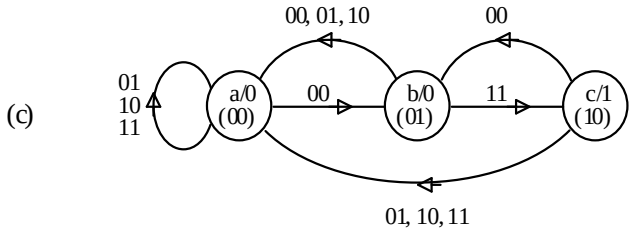
Az automata precíz és jól áttekinthető leírási módja az állapotgráf. A 3-14a ábrán megadott állapotgráfok egy olyan automatára vonatkoznak, amely akkor ad az *Y* kimenenten logikai egyest, ha az előző két órajel ciklusban az *A* és *B* bemeneteken *00* majd *11* kombináció jelent meg, minden más esetben a kimenet logikai nullán van.

A gráf körökkel jelzett csomópontjai felelnek meg az automata egyes állapotainak. Az állapotokat megfelelő számú *flip-flop*-pal fogjuk kódolni. A körökbe vagy az állapot kódját, vagy annak valamilyen közérthetőbb jelét írjuk. A csomópontokat összekötő, nyilakkal irányított vonalak jelzik a lehetséges állapotváltozásokat. A vonalak mellet fel kell tüntetni, hogy a bemeneti változók mely variációja mellett történik az adott állapotváltozás (átmenet).

Az állapotgráf némileg eltér, attól függően, hogy az automatára a *Mealy* vagy a *Moore* féle modellt alkalmazzuk. A *Mealy* modellnél a gráf ágain tüntetjük fel, hogy adott átmenet közben a kimeneten milyen bitkombináció jelenik meg (3-14b ábra), míg a *Moore* féle modellnél a kimeneteket az adott állpotok dekódolásával kapjuk, ezért a kimeneti kombinációkat az állapotokat jelző körökben tüntejük fel (3-14c ábra). Az állapotok száma is eltérhet a két modellnél: a megadott automatánál a a *Mealy* modell két állapotú, a *Moore* modell három állapotú.



3-14 ábra: Egy sorrendi hálózat (a) és állapotgráfja (b) Mealy féle modell esetére, (c) Moore féle modell esetére.



3.2.2. Az állapottáblázat

Az automata szintézisének következő fázisa az állapottáblázat felírása. A táblázat ugyanazt az információt tartalmazza, mint az állapotgráf, de a hálózat szintéziséhez alkalmasabb formában. A 3-14 ábrán megadott állapotgráfoknak megfelelő állapottáblázatokat a 3-1 és 3-2 táblázatokban láthatjuk.

3-1 táblázat: Állapottáblázat a logikai automatához a Mealy féle modell esetére.

Jelenlegi állapot	Következő állapot BA= 00	01	10	11	Kimenet Y BA= 00	01	10	11
a (0)	b	a	a	a	0	0	0	0
b (1)	a	a	a	a	0	0	0	1

3-2 táblázat: Állapottáblázat a logikai automatához a Moore féle modell esetére.

Jelenlegi állapot	Következő állapot BA= 00	01	10	11	Kimenet Y
a (00)	b	a	a	a	0
b (01)	a	a	a	c	0
c (10)	b	a	a	a	1

A táblázatokban fel kell tüntetni a sorrendi hálózat összes lehetséges állapotát, az összes bemeneti jelkombinációt, minden egyes esetre a hálózat következő állapotát és a kimeneti változó(k) értékeit. A 3-1 és 3-2 táblázatokban a jelenlegi állapotokat az első oszlopba írtuk fel (*a, b* illetve *a, b, c*), a megfelelő kódokat zárójelben adtuk meg. A táblázatok központi részeiben foglalnak helyet a következő állapotok, amelyek mindkét modellnél a bemeneti változók értékeitől is függenek, ezért az új állapotok feletti sorban feltüntettük a bemeneti jelkombinációkat.

A táblázatok jobb oldalán definiáltuk a kimeneti változó (*Y*) értékeit. A *Mealy* féle modellnél (3-1 táblázat) a kimenet a pillanatnyi bemeneti értékektől is függ, ezért a kimeneti értékek feletti sorban megadtuk az összes lehetséges bemeneti jelkombinációt. A *Moore* féle automatánál (3-2 táblázat) a kimeneteket egyértelműen meghatározza a jelenlegi állapot, ezért ott az összes lehetőség egy oszlopban szerepel.

3.2.3. Az állapotok kódolása

Ha *n* darab *flip-flop*-ot alkalmazunk, elvileg 2^n állapotot tudunk velük kódolni, így a bonyolultabb automaták is viszonylag kevés számú elemi memóriával megoldhatók. Az *MSI* áramkörökből történő hagyományos tervezésnél általában igyekszünk a minimális számú *flip-flop*-ot alkalmazni. Ez rendszerint bemeneti és a kimeneti kombinációs hálózatok bonyolításához vezet. Elsősorban a kimeneti hálózat bonyolításától kell tartani, mivel az állapotok dekódolása alkatrészigényes feladat.

A minimális számú *flip-flop*-pal történő megvalósításnál nem mindegy, hogy az egyes állapotokat hogyan fogjuk kódolni. A helyes kódválasztás lényegesen egyszerűsítheti a sorrendi hálózat megvalósítását, ugyanakkor javíthatja a megbízhatóságot. Az optimalizációra léteznek bizonyos szisztematikus kézi módszerek, de ezek nem vezetnek egyértelműen az optimális megoldáshoz. Ma az optimalizáció alatt a különféle megvalósítási lehetőségek számítógépes szintézisét és szimulációját értjük. Az egyes esetek kiértékelése alapján döntünk a végleges változatról.

Az alkalmazott elemi memóriák típusa (*D, T, JK, SR*) is kihatással van a megvalósítandó hálózat bonyolultságára. A típusválasztásra vonatkozóan sincs egyértelmű szisztematikus eljárás, amely az optimális megoldáshoz vezet. Különösen nehézkes az optimalizáció, ha egyidőben különböző típusú *flip-flop*-ot használunk. Itt is a számítógépes szintézis javasolható. A tervezői szoftverek lehetővé teszik, hogy gyorsan kielemezzünk többéle lehetséges megoldást és érdemi döntést hozzunk.

A *PLD*-kel történő modern tervezésnél rendszerint nem szükséges vagy nem célszerű a minimális számú tárolóelemmel történő állapotkódolás. Igen gyakori az egy állapot-egy tárolós megoldás. A flip-flop-ok száma így nagy, de nincs szükség az állapotok dekódolására.

3.2.4. A *flip-flop*-ok vezérlési egyenletei

Mindkét automata modellnél a hálózat következő állapotát a pillanatnyi állapot és a bemeneti változók pillanatnyi értékei határozzák meg. A tervező feladata, hogy megszerkessze a megfelelő kombinációs hálózatot, amely előkészíti a *flip-flop*-ok bementeire a megfelelő jeleket, amelyek hatására az órajel következő megfelelő (felfutó vagy lefutó) élénél megtörténik a kívánt beírás.

Legegyszerűbb a kombinációs hálózat egyenleteinek felírása D *flip-flop*-ok esetére: itt a D bemenetekre a *flip-flop*-ok új állapotát kell hozni. Más típusú *flip-flop*-oknál figyelembe kell venni, hogy a kívánt új értéket milyen bemeneti jel vagy jelkombináció tudja létrehozni.

A bemeneti és a kimeneti kombinációs hálózat megszerkesztéséhez a 3-1 táblázatot (*Mealy* féle modell) olyan alakra hozzuk, hogy abból a hálózatok egyenletei könnyen levezethetők legyenek (3-3 táblázat).

Qn	BA	Qn+1=D	Y
0	00	1	0
	01	0	0
	10	0	0
	11	0	0
1	00	0	0
	01	0	0
	10	0	0
	11	0	1

3-3 táblázat: A 3-14b ábrán megadott Mealy féle automata gerjesztési és kimeneti táblázata.

A keresett gerjesztési egyenlet a táblázat alapján:

$$D = \overline{Q}\overline{B}\overline{A}$$

Összetettebb esetekben logikai minimalizációt kell végezni, hogy az áramkör megvalósítása minél egyszerűbb legyen.

A *Moore* féle modellnek (3-14c ábra) megfelelő gerjesztési és kimeneti táblázat a 3-4-es táblázat. A táblázat alapján a keresett gerjesztési egyenletek D *flip-flop*-pal történő megvalósítás esetére a következők:

$$D1 = \overline{Q1}Q0BA$$

$$D0 = \overline{Q1}\overline{Q0}\overline{B}\overline{A} + Q1\overline{Q0}\overline{B}\overline{A} = \overline{Q0}\overline{B}\overline{A}$$

$Q1_nQ0_n$	BA	$Q1_{n+1}$=D1, $Q0_{n+1}$=D0	Y
00	00	01	0
	01	00	0
	10	00	0
	11	00	0
01	00	00	0
	01	00	0
	10	00	0
	11	10	0
10	00	01	1
	01	00	1
	10	00	1
	11	00	1
11	00	00	0
	01	00	0
	10	00	0
	11	00	0

3-4 táblázat: A 3-14c ábrán megadott Moore-féle automata gerjesztési és kimeneti táblázata.

3.2.5. A kimenetek képzése

A *Mealy* féle automatánál rendszerint kisebb az állapotváltozók száma, de a kimeneti kombinációs hálózat bemeneteire rá kell vezetnünk a bemeneti változókat is. A 3-3 táblázat alapján a kimeneti egyenlet:

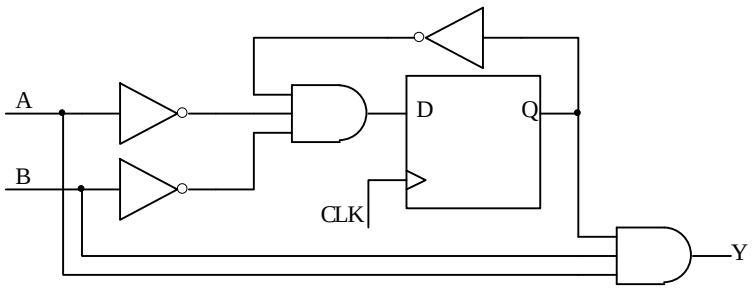
$$Y = QBA.$$

A *Moore* féle modellnél a kimeneti változó(k) értékei csak a hálózat jelenlegi állapotától függenek. Legegyszerűbb az eset, ha a *flip-flop*-ok kimenetei egyben a hálózat kimenetei is. Ha nem ez a helyzet a kimeneteket megfelelő kombinációs hálózattal állítjuk elő. A hálózat bemenetei a *flip-flop*-ok állapotai. A 3-4 táblázat alapján a 3-14c ábrán közölt *Moore* féle automata kimeneti egyenlete a minimalizáció után következő:

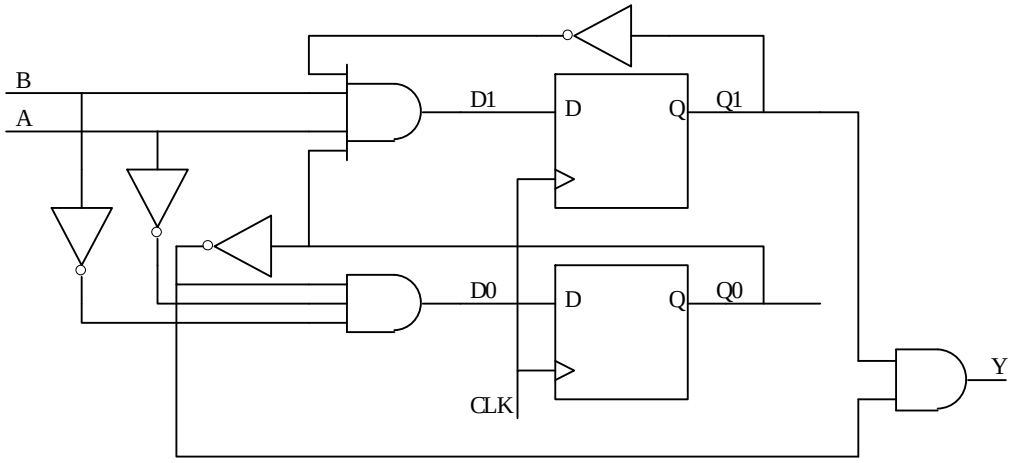
$$Y = Q1\overline{Q0}.$$

3.2.6. A teljes hálózat kialakítása

A teljes sorrendi hálózat megrajzolását az eddigi eredmények összegzésével végezzük. A hálózat központi részét képezik a tárolóelemek, amelyeket a választott kódnak megfelelően kell felsorakoztatni. A *flip-flop*-ok vezérlési egyenletei alapján felrajzoljuk a bemeneti kombinációs hálózatot. A végén megrajzoljuk a kimeneti kombinációs hálózatot. A 3-14b ábrán megadott *Mealy* típusú automatára a logikai hálózatot a 3-15 ábrán láthatjuk, hasonlóan a 3-14c ábra *Moore* féle automatájának megfelelő hálózatot a 3-16 ábrán közöljük.



3-15 ábra: A 3-14b ábrán megadott Mealy típusú automatát megvalósító logikai hálózat.



3-16 ábra: A 3-14c ábrán megadott Moore típusú automatát megvalósító logikai hálózat.

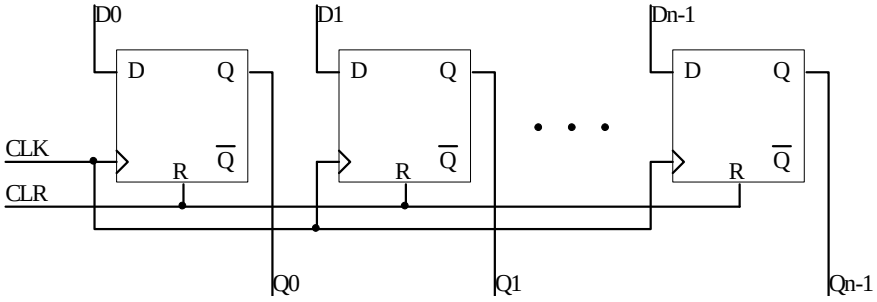
Az itt kifejlesztett sorrendi hálózatok szerepét (a bemenő sorozatok figyelését) elláthatja egy megfelelően programozott mikroprocesszor is. Figyelembe kell azonban venni, hogy a hardveres megoldás összehasonlíthatatlanul egyszerűbb és a sebessége is lényegesen nagyobb.

3.3. Regiszterek

A regiszterek sorrendi hálózatok amelyek kis mennyiségű információ ideiglenes tárolására alkalmasak. A tárolást általában a regiszterben felsorakoztatott *D flip-flop*-ok, esetleg más flip flop-ok vagy *latch* áramkörök végzik. Az információ bevitele és kiolvasása történhet sorosan (bitenként) vagy párhuzamosan (minden bit egyszerre). Megkülönböztetünk stacionáris és shift regisztereket. A shift regiszterek a tárolás mellett képesek az információ léptetésére is.

3.3.1. Közönséges (stacionáris) regiszterek

A 3-17 ábrán egy n bit kapacitású stacionáris regiszter logikai rajzát aztuk meg. Közös órajellel (*CLK*) vezérelt *D flip-flop*-ok végzik a tárolást. A beírás párhuzamosan történik, az órajel felfutó élénél. A *flip-flop*-ok tartalma nullázható az aszinkron jellegű *CLR* jellel. A kiolvasás elvileg mindig végezhető, kivéve a beírást követő késleltetési időt, amikor a *flip-flop*-ok kimeneti állapotai még nem stabilizálódtak. Szükség szerint használhatunk olyan *flip-flop*-okat, amelyeknek három állapotú kimenetük van, ez által több regiszter kimenete közös adatvonalra köthető.



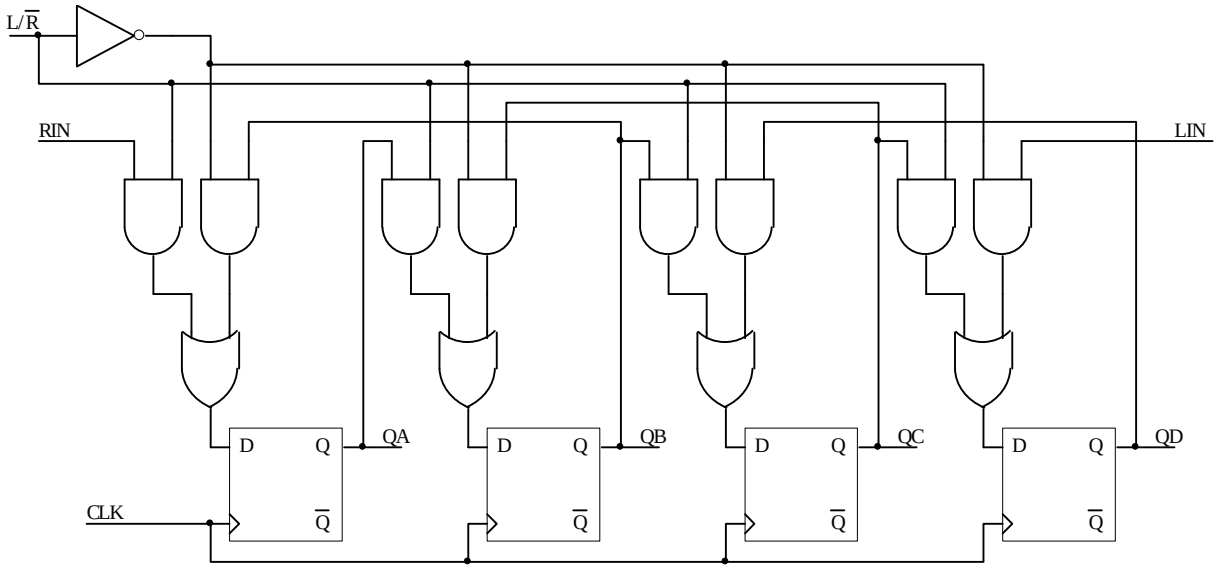
3-17 ábra: Stacionáris regiszter logikai rajza.

3.3.2. Léptető *(shift-)* regiszterek

A *shift* regisztereknél az elemi tárolók tartalma a szomszédos tárolókba írható át (léptethető) az órajel hatására. A lépetetés történhet jobb vagy bal irányba, univerzális esetben vezérlőjellel választható az irány. A léptetés során a regiszter bemeneti oldalán elhelyezkedő tárolóba a korábbi tartalom helyébe a bemeneti változó értéke íródik. A kimeneti oldalon az utolsó

tároló tartalma elveszik a léptetéskor. Összetettebb léptető regisztereknél lehetőség van párhuzamos beírásra és kiolvasásra is a soros (bitenkénti) eljárás mellett.

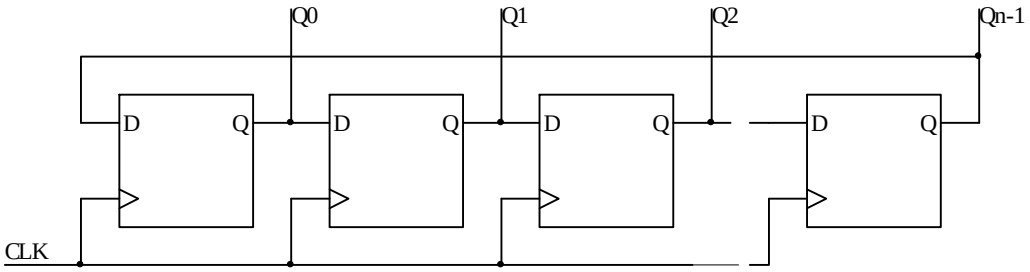
A 3-18 ábrán egy négybites univerzális shift regiszter logikai rajzát láthatjuk. A léptetés irányát az $L/\overline{R}$ jellel választjuk, *RIN* a soros bemenet jobb irányba történő léptetésnél, *LIN* a bemenet balra léptetésnél. A soros kimenetet a *QA* vagy a *QB* kimenet képezi, a léptetés irányától függően. A kiolvasás történhet párhuzamosan is. Az adott egoldásnál párhuzamos beírásra nincs lehetőség. A léptetést végző órajel (*CLK*) közös minden *flip-flop*-ra.



3-18 ábra: Kétirányú léptetőregiszter logikai rajza.

3.3.3. Gyűrűs regiszterek (gyűrűs számlálók)

A léptetőregiszter soros kimenetét visszacsatolva ugyanazon regiszter soros bemenetére, gyűrűs regisztert kapunk. Az egyszer beírt információ ebben a regiszterben mindaddig kering, amíg az órajellel vezérlést adunk. Mivel ennek a regiszternek a normális üzemben nincs adatbemenete, hanem önállóan ismétlődő sorozatok előállítására képes, ezt a kapcsolást (gyűrűs) számlálónak is nevezzük (3-19 ábra).

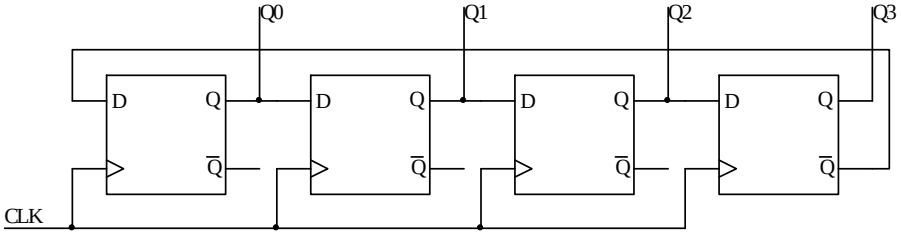


3-19 ábra: Gyűrűs regiszter (számláló) logikai rajza.

Gyakorlati alkalmazásnál az információ keringetése előtt megfelelő tartalmat kell beírni a gyűrűs regiszterbe. Ezt rendszerint párhuzamosan végezzük. A leggyakoribb esetben csak egy *flip-fop*-ot szetelünk, a többibe logikai nullát írunk, vagy fordítva. Léteznek olyan megoldások, amelyeknél induláskor tetszőleges tartalom áll be, de néhány órajel impulzus után már csak egy tároló van szetelve. Ezt követően a működés szabályos.

A Johnson féle gyűrűs regisztert (számlálót) úgy kapjuk, hogy az utolsó tároló kimentét invertálva vezetjük vissza a soros bemenetre (3-20a ábra). Az ún. szabályos esetben a nullák és

egyesek a 3-20b ábrán megadott táblázat szerint keringenek a regiszterben. Ha *n* tárolót alkalmazunk, a keringési ciklus *2n* órajel periódus után ismétlődik. A szabályos eset önmagától kialakul, ha induláskor minden *flip-flop*-ot reszetelünk. Ha ezt nem tesszük meg, megtörténhet, hogy a kapcsolás szabálytalan kimeneteket ad. Különböző visszacsatolásokkal is megoldható a szabályos viselkedés elérése.



(a)

ÁLLAPOT	Q0	Q1	Q2	Q3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1
0	0	0	0	0

(b)

3-20 ábra: A Johnson féle gyűrűs regiszter (számláló) lehetséges kapcsolása (a) és állapot táblázata (b).

3.4. Számlálók

Az olyan regisztert, amely a vezérlőimpulzusok hatására előre meghatározott állapotokon halad keresztül, számlálóknak nevezzük. A vezérlőimpulzusok eredhetnek valamilyen órajel forrásból, vagy bármilyen más digitális jeleket létrehozó áramkörből. Az impulzusok általában periódikus jellegűek, de a számlálók működhetnek véletlenszerűen megjelenő impulzusok hatására is. A vezérlőimpulzusok mellett egyes számlálóknak más bemeneti vonalaik is vannak, amelyekkel a számlálás módját tudjuk befolyásolni.

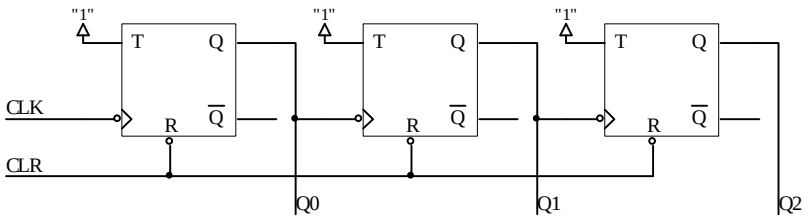
A számláló állapotainak sorrendje megegyezhet a bináris számlálási sorrenddel (bináris számláló), de szükség szerint bármilyen más sorrend kialakítható. Az összes lehetséges állapot száma a számláló modulusa. A tárolóelemek mellett a számlálók rendszerint megfelelő bemeneti kombinációs hálózatot is tartalmaznak, ez biztosítja a számláló központi részét alkotó regiszter tartalmának előírt változásait. A sorrendi hálózatokra egyébként jellemző kimeneti kombinációs hálózat általában nem létezik a számlálóknál, a tárolóelemek kimenetei egyben a számláló kimenetei is. Ez egyben azt is jelenti, hogy a számlálók *Moore* típusú hálózatok.

A számlálókat két osztályba soroljuk: vannak aszinkron (soros) jellegű számlálók és szinkron (párhuzamos) számlálók.

3.4.1. Aszinkron (soros) számlálók

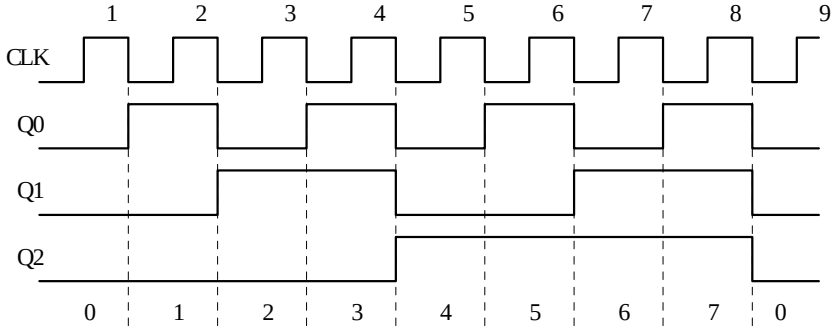
Az aszinkron számlálóknál az egyes *flip-flop*-ok órajel bemenetei nem közös vezérlést (órajelet) kapnak, hanem bizonyos tárolóelemek kimeneteit használjuk fel más *flip-flop*-ok vezérlésére. Ez által a bemeneti kombinációs hálózat rendszerint lényegesen leegyszerűsödik, esetleg feleslegessé is válik.

A legegyszerűbb szerkezetű aszinkron számláló *flip-flop*-ok kaszkád kötésével kapható (3-21 ábra). A vezérlőimpulzusokat az első tároló órajel bemenetére kötjük, majd az egyes *flip-flop*-ok kimeneteit a következő *flip-flop*-ok órajel bementére vezetjük. Szükség szerint a tárolóelmekre alkalmazható egyidejű reszet (*CLR*) a számláló kezdeti állapotának a beállítására.



3-21 ábra: Aszinkron számláló T flip-flop-okkal.

A számláló a bemutatott elrendezésben akkor fog előre számlálni (természetes bináris sorrendben), ha az alkalmazott *flip-flop*-ok az órajel lefutó élére reagálnak. Erre az esetre a jeldiagramokat a 3-22 ábrán adtuk meg. Ha megfigyeljük a *flip-flop*-ok állapotait az egyes órajel ciklusok alatt (3-5 táblázat), elmondható hogy a számláló természetes bináris sorrendben számlál. A három tárolóelemmel a számláló modulusa nyolc, általános esetben *n* tárolóelemmel 2^n modulusú számlálót valósíthatunk meg. Visszafelé számlálást akkor kapunk, ha a *flip-flop*-ok pozitív élre reagálnak, ugyanakkor nem a Q, hanem a $\overline{Q}$ kimeneteket vezetjük a következő *flip-flop*-ok órajel bemeneteire.



3-22 ábra: Az aszinkron számláló vezérlőjele és kimeneti diagramjai.

Q2	*Q1*	*Q0*
0	*0*	*0*
0	*0*	*1*
0	*1*	*0*
0	*1*	*1*
1	*0*	*0*
1	*0*	*1*
1	*1*	*0*
1	*1*	*1*

3-5 táblázat: Három bites aszinkron számláló állapotainak sorrendje.

A 3-22 ábrán megadott diagramok megrajzolásakor nem vettük figyelembe a *flip-flop*-ok késéseit. A kaszkád kötésből eredően az egyes új állapotok a bemeneti vezérlőjelhez képest váltakozó értékű késés után jelennek meg. Ha időközben kiolvassuk az egyes *flip-flop*-ok állapotait, valószínűleg téves eredményt fogunk kapni.

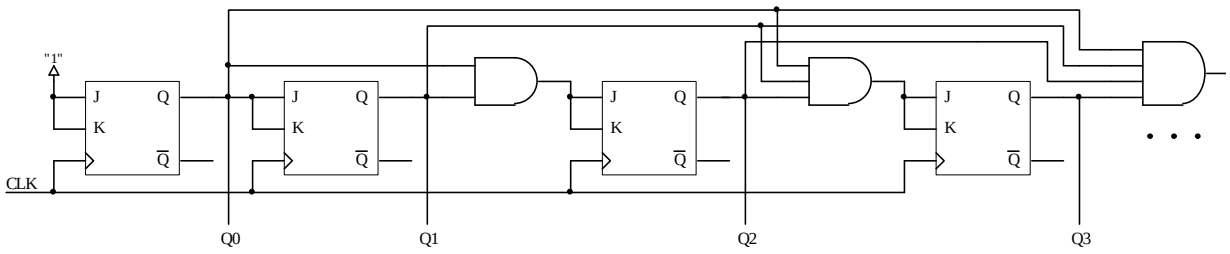
A megoldás egyrészt az órajel frekvenciájának korlátozásában keresendő (így elegendő idő marad az egyes hazárdjelenségek lecsengésére), másrészt a kiolvasást közvetlenül az új vezérlőimpulzus megjelenése előtt kell végezni (nem pedig a vezérlőjel után). Ha az aszinkron számlálót csak frekvenciaosztóként használjuk (pl. a 3-21 ábrán csak a *Q2* jelet vezetjük tovább), az órajel frekvenciáját csak az első *flip-flop* reakcióideje korlátozza.

3.4.2. Szinkron (párhuzamos) számlálók

A szinkron számlálóknál az egyes *flip-flop*-ok órajel bemenetei közös vezérlést kapnak, így érjük el, hogy a kimenetek nagyjából egy időben válnak érvényessé. Megfelelő kombinációs hálózatra van szükség, hogy az egyes *flip-flop*-ok a megfelelő pillanatban változtassák állapotukat. Bináris számlálónál megfigyelhető (3-23 ábra), hogy a legkisebb helyi értékű bitet képviselő *flip-flop* az órajel minden egyes periódusában változtatja értékét. Ez egy *T* tárolóval oldható meg a legkönnyebben.

Ha az egész számlálót *T* tárolókból építjük fel, érvényes, hogy a *T* bemenetekre akkor kell logikai egyest hozni, ha az adott *flip-flop* állapotának változnia kell. A bináris számsort figyelve (3-5 táblázat) megállapítható, hogy a nagyobb helyi értékű bitet akkor kell szetelni, ha az addigi számlálás során minden kisebb helyi értékű bit logikai egyesen van (pl. *0111* állapotból *1000* állapotba kell lépni). Ezeknek az eseteknek a dekódolását *ÉS* kapukkal végezzük. Így alakítható ki a

3-23 ábrán bemutatott szinkron bináris számláló. Az áramkör módszeres szintézise a logikai automatáknál bemutatott módon is ugyanezt az eredményt adja.



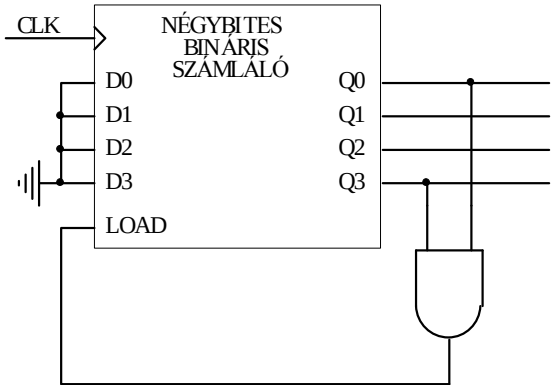
3-23 ábra: Szinkron bináris számláló logikai rajza.

Ez a kapcsolás csak két fokozatú: az órajel előző aktív élét követően csak egy *ÉS* kapu és egy *flip-flop* késésének kell elmúlni, hogy a tárolók kimenetei érvényesek legyenek.

A szinkron számlálóknál a bemeneti kombinációs hálózat megfelelő átalakításával elérhető, hogy a számlálás visszafelé történjék. Az MSI árakörök formájában gyártott számlálók között vannak kétirányú megoldások is: itt a bemeneti kombinációs hálózatot létrehozzák mindkét esetre, és egy ellenőrző jel határozza meg, melyik hálózat működjön, ettől függ a számlálás iránya.

Gyakori igény a számláló modulusának növelése. Az MSI számláló áramkörök rendszerint tartalmaznak megfelelő kimenetet és bemenetet az egyes számláló modulok kaszkád kötéséhez (átvitel – *carry*). Az átvitelen keresztül értesíti az előző egység a következőt, hogy végigszámolt és a következő fokozatnak kell lépnie.

Az *n* darab tárolót tartalmazó számláló modulusa lecsökkenthető a 2^n érték alá. Erre két mód kínálkozik. Az egyik az automaták tervezésénél bemutatott szisztematikus módszer a állapotgráf és állapottáblázat segítségével. A másik módszer kész 2^n modulusú számlálóból indul ki. A tervezett utolsó állapot eléréséig a számláló a szokásos módon működik. Az utolsó állapot detektálására megfelelő külső dekódert építünk ki (3-24 ábra), amely szinkron vagy aszinkron módon reszeteli az összes *flip-flop*-ot és így újraindul a számlálás. A bemutatott esetben tizes modulusú számlálót valósítunk meg, mivel a reszetelés a kilencedik (1001) állapot után nulla értékek (D0...D3) szinkron beírásával történik.



3-24 ábra: Modulus rövidítésével kapott tizes modulusú számláló logikai rajza.

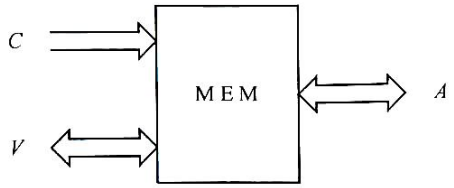
4. Vegyes hálózatok

Az itt tárgyalandó digitális áramköröket azért nevezzük vegyes hálózatoknak, mert rendszerint tartalmaznak kombinációs és sorrendi elemeket is. A memóriáknál az alapfunkció sorrendi jellegű, de a vezérlést kombinációs hálózat valósítja meg. Az aritmetikai áramkörök a műveletvégzéshez elsősorban kombinációs hálózatot használnak, de a vezérlést rendszerint logikai automata valósítja meg. A *D/A* és *A/D* átalakítók a kétfajta (kombinációs és sorrendi) digitális elemek mellett analóg kapcsolásokat is tartalmaznak.

4.1. Memóriák

A memóriák nagyobb mennyiségű adat tartós vagy ideiglenes tárolására alkalmas digitális áramkörök. A korábban említett regiszterek (4.3 pont) szerepe is az információ tárolása, de a memóriáknál a nagy mennyiségű adat hatékonyabb szervezést igényel. A tárolási eszközök ma nagyon sokrétűek (pl. mágneses- és optikai eszközök), mi itt csak a félvezetős tárolókkal foglalkozunk.

Minden memória áramkörre a 4-1 ábrán megadott tömbvázlat érvényes. A *C* jelzésű címvonalakon keresztül jelöljük ki, hogy a memória melyik elemi cellájával kívánunk kommunikálni. A *V* vezérlőbementek által utasítjuk az áramkört a kommunikáció céljáról (beírás, kiolvasás, harmadik állapot érvényesítése), esetleg visszajelzést kapunk a memória állapotáról (pl. foglaltság). Az adatokat az *A*-val jelölt vonalakon keresztül vezetjük a memóriába vagy a memóriából más áramkörökbe.



4-1 ábra: Memória áramkör tömbvázlata.

4.1.1. A memóriák felosztása és jellemzőik

A sokféle hardveres megoldást tartalmazó memória áramköröket különböző elvek szerint osztályozhatjuk.

- A szoftveres programozású berendezéseknél alkalmazott memóriák tartalmához gyorsan hozzá kell férni a tartalom kiolvasása vagy megváltoztatása érdekében, ezért itt gyors és változtatható tartalmú memóriák szükségesek. Ugyanitt igény van az állandó tartalommal rendelkező memóriákra egyes alapvető programok tárolására.
- A tárolás módja a memóriacellákban lehet statikus vagy dinamikus. A statikus cellák tulajdonképpen *flip-flop*-ok (mint a regisztereknél), ezeket szervezik megfelelő módon, hogy az egyes cellák könnyen elérhetőek legyenek. A statikus cellák addig tartják az adatokat, amíg új beírás nem történik vagy amíg a tápfeszültség meg nem szünik. Léteznek olyan különleges megoldások is, amelyek beépített *Li-ion* elemet tartalmaznak, amelynek köszönhetően a memória évekig is megőrzi a tartalmát külső táplálás nélkül.
- A dinamikus celláknál az adatok tárolását egy-egy *MOSFET* bemeneti kapacitása végzi. Beíráskor ezt a parazita kapacitást töltjük fel vagy ürítjük ki, attól függően, hogy logikai egyest vagy nullát kívánunk tárolni a cellában. A kiolvasás a *MOSFET* csatorna állapotának ellenőrzésével történik: a vezető állapot logikai egyesnek felel meg, a nem vezető állapot pedig logikai nullának. A kapacitásban tárolt töltésmennyiség fokozatosan elszivárog, így a beírt adatok lassan elvesznének. A megoldás az adatok időszakos frissítése. A frissítés az olvasáshoz hasonló módon történik, külön áramkör gondoskodik a frissítés folyamatáról.
- A cellát, amellyel kommunikálni akarunk, általában címzéssel nevezzük meg illetve választjuk ki. Ezt nevezik tetszőleges vagy közvetlen elérésnek (*random access*). A címzés

által bármely cellához kb. ugyanannyi idő alatt hozzá tudunk férni, de a címzést végző hálózat viszonylag bonyolult. Lényegesen egyszerűsödik a memória felépítése, ha az adatokat sorban írjuk be illetve olvassuk ki, az így kapott memóriák soros hozzáférésűek. A hátrány, hogy a sorban távolabb levő cellák tartalmához arányosan több idő után tudunk csak hozzáférni.

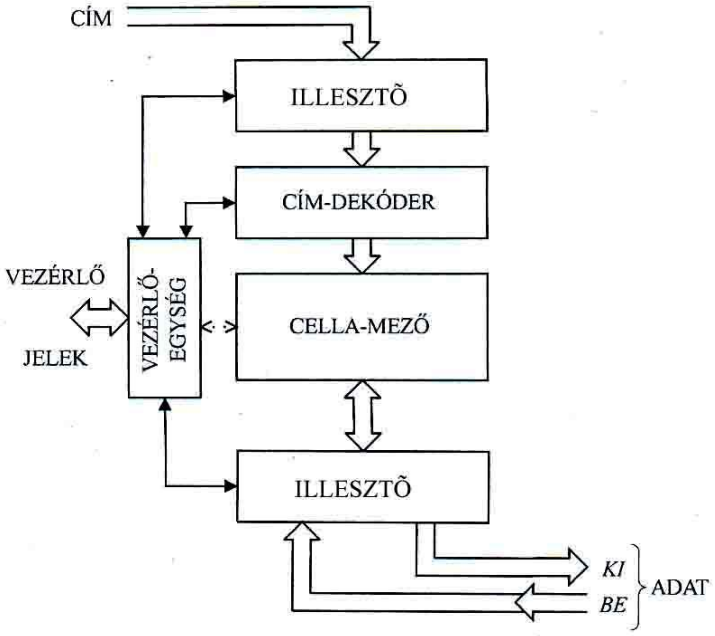
- A memóriaáramkörök nagy részét ma *CMOS* technológiával készítik, de vannak bipoláris termékek is. Az egy lépésben (egy címzéssel, vagy egy léptetéssel) elérhető adat lehet egy bit vagy több bit (pl. nyolc vagy tizenhat).
- A memória áramkörökre a legfontosabb adatok a tápfeszültség és a logikai szintek mellett a kapacitás (a memória cellák száma) és a sebesség (írás, olvasás, törlés).

A memóriaáramkörök különböző kereskedelmi elnevezések alatt futnak.

- A *RAM* (*random accesss memory*) a berendezések operatív memóriáját képező alkatrész, közvetlen hozzáférésű memória, viszonylag gyors hozzáféréssel (általában néhányszor tíz *ns*), lehet statikus vagy dinamikus kivitel. A dinamikus *RAM*-ba ugyanarra a szilícium lapocskára néhányszor több memóriacella elfér mint a statikus megoldásnál, ezért olcsóbb megoldásnak számít, sajnos a frissítést végző külső áramkör bonyolítja a rendszer megépítését.
- A *ROM* (*read only memory*) olyan memória, amelynek a tartalmát nem lehet változtatni, viszont tetszőleges számszor kiolvasható. A tüzetesebb elemzés szerint a ROM-ok tulajdonképpen tisztán kombinációs hálózatok: egy dekóder és egy kóder kaszkád kötésének tekinthetők. A dekóder dekódolja a bemeneti címvonalakat, a kóder pedig létrehozza a kimeneti értékeket. A berendezések alapszoftverét szokták ilyen áramkörben tárolni. A tartalom beírása történhet gyárilag (mask programmable *ROM*) vagy a felhasználó által (*OTP*-one time programmable *ROM*). A *ROM*-ok csak sorozatgyártásban alkalmazhatók, fejlesztési munkáknál az egyszeri programozási lehetőség nem elégséges.
- Az *EPROM* (*electrically programmable ROM*) megfelelő villamos vezérlőjelekkel programozható illetve újraprogramozható memória, de újraprogramozás előtt ibolyántúli sugárzás segítségével törölni kell a tartalmat. Ezek a memóriaáramkörök a tokozás tetején levő ablakról ismerhetők fel. A beírás hasonló módon történik, mint a dinamikus memóriákba, csak a csatorna állapotát módosító töltést úgy helyezik el, hogy az sok évig sem tud távozni. Az *EPROM*-ok sokszor újraprogramozhatók, figyelembe kell azonban venni, hogy a törlés több percig tart, ami gyakran hátráltatja a fejlesztést illetve a termelést, így az *EPROM*-ok népszerűsége esik.
- Az *EEPROM* (electrically erasable *PROM*) tokozása ablak nélküli, mivel az adatok törlése villamos jelekkel történik, ugyanígy a beírás. Természetesen a törlés és az újraírás sebessége messze elmarad a *RAM*-ok írási és olvasási sebességétől. Gyakori az *EEPROM*-ok soros kommunikációs kivitele. Ha nincs szükség gyors adatcserére a külvilággal, a soros szerkezet nagyban egyszerűsíti a tokozást, hiszen csak egy adatvonalra van szükség.
- A *flash EEPROM* vagy egyszerűen csak *flash* memória név alatt olyan áramköröket értünk amelyek villamos jelekkel viszonylag gyorsan (másodperc nagyságrend) törölhetők és újraírhatók, ugyanakkor ezekhez a műveletekhez nincs szükség külön tápfeszültségre illetve megnövelt amplitúdójú vezérlőjelekre, mint az előző pontokban ismertetett *EPROM* és *EEPROM* áramköröknél. A törlést és az újraírást elvégezheti a házigazda szerepét betöltő processzor. Rendszerint a törlésre vonatkozóan bizonyos korlátozások vannak, csak blokkonként végezhető a művelet. A *flash* memóriák használhatók programmemóriaként, a kiolvasási sebesség elég nagy, de a *RAM* szerepét nem tudják ellátni, mivel a beírás ideje hosszabb. Nagy előny viszont, hogy a beírt tartalom tartósan megőrződik a tápfeszültség kikapcsolása után is.

4.1.2. A memória áramkörök belső szerkezete

Mint említettük, a nagy számú adat tárolása a memóriában megfelelő szervezettséget igényel. A címzéssel működő memória áramkörök szerkezetét a 4-2 ábra mutatja.



4-2 ábra: Memória áramkör belső szerkezete.

A központi rész a cella-mező, amely kellő számú elemi memóriát tartalmaz. Az adott cellát megnevező cím egy illesztő áramkörön keresztül érkezik a cím dekóderbe. Ennek szerepe, hogy kiválassza a cellát amelyikkel kommunikálni szeretnénk. A dekóder összetetsége a memória kapacitásának növekedésével rohamosan (négyzetesen) nől. Ez okból az elemi memóriákat általában nem lineárisan (egy sorban) helyezik el hanem mátrix alakban. A címzés ekkor két kisebb dekóderrel történik (sor dekóder és oszlop dekóder), amelyek együttesen is sokkal kevesebb számú logikai kaput tartalmaznak. Ez az eljárás feltételezi, hogy az egyes memória celláknak két címbemenetük van. A címvonalakat a nagy kapacitású dinamikus memóriáknál rendszerint multiplexálják a tokozás egyszerűsítése végett: külön beolvassák és tárolják az oszlop címeket majd a sor címeket.

Az adatok bevitele illetve kiolvasása az adat illesztőn keresztül történik. Általában ugyanazok az adatvonalak használatosak mindkét irányban. Ennek oka a tokozás egyszerűsítése. Beírásnál az adatoknak már kevéssel a beírás előtt érvényeseknek kell lenniük. Kiolvasásánál számítani kell rá, hogy az adatok kis késéssel jelennek meg. Az adatlapok ezzel kapcsolatban konkrét értékekkel szolgálnak.

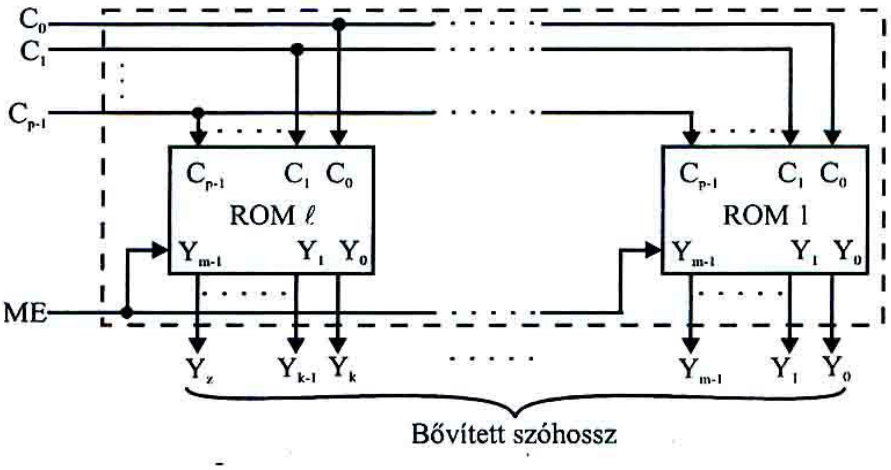
A vezérlőegység feladata, hogy megfelelő vezérlőjeleket biztosítson a kijelölt cella számára, amelyeknek hatására megtörténik a beírás, az olvasás vagy a törlés. A beírás kezdeményezése általában a vezérlőegységre érkező *WE* (*write enable*) jellel történik. A kiolvasáshoz az *OE* (*output enable*) jelet kell aktivizálni. A végzendő műveletre vonatkozó jelek mellett a vezérlőegység általában kap egy *CS* (*chip select*) jelet is, amellyel az egész áramkör engedélyezhető illetve letiltható. Ennek a jelnek általában a memóriák kapacitásának bővítésénél van szerepe.

4.1.3. A kapacitás bővítése

Gyakran az egy tokozásban beszerezhető memória áramkör kapacitása nem kielégítő. Vagy az egyszerre kiolvasható bitek számát (szóhossz) kell megnövelni, vagy a címezhető szavak számát. Ezt több memória áramkör közös nyomtatott lapra történő szerelésével és megfelelő összekapcsolásával érjük el.

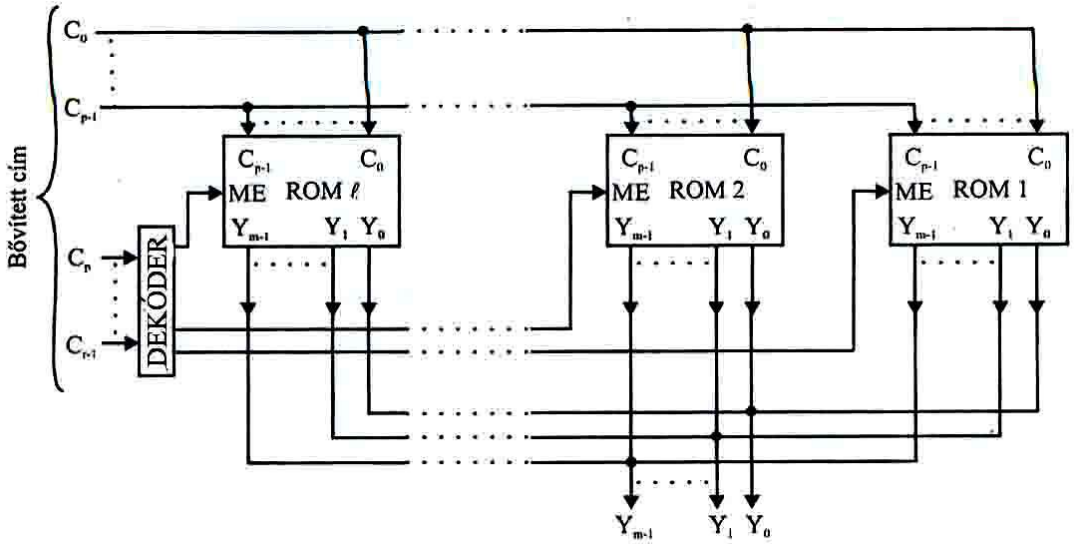
A szóhossz bővítés az egyszerűbb feladat: itt csak felsorakoztatjuk a kellő számú áramkört és az azonos jelzésű címvezetékeiket összekötjük (4-3 ábra). A megfelelő vezérlőbemeneteket is

össze kell kötni. Mivel adott esetben *ROM*-okról van szó, az egyetlen vezérlőjel a kimenetek nyitását végzi. A kimeneti adatvonalakat egy közös adatsin részeinek tekintjük.



4-3 ábra: Szóhossz bővítése ROM-ok összekapcsolásával.

A címezhető szavak számának növelése szintén több memória áramkör összekapcsolásával történik, de itt szükséges egy külső áramkör (dekóder) is amely felváltva aktivizálja az egyes áramköröket (4-4 ábra). Az egyes áramkörök adatvonalai közös adatsinre csatlakoznak. A címvonalak egy része is közös, míg a maradék címvonalakat a dekóderre vezetjük, amely eldönti, hogy melyik áramkört kell adott pillanatban aktivizálni. A bemutatott esetben (*ROM*-ok összekapcsolása) a dekóder az egyes áramkörök kimeneteit engedélyező jeleket (ME) kell hogy létrehozza. *RAM*-ok esetében az *WE* és *OE* jelek közösek minden áramkörre, adott áramkör engedélyezését a dekóder a *CS* jellel végzi.



4-4 ábra: A címezhető szavak számának növelése memória áramkörök összekapcsolásával.

4.2. Aritmetikai egységek

Az első összetettebb digitális berendezések a számítógépek voltak, amelyeket elsősorban a tömeges és bonyolult számítások gyors és pontos elvégzésére fejlesztettek ki. Ma a számítógépek többségének fő feladata nem a számítás, hanem az adatok különböző rendezése de továbbra is jelentős kapacitásokat alkalmaznak számításra.

A gépi számítás alapját egyszerű aritmetikai egységek képezik. Elsősorban kombinációs hálózatokról van szó, amelyek a bemeneti logikai állapotokat bináris számoknak tekintik és

megfelelő logikai hálózattal kiértékelik a kívánt művelet eredményét. A továbbiakban egyszerű összeadó- és szorzó áramkörökről illetve aritmetikai komparátorokról lesz szó.

4.2.1. Összeadók

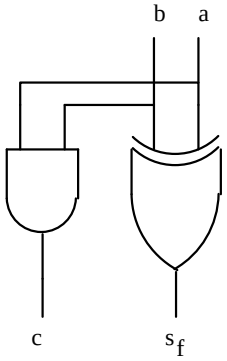
A bináris aritmetikában az összeadók számítanak az alapáramköröknek. A kivonást rendszerint a kettes komplemenssel történő összeadásra vezetik vissza, de gyakran a szorzóáramköröket is összeadók alkalmazásával szerkesztik meg.

Az összetettebb összeadó áramkörök alapegysége a félösszeadó, amely két egybites bináris számot ad össze a 4-5a ábrán megadott táblázat szerint. Az összeg 0,1 és 2 (decimális) értékeket vehet fel, amit két bittel írunk. Ebből a kisebb helyi értékű bit a tulajdonképpeni öszeg (s_f) a nagyobb helyi értékű bit (c) viszont az átvitel a következő helyi értékre.

4-5 ábra: A félösszeadó kombinációs táblázata (a) és logikai rajza (b).

a	b	c	s_f
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(a)



(b)

A táblázatnak megfelelő egyenletek a következők:

$$s_f = \bar{a}b + a\bar{b} = a \oplus b$$

$$c = ab\,.$$

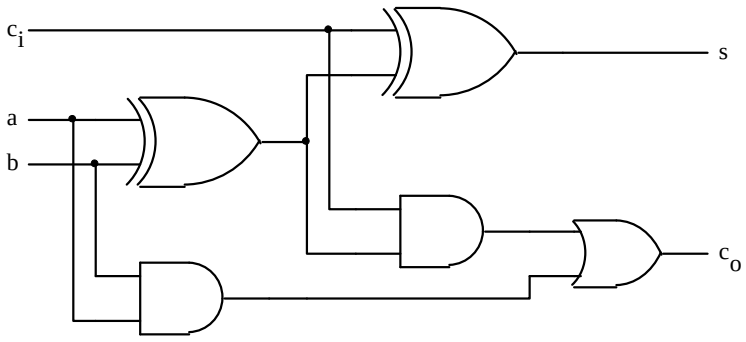
Az egyenletek alapján rajzoltuk meg a 4-5b ábrán a félösszeadót megvalósító kapcsolást.

A félösszeadó hiányossága, hogy nem alkalmas kaszkád kötésre több bites számok összeadása céljából. Ehhez meg kell oldani, hogy az előző helyi értékről érkező átvitelt összeadjuk a következő helyi érték összeadandó bitjeivel. Így jutunk el a teljes összeadó fogalmához. Ebben az esetben a következő egyenletek érvényesek:

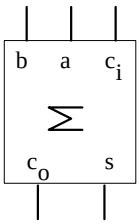
$$s = a \oplus b \oplus c_i$$

$$c_o = (a \oplus b)c_i + ab$$

A teljes összeadó logikai rajzát az egyenletek alapján az 4-6a ábrán adtuk meg, a későbbiekben használatos rajzjel pedig a 4-6b ábrán látható.



(a)

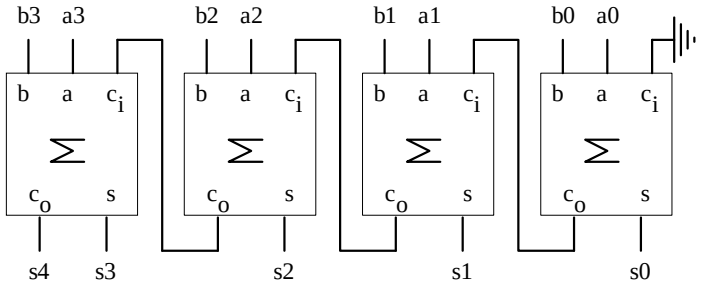


(b)

4-6 ábra: A teljes összeadó logikai rajza (a) és rajzjele (b).

Teljes összeadókból kellő számú kaszkádba köthető a 4-7 ábrán megadott módon, több bites számok összeadása végett. A bemutatott hálózat az összeget párhuzamosan képezi, de az

átvitel az egyes fokozatok között sorosan terjed, ami lényegesen lassítja az áramkör működését. Léteznek olyan áramkörök, amelyek kétfokozatú logikai hálózattal kiértékelik, hogy az adott helyi értéknél várható-e átvitel, ezzel az összeadó sebessége jelentősen növelhető.



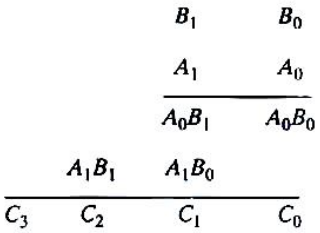
4-7 ábra: Két négybites szám összeadása teljes összeadók kaszkád kötésével.

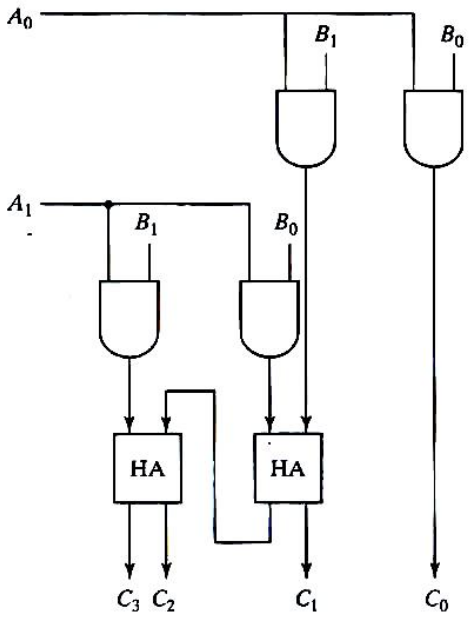
4.2.2. Szorzók

A digitális szorzóáramkörök megvalósítására többféle lehetőség kínálkozik. Visszavezethetjük a szorzást többszöri összeadásra: a szorzó egyes számjegyeivel szorozzuk a szorzandót, majd a részeredményeket összeadjuk. Az ilyen elgondolás sorrendi hálózatot eredményez amely egy logikai automatával vezérli az egyes regiszterek közötti adatáramlást.

Megoldható a szorzás tisztán kombinációs hálózattal is. Ez esetben teljes kombinációs táblázatot kell kialakítani, amely figyelembe veszi a szorzó és a szorzandó számjegyeinek összes variációit és az összes lehetséges eredményt (szorzatot). A sorrendi hálózattal megvalósított szorzók egyszerűbb szerkezetűek (kevesebb logikai kaput tartalmaznak), de a több lépésben történő művelet végzésből eredően lényegesen lassúbbak mint a kombinációs szorzók.

A kombinációs táblázat felírása nélkül is megszerkeszthető a kombinációs alapelven működő-, két bites számot két bites számmal szorzó áramkör (4-8 ábra). A kisebb helyi értékű A0 bittel, két *ÉS* kapu segítségével, megszorozzuk a B szám számjegyeit (bitjeit). A részeredmény kisebb helyi értékű bitje (C0) maga a szorzat legkisebb helyi értékű bitje, a nagyobb helyi értékű bitet viszont egy félösszeadó (HA) segítségével az A1 bit és a B szám szorzatának kisebb helyi értékű bitjével adjuk össze, így keletkezik a szorzat második bitje (C1). A másik félösszeadó az első összeadásból eredő esetleges átvitelt és az A1B1 szorzatot adja össze és képezi a C2 és C3 biteket. Az eljárás helyességét a logikai rajz mellett közölt írásbeli szorzás igazolja.

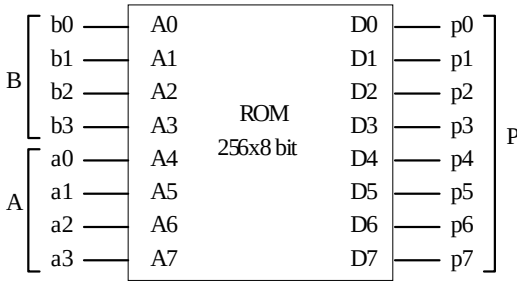




4-8 ábra: Kétbites számot kétbites számmal szorzó kombinációs hálózat.

Kettőnél több bites számok szorzóáramkörének megvalósítása mind több logikai kaput igényel. Egy áthidaló megoldásnak tekinthető a *ROM*-mal történő szorzás. A 4-9 ábra szerint a 256x8 bit kapacitású *ROM* megfelelő programozással elláthatja a 4x4 bites szorzó szerepét. A

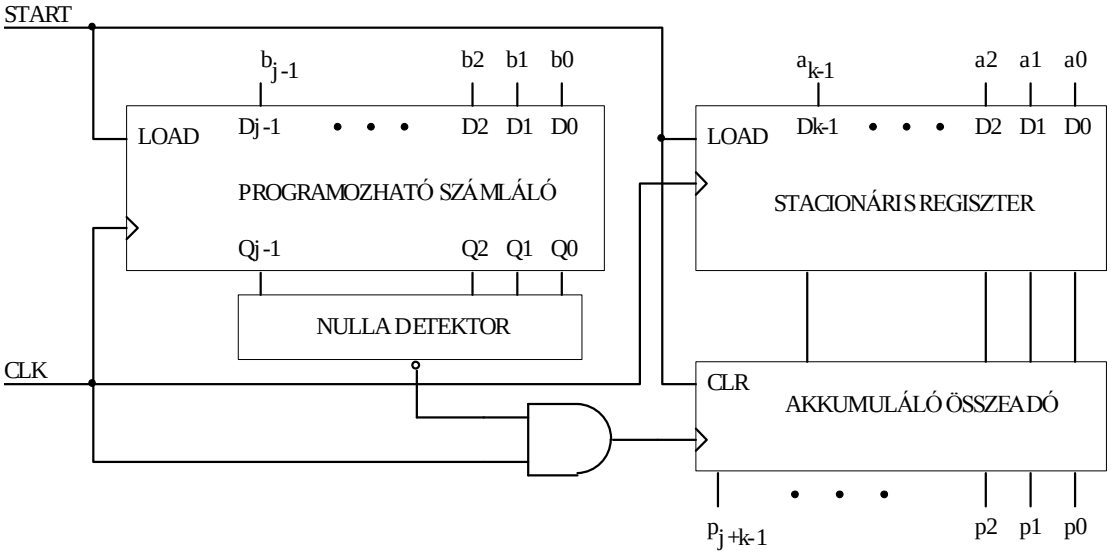
szorzandó (*A*) és a szorzó (*B*) megfelelő bitjeit a címvonalakra kötöttük. A *ROM* tartalmát egy táblázatnak tekintjük, amely tartalmazza az összes lehetséges szorzatokat (P).



4-9 ábra: Szorzóáramkör megvalósítása ROM-mal.

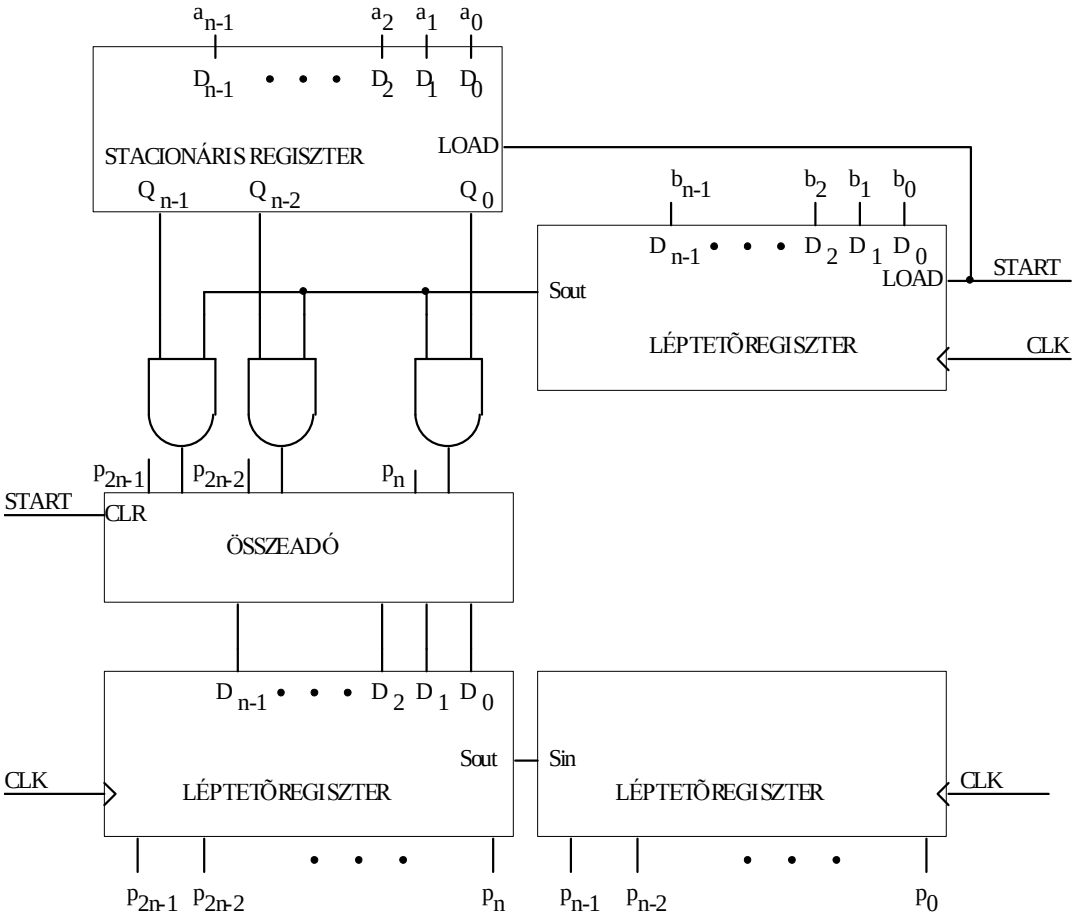
A *VLSI* technikában, elsősorban mikroprocesszorok belső egységeiként léteznek 16x16 bites- vagy ennél nagyobb szorzóáramkörök is kombinációs hálózat formájában.

A sorrendi hálózatként megvalósított szorzóáramkörökkel kapcsolatban két alapelvet említünk meg. A 4-10 ábrán látható hálózat a *START* jel hatására törli az akkumulátor tartalmát, a stacionáris regiszterbe beírja a szorzandót, a programozható számlálóba pedig betölti a szorzót. Az órajel hatására számláló visszafelé számlál, ugyanakkor az akkumuláló összeadó minden alkalommal összeadja az akkumulátor pillanatnyi tartalmát a szorzóval. A folyamat addig tart, amíg a számláló nulláig nem ér, ekkorra az akkumulátorban a többszöri összegzésnek köszönhetően kialakul a szorzat.



4-10 ábra: Szorzás visszavezetése többszöri összeadásra.

A másik alapelv a kézi szorzásnál alkalmazott bitenkénti eljárásra épül (4-11 ábra). A folyamat kezdetén a szorzandót a stacionáris regiszterbe, a szorzót pedig a léptető regiszterbe töltjük be. A szorzó adott bitjével való szorzást az *ÉS* kapuk sora végzi. Az összeadó, megfelelő léptetés mellett, összeadja az eddigi tartalmat a pillanatnyilag generált részeredménnyel. A szorzás akkor fejeződik be, amikor a léptetőregiszterből kiléptettük a szorzó minden bitjét.



4-11 ábra: Bitenkénti szorzást megvalósító sorrendi hálózat.

4.2.3. Aritmetikai komparátorok

Két bináris szám összehasonlítása gyakori feladat a digitális rendszerekben. Az összehasonlítást végző áramkör neve (aritmetikai) komparátor. Az összehasonlítás eredménye lehet pusztán annak megállapítása, hogy a két szám egyenlő-e vagy sem, de egyenlőtlenség esetén eldönthető az is, hogy melyik szám a nagyobb illetve a kisebb. A továbbiakban csak ezzel az összetettebb, úgynevezett univerzális komparátorral foglalkozunk.

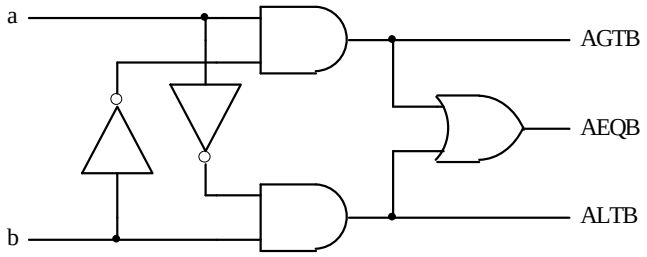
Két egybites szám összehasonlítása a 4-12a ábrán megadott kombinációs táblázat szerint történik. Az egyes kimeneti változókat a következő egyenletek szerint képezzük:

$$AGTB = a\bar{b}$$
$$AEQB = ab + \bar{a}\bar{b} = \overline{a \oplus b}$$
$$ALTB = \bar{a}b,$$

ahol az AGTB változó akkor egyes ha a>b, AEQB akkor egyes, ha a=b, ALTB pedig akkor egyes, ha a<b. Az egyenletek alapján megvalósított egybites univerzális komparátor logikai rajzát a 4-12b ábrán adtuk meg.

A	B	AGTB	AEQB	ALTB
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

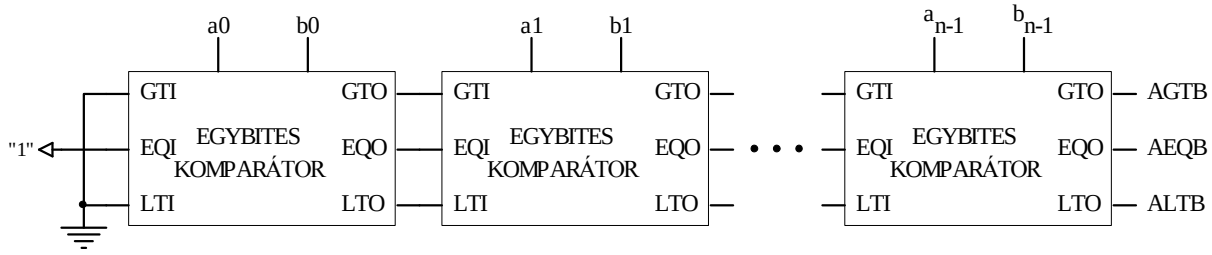
(a)



(b)

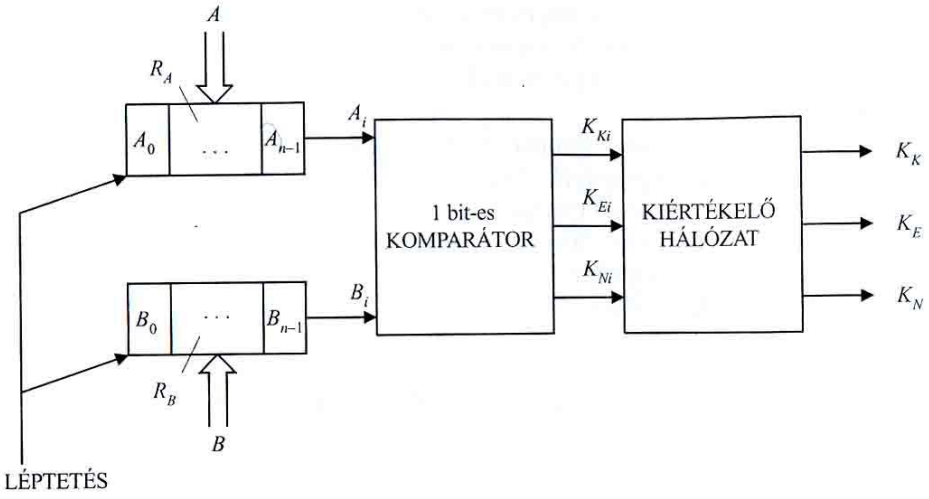
4-12 ábra: Egybites univerzális komparátor kombinációs táblázata (a) és logikai rajza (b).

Az igazi cél a több bites számok összehasonlítását végző áramkör kialakítása. Elvileg elképzelhető, hogy több bit esetére is felírjuk a 4-12a ábrán megadott módon a több bites komparátor táblázatát, majd ez alapján elvégezzük a megfelelő hálózat szintézisét. Sokkal célszerűbb azonban a bitenkénti összehasonlítás, ennek érdekében a 4-12b ábrán bemutatott hálózatot úgy kell módosítani, hogy az egyes fokozatok kaszkád kötése lehetséges legyen. Az egybites komparátorok kaszkád kötésének módját a 4-13 ábra mutatja. A működési sebesség korlátozott, de az áramkör viszonylag egyszerű.



4-13 ábra: Aritmetikai komparátorok kaszkád kötése.

A 4-14 ábra szerint az összehasonlítás végezhető soros eljárással is, megfelelő sorrendi hálózattal. A regiszterekbe betöltött értékeket fokozatosan léptetjük a komparátor bemenetére. Az összehasonlítást a nagyobb helyi értékek felől célszerű kezdeni. Amint különbség mutatkozik, valamelyik bitnél, az A>B illetve az A<B kimenetek azonnal aktivizálhatók. Az egyenlőség csak akkor állapítható meg, ha minden bitet kivizsgáltunk.



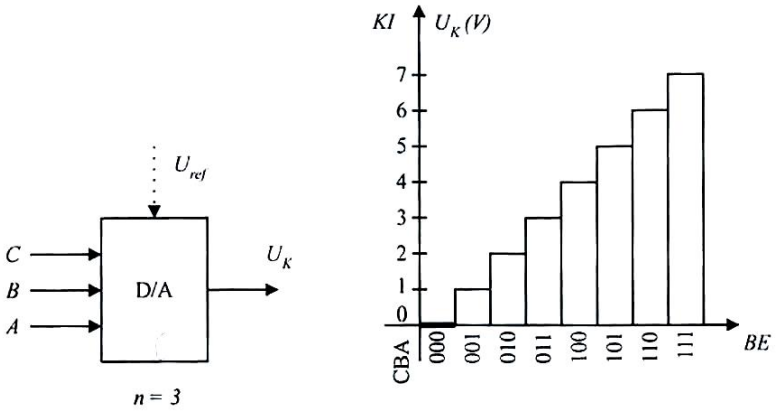
4-14 ábra: Soros üzemű aritmetikai komparátor logikai rajza.

4.3. D/A átalakítók

Különböző műszaki feladatok megoldásánál szükség mutatkozik a digitális jelek (számadatok) analóg jellé történő átalakítására. A digitális/analóg (*D/A*) átalakítás során rendszerint a számokkal arányos feszültséget képezünk.

4.3.1. Működési elv

A *D/A* átalakító minden egyes bemeneti számhoz egy diszkrét feszültségértéket rendel hozzá. A lehetséges feszültségértékek rendszerint egy skálán lineárisan helyezkednek el. Az értékek száma-, így a skála finomsága is az adott kódrendszerben kódolható számértékek számától függ. Rendszerint természetes bináris kódot alkalmazunk, így az *n* bites kód 2^n kimeneti feszültségértéket definiál. A 4-15 ábrán hárombites bináris kóddal működő *D/A* átalakító rajzjelét és átviteli diagramját láthatjuk. Általában a bitek száma lényegesen nagyobb pl. 8, 10, 12 vagy annál is több.

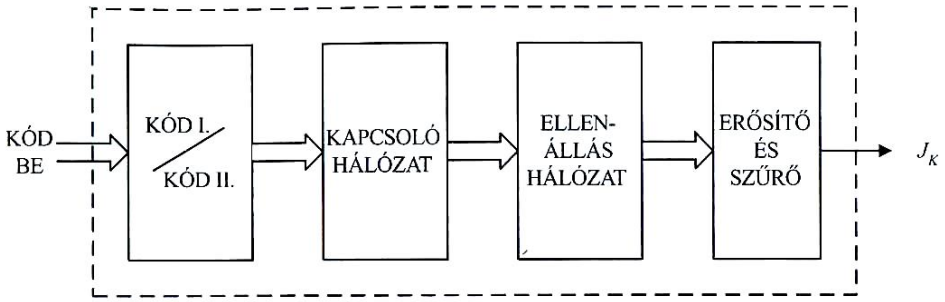


4-15 ábra: D/A átalakító rajzjele (a), és átviteli diagramja (b).

A kimeneti feszültséget a *D/A* átalakító megfelelő alapjelből (referens feszültség) képezi leosztással. A pontos átalakítás feltétele, hogy az alapjel pontos és stabil legyen, ugyanakkor a leosztásban sem vétsünk hibát.

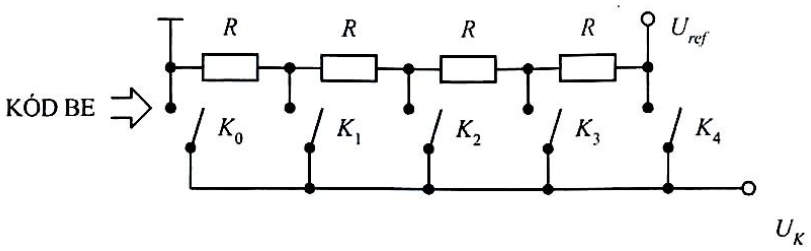
4.3.2. Felépítés

A *D/A* átalakítók központi része egy ellenálláshálózat (4-16 ábra), amely az alapjel leosztását végzi. Az osztási arányt vezérelt analóg kapcsolókkal állítjuk be. A kapcsolók vezérlését végezhetik közvetlenül a bemeneti kódolt szám bitjei, de megtörténhet, hogy kódátalakítást kell végezni. Az ellenálláshálózat kimeneti jele használható közvetlenül is, de általában erősítő és szűrő fokozatokat iktatunk közbe.



4-16 ábra: Digitális/analóg átalakító tömbvázlata.

A 4-17...4-20 ábrákon az ellenálláshálózat és a kapcsolók jellemző elrendezéseit mutattuk be. Az úgynevezett direkt átalakítónál (4-17 ábra) az ellenálláshálózat azonos értékű ellenállások soros kötéséből áll. Az így kapott osztó lecsapolásain előállítjuk az összes lehetséges diszkrét feszültségértéket. A kapcsolók közül mindig csak egyet kapcsolunk be, a bementi számkódnak megfelelően, így kerül a kiválasztott feszültségérték a kimenetre. Vagy a kódrendszert kell úgy megválasztani, hogy minden kódban csak egy egyes legyen, vagy ezt a feltételt megfelelő dekóderrel, esetleg kódátalakítóval valósítjuk meg. A hálózat elvileg egyszerű, de nagy felbontás esetére nagy számú kapcsolót és nagy számú, pontos értékű ellenállást tartalmaz, amelyek megvalósítása költséges.

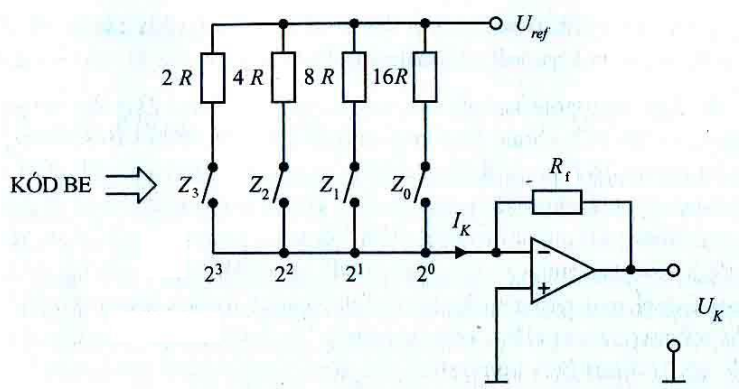


4-17 ábra: Direkt típusú D/A átalakító elvi rajza.

A súlyozott ellenálláshálózattal működő *D/A* átalakító elvi rajzát az 4-18 ábrán láthatjuk (négy bites kód esetére). A referens feszültségforrásból az ellenállásokkal képzett áramok a műveleti erősítő bemenetén összeadódnak és a kimeneten a számkódnak megfelelő feszültséget képeznek:

$$V_O = -R_f \cdot V_{REF} \cdot \frac{1}{2^n R}(2^0 Q_0 + 2^1 Q_1 + 2^2 Q_2 + ... + 2^{n-1} Q_{n-1})$$

ahol a Q_i értékek a bemeneti kód egyes bitjei. Ennél a megoldásnál a fő nehézség a súlyozott ellenállásértékek pontos megvalósítása. Az integrált technikában különösen nehéz nagyon nagy és nagyon kicsi ellenállásértékeket megfelelő tűréssel megvalósítani. Gondot okoz az is, hogy az egyes kapcsolókon áthaladó áramok nagyon különbözőek, így a kapcsolókon eső feszültség is különbözik, ami hibát okoz az átalakításban.

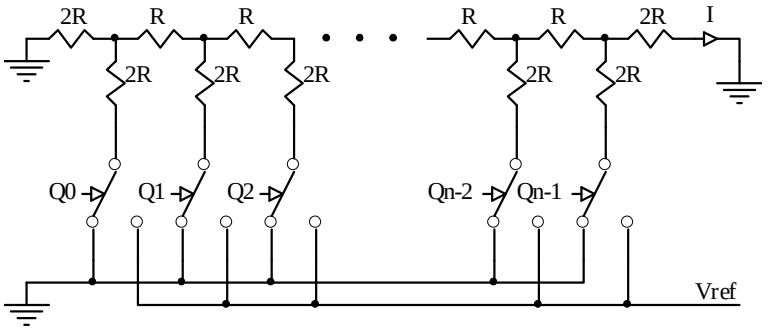


4-18 ábra: Súlyozott ellenálláshálózattal működő D/A átalakító.

Az integrált technikában a létrahálózatokkal működő *D/A* átalakítók váltak be, mivel csak két ellenállásértéket (*R* és *2R*) kell precízen reprodukálni (4-19 ábra). Minden egyes kapcsoló referens feszültségre való kötésével egy-egy áramkomponenst lehet létrehozni a kapcsolás jobb oldalán bejelölt *I* kimeneti áramban. A kimenethez közelebb álló kapcsolók nagyobb áramot hoznak létre, ahogy haladunk balra, az egyes kapcsolók hatása a kimeneti áramra feleződik az előző fokozathoz képest. Az eredő áram a következő képlettel adott:

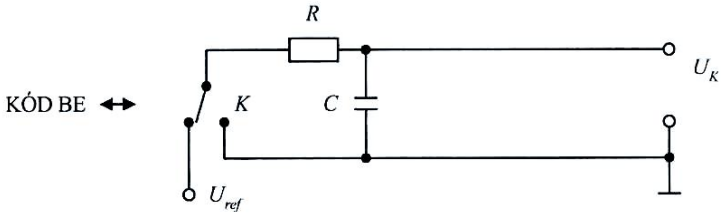
$$I = \frac{V_{REF}}{6R} \cdot \frac{1}{2^{n-1}} \cdot (2^{n-1} Q_{n-1} + 2^{n-2} Q_{n-2} + ... + 2^2 Q_2 + 2^1 Q_1 + 2^0 Q_0)$$

ahol Q_i a bemeneti számkód bitjei. Látható tehát, hogy az áram ebben az esetben is a bemeneti bináris kóddal arányos. Ez az áram közvetlenül is felhasználható mint kimeneti jel, de szükség szerint egy műveleti erősítővel feszültséggé alakítható.



4-19 ábra: R-2R létrahálózattal működő D/A átalakító.

A legkevesebb alkatrésszel megvalósított *D/A* átalakító megoldást az 4-20 ábrán szemléltetjük. A *K* kapcsoló periódikusan változtatja helyzetét. A kapcsolgatást úgy oldjuk meg, hogy a bal állásban töltött idő aránya a periódushoz megegyezzen az átalakító bemenetére vezetett pillanatnyi kódolt számérték és a maximális lehetséges számérték arányával (ezt hívják impulzus-szélesség modulációnak). Megfelelő szűrés után (ezt az *RC* tag végzi) a kimeneten kapott feszültségérték (az impulzusok átlagértéke) arányos lesz a bemeneti számértékkel.



4-20 ábra: Impulzus-szélesség modulációval működő D/A átalakító.

4.3.3. Jellemzők

A *D/A* átalakítók fő jellemzői a felbontás és a beállási idő. A felbontást általában a bitek számával adják meg, bináris kódot feltételezve a bemeneten. A felbontás egyben utal az átalakítás pontosságára is. Az átalakítókat úgy kell megszerkeszteni, hogy az átviteli diagram monoton legyen. Ezzel feltételeztük, hogy a legkisebb helyi értéknek megfelelő feszültségnél nagyobb hibát nem véthet az átalakító.

A kimeneti feszültség stabilizálódása bizonyos időt vesz igénybe. Az átalakítók többségénél ez az idő μs vagy *ns* nagyságrendű, míg az impulzus-szélesség modulációval működő átalakítónál lényegesen hosszabb időről van szó, mivel az *RC* tag időállandója többszöröse a kapcsolási periódusnak, amely önmagában is viszonylag hosszú.

4.4. A/D átalakítók

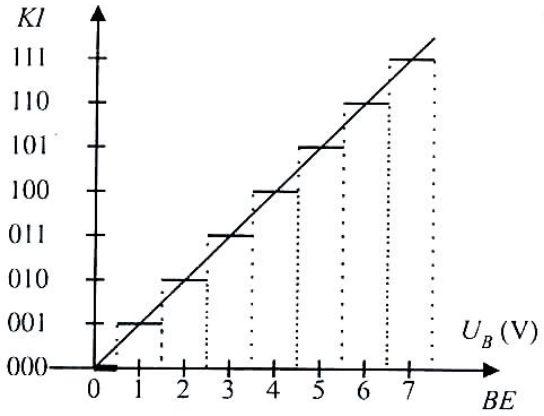
Ha az analóg jeleket (feszültségértékek vagy más folyamatosan változó mennyiségek) digitális formában kívánjuk tárolni, feldolgozni, továbbítani, megjeleníteni stb., el kell végezni az analóg-digitális átalakítást. Az átalakítás során az analóg jelnek megfelelő kódolt számot képezünk.

4.4.1. Működési elv

Az *A/D* átalakítás során több feladatot kell megoldani. Elsőként az átalakítandó analóg jelből szabályos időközökben (periódikusan) mintát kell venni. Ezt a műveletet nevezik idő szerinti diszkretizációnak. Az átalakítási folyamat időt vesz igénybe, a mintavételezés periódusát úgy kell meghatározni, hogy az átalakítás befejeződjön a következő minta beérkezése előtt.

A második feladat a minta összehasonlítása egy diszkrét értékskála adataival. Az összehasonlítás eredményeként a minta értékét a skála egy megfelelő értékével (rendszerint a legközelebbivel) helyettesítjük. Ezt nevezzük amplitúdó szerinti diszkretizációnak.

Az átalakítás utolsó fázisa a diszkretizált értéknek megfelelő kód képzése. Az *A/D* átalakító ezt a folyamatot az előre definiált átviteli karakterisztika (4-21 ábra) alapján végzi.

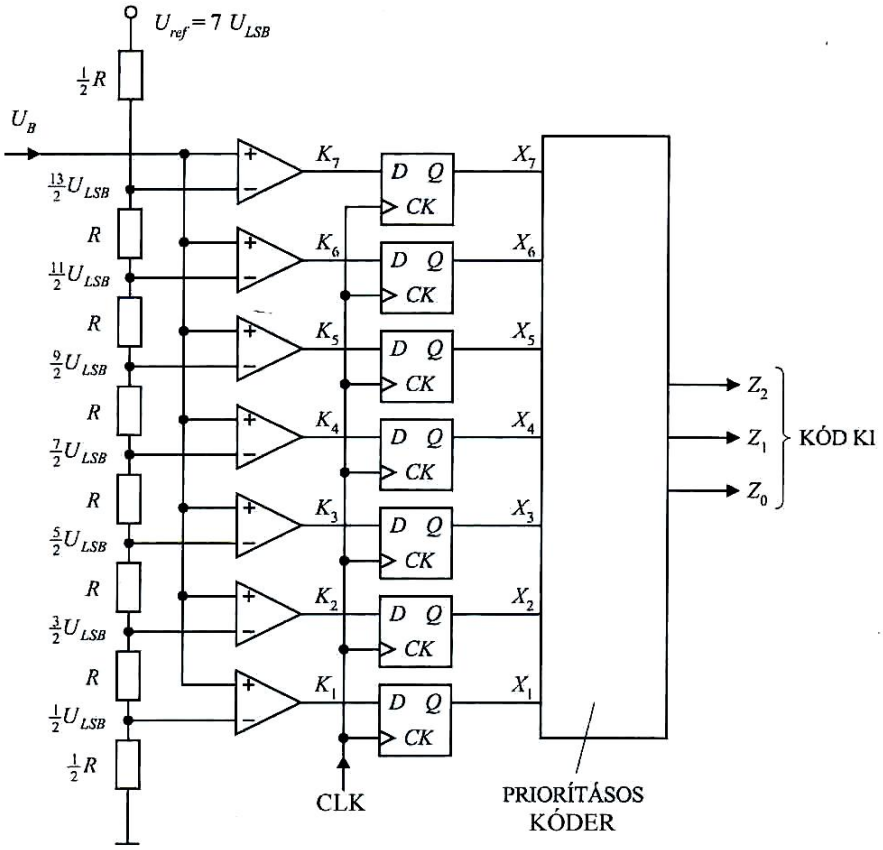


4-21 ábra: Három bites A/D átalakító átviteli karakterisztikája.

4.4.2. Felépítés

Ma az *A/D* átalakítóknak három különböző felépítése ismeretes. Ezek a direkt (*flash*) típus, a fokozatos közelítéses (szukcesszív approximációs) megoldás és a számlálós eljárás.

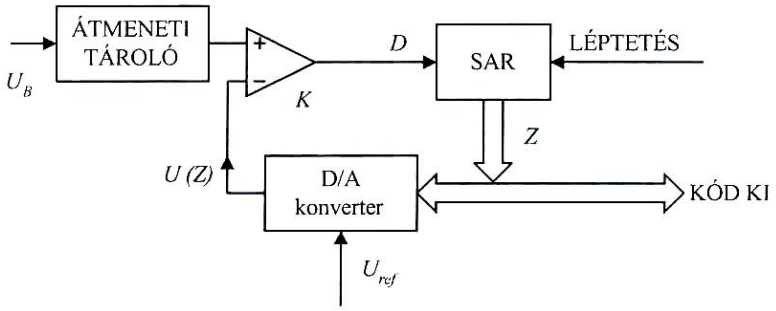
A direkt *A/D* átalakító *n* bites kód esetére 2^n*-1* darab feszültségkomparátort tartalmaz (4-22 ábra). A komparátorok egyik bementére az átviteli karakterisztikának megfelelő küszöbfeszültségeket kötünk. Ezeket a feszültségeket ellenállásláncal állítjuk elő a referencia feszültségből, így alakul ki a kívánt diszkrét értékskála. A komparátorok másik bementére magát az átalakítandó analóg feszültségértéket kötjük.



4-22 ábra: Direkt A/D átalakító három bites kimenettel.

A feszültségértéktől függően egyes komparátorok kimenete alacsony-, másoké magas logikai szintet fog adni. Az idő szerinti diszkretizációt a *CLK* órajel végzi: a komparátorok kimeneti állapotait periódikusan átírja a tárolókba. A kimeneti fokozat egy prioritásos kóder: csak a legmagasabb pozíción levő logikai egyest figyelembe véve alakítja ki a kimeneti kódot.

A fokozatos közelítéses *A/D* átalakító *n* lépésben alakítja ki az *n* bites kódot. A működési elvet az 4-23 ábra szemlélteti. A mintavételező áramkör egy átalakítási periódus időtartamára rögzíti a *K* komparátor bemenetére vezetett értéket. Az átalakító tartalmaz egy azonos felbontású *D/A* átalakítót, ennek a kimenete van a komparátor másik bementére kötve. A komparátor kimenete a kapcsolás központi részét képező fokozatos közelítéses regisztert (successive approximation register – SAR) vezérli. Ugyanakkor a regiszter órajel bemenete is vezérlést kap.



4-23 ábra: A fokozatos közelítéses A/D átalakító elvi rajza.

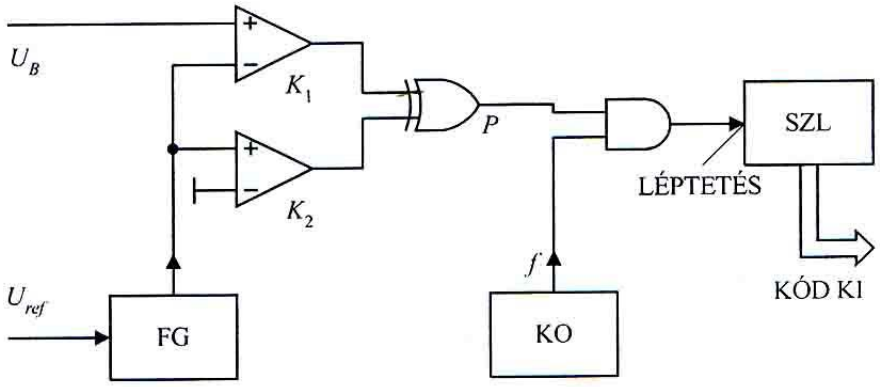
Az átalakítási folyamat kezdetén a regiszter a legnagyobb helyi értékű bitjét (*MSB*) logikai egyesre állítja. A *D/A* átalakító ennek megfelelő analóg jelet állít elő, ami alapján a komparátor értesíti a regiszter, hogy az analóg jel a teljes értékskála felénél kisebb vagy nagyobb értékű-e.

Ha az analóg feszültség nagyobb, az órajel megfelelő élénél az *MSB* marad az egyes értéken, ellenkező eseben nullára állítódik. Ugyanakkor a regiszter következő helyi értékű bitje állítódik egyesre. A *D/*A átalakító kimenentét a bemeneti értékkel összehasonlítva a komparátor újra informálja a regisztert, hogy helyes volt-e az újabb bit egyesre állítása. Ha igen, az egyes megmarad, ha nem, törlődik, de a következő kisebb helyi értékű bit egyesre állítódik az újabb órajel ciklus idejére.

Ez a folyamat addig folytatódik, amíg minden bitet meghatározunk. Ekkor kell kiolvasni a regiszter tartalmát, ez képezi az analóg értéknek megfelelő digitális kódot. Az egész áramkör beszerezhető integrált formában egy tokozásban.

Az A/D átalakítók megoldásánk harmadik csoportját a számlálós átalakítók képezik. Többféle számlálós megoldást fejlesztettek. Az alapelv minden esetben az analóg feszültséggel arányos időtartamú négyszögimpulzus létrehozása. A számláló megszámlálja, hogy a négyszögimpulzus időtartama alatt hány órajel ciklus játszódik le. A számlálás eredményét képező kód arányos lesz az analóg feszültséggel.

A működési elvet az 4-24 ábra szemlélteti. A négyszögimpulzust az analóg feszültség és egy lineárisan növekvő fűrészfeszültség összehasonlításával kapjuk. A feszültségkomparátorok (a kizáró *VAGY* kapu segítségével) addig engedélyezik a számláló működését, amíg a fűrészfeszültség nulla felett van, de még nem haladta meg a bemeneti értéket.



4-24 ábra: Fűrészjellel és számlálóval működő A/D átalakító elvi rajza.

A bemutatott megoldás hátránya, hogy az átalakítás pontosságára kihat a fűrészjel meredekségén kívül az órajel frekvenciájának pontossága is. A fűrészjel meredekségét megfelelő referenciafeszültséggel stabilizálni lehet, de további gondok lehetnek a fűrészgenerátort megvalósító RC elemek csúszása miatt. Léteznek olyan számlálós megoldások, amelyek emelkedő és eső szakaszt tartalmazó fűrészfeszültséggel működnek. Ezzel a technikával ki lehet küszöbölni úgy az RC elemek változásaira való érzékenységet, mint az órajel frekvenciára való érzékenységet. Egyedül a referens feszültség pontos szintentartása a fontos.

4.4.3. Jellemzők

A direkt típusú *A/D* átalakítónál az átalakítási folyamat rendkívül gyors (általában *1μs* alatti az átalakítási idő), de nagy az alkatrészigénye, így költséges. Általában csak hat-nyolc bites felbontásig alkalmazzuk ezt a megoldást, akkor is csak gyorsan változó jelek átalakítására (digitális oszcilloszkópok, híradástechnika).

A fokozatos közelítéssel működő átalakítók közepes sebességűek, az átalakítás rendszerinyt néhány *μs*-ig tart, a felbontás rendszerint nyolc bit vagy az feletti. Leginkább gyors folyamatok vezérlésénél alkalmaznak ilyen megoldást.

A fűrészjellel és számlálóval működő átalakítókkal nagy felbontás érhető el viszonylag kevés alkatrész alkalmazásával, olcsó áron. Hátrány, hogy az átalakítási folyamat lassú, gyakran másodperc nagyságrendű időt vesz igénybe. Digitális kéziműszereknél mindig ilyen átalakítót építenek be, mivel a felhasználó nem is tudná leolvasni a kijelzett értéket, ha a számjegyek gyorsan változnának. Lassan változó fizikai mennyiségek (pl. hőmérséklet, vízszint stb.) digitalizálására is alkalmas ez az *A/D* átalakító megoldás.

III. Tervezés programozható logikai áramkörökkel (PLD)

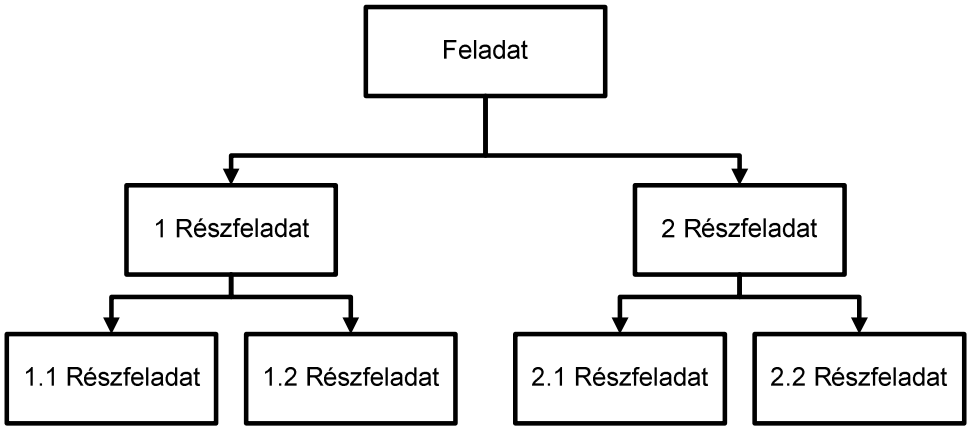
A *Verilog* hardver leíró nyelv

5. A tervezés menete

A jegyzetnek ebben a részében a *Verilog* hardver leíró nyelven (angolul: *hardware description language – HDL*) alapuló digitális tervezéssel foglalkozunk. Mielőtt áttérnénk a nyelvi szerkezetek ismertetésére, általánosságban áttekintjük a tervezési munkáknál alkalmazott hozzáállásokat.

- ***Top-down* irány:**

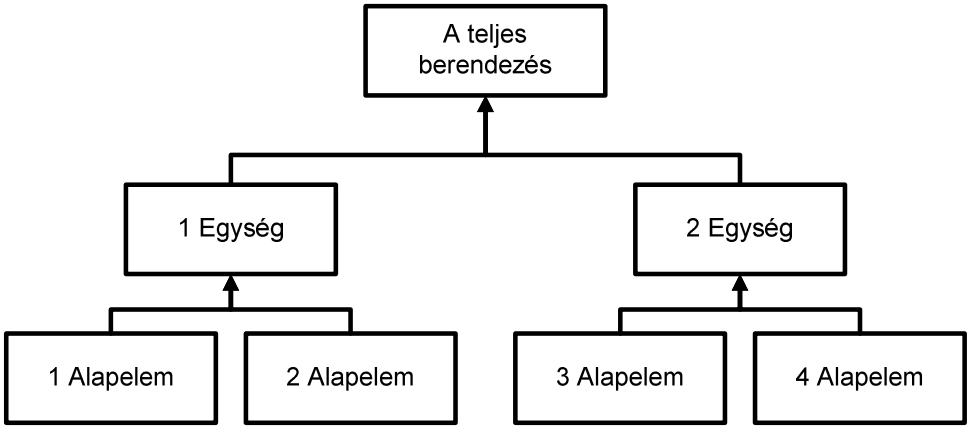
A tervezés tárgyát képező összetett feladatot részekre bontjuk, majd ezeket a részfeladatokat újabb részfeladatokra stb. Az eljárást addig folyattjuk, amíg a részfeladatok nem lesznek elég kicsik és egyszerűek a megoldásra. A 5-1 ábra a *top-down* (felülről lefelé történő) tervezés menetét szemlélteti.



5-1 ábra: Felülről lefelé történő (top-down) tervezési menet.

- ***Bottom-up* irány:**

Ez az eljárás a berendezés megépítéséhez szükséges alapelemekből indul ki. Az alapelemek kombinálásával egyszerű funkcionális egységeket alakítunk ki. Az így kapott egységeket tovább kombináljuk és összetettebb egységeket kapunk. Az eljárást addig folytatjuk, míg ki nem alakul a teljes berendezés terve. A 5-2 ábra a *bottom-up* (alulról felfelé történő) tervezés menetét szemlélteti.



5-2 ábra: Alulról felfelé történő (bottom-up) tervezési menet.

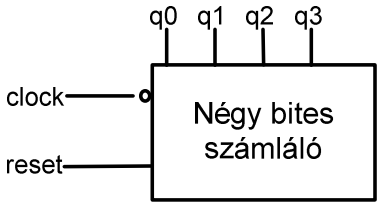
Mindkét esetben bizonyos hierachia van jelen. A digitális áramkörök tervezésénél rendszerint ötvözzük ezt a két módszert. A tervezők a tervezési feladatot részfeladatokra bontják, majd ezeket a részfeladatokat kisebb feladatokra, míg el nem érnek addig a szintig, ahol már a részfeladatok megoldhatók előregyártott alkatrészekkel, illetve ahol már a részfeladatok az adott fejlesztői környezet alapelemei.

Másrészt, a digitális funkcionális egységek tervezői félvezető kapcsolókból indulnak ki, amelyek az illető technológián belül alapelemnek számítanak. A kapcsolókból először egyszerű funkcionális egységeket (pl. logikai áramkörök) alakítanak ki, majd ezeket kombinálva alakulnak ki a bonyolultabb egységek, amelyek az adott fejlesztői környezetben kiinduló alkatrészekként jelentkeznek.

Az elmondottak egyaránt érvényesek a SSI és MSI áramkörökkel kapcsolatos tervezésekre és a *PLD*-s tervezésekre is. Az ismertetett tervezési eljárásokat egy négy bites aszinkron számláló tervezésével illusztráljuk. A számlálót úgy tervezzük meg, hogy alacsony logikai szinttel aszinkron reszetet lehessen rá alkalmazni, ugyanakkor az órajel bemenet a lefutó élre reagáljon.

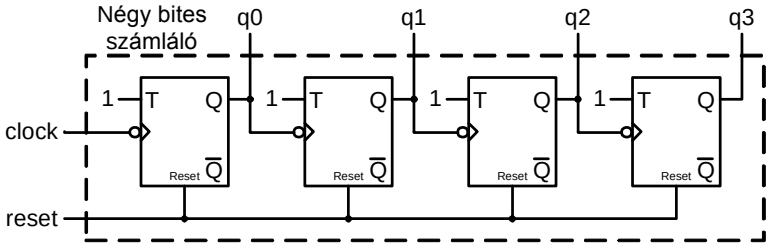
5.1. Négy bites számláló

A hierarchikus tervezést bemutató számlálónak négy kimenete lesz (*q0*, *q1*, *q2* és *q3*) és két bemenete (*clock* és *reset*), ahogyan azt a 5-3 ábrán láthatjuk.



5-3 ábra: A négy bites számláló rajzjele.

Tételezzük fel, hogy a fejlesztői környezetben a számláló megvalósítására *D flip-flop*-ok és megfelelő logikai áramkörök állnak rendelkezésre. A négy bites számláló egy lépésben történő megtervezése *D flip-flop*-okból és logikai kapukból túl nehéz feladatnak bizonyulhat. Ugyanakkor *T flip-flop*-ok felhasználásával a tervezés meglehetősen leegyszerűsödik. A 5-4 ábrán láthatjuk a négy bites számláló *T-flip-flop*-okból történő megvalósításának módját.

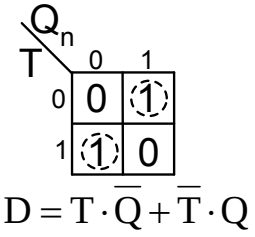


5-4 ábra: A négy bites számláló megvalósítása T flip-flop-okkal.

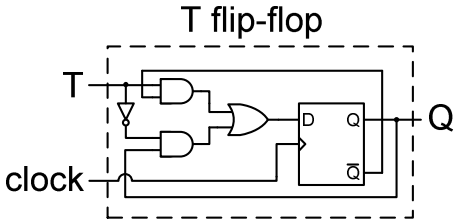
A következő lépés a számláló tervezésénél a *T flip-flop*-ok megvalósítása *D flip-flop*-ok és logikai kapuk segítségével. A *T flip-flop*-nak *D flip-flop*-pal történő megvalósítása a 5-1 táblázat szerint történik. A 5-5 ábrán láthatjuk a *D* bemenetet megvalósító logikai függvény minimalizálásához szükséges *Karnaugh* táblát (nincs minimalizációs lehetőség). A 5-6 ábrán adtuk meg a *T flip-flop*-nak *D flip*-flop-pal és logikai áramkörökkel történő megvalósításának logikai rajzát.

5-1 táblázat: T flip-flop-nak D flip-flop-pal történő megvalósításához szükséges állapottáblázat.

T	**Q_n**	**Q_{n+1}**	**D**
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

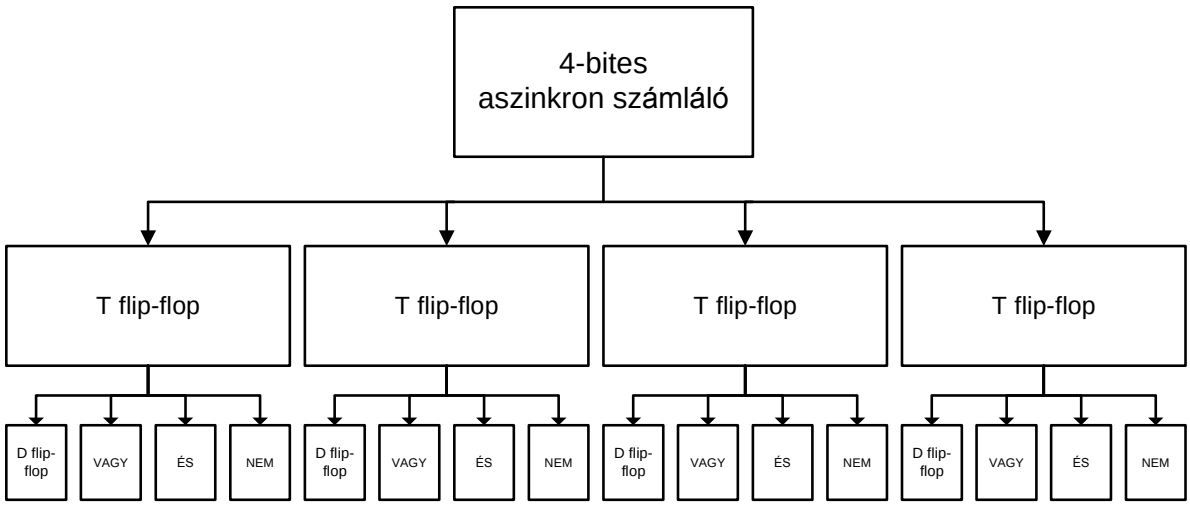


5-5 ábra: A D flip-flop bemeneti logikai függvényének minimalizálására szolgáló Karnaugh tábla



5-6 ábra: T flip-flop-nak D flip-flop-pal és logikai áramkörökkel történő megvalósításának logikai rajza.

A feladat megoldásánál felülről lefelé haladó tervezést alkalmaztunk. A végeredményként kapott hierarchikus szerkezetet a 5-7 ábra tömbvázlata szemlélteti.



5-7 ábra: A négy bites aszinkron számláló felülről lefelé történő tervezésénél kapott hierarchikus szerkezet.

A négy bites számláló felülről lefelé történő tervezésénél az első lépés a legfelső tömb definiálása volt, itt írtuk le a tervezendő számláló működését. A következő szinten a számlálót *T flip-flop*-okkal valósítottuk meg. A harmadik szinten az egyes *T flip-flop*-okat *D flip-flop*-ok valamint *VAGY*, *ÉS* és *NEM* logikai kapukkal valósítottuk meg. Ezen a szinten kerül kapcsolatba a felülről lefelé és az alulról felefelé történő tervezés: a fejlesztői környezet alapelemeiként alkalmazott *D flip-flop*-ok és logikai áramkörök egy korábbi alulról felfelé történő tervezés eredményei.

Ahelyett, hogy a D flip-flop-okat alapelemeknek tekintettük, logikai kapuk segítségével megvalósíthatjuk őket. Így a felülről lefelé történő tervezési hierarchiában még egy szintet vezetünk be. Ekkor a felülről lefelé és az alulról felfelé történő tervezés tisztán a logikai kapuk szintjén találkozik.

5.2. Modulok a Verilog HDL-ben

A *Verilog* hardver leíró nyelv (angolul: *hardware description language – HDL*) a modul (angolul: *module*) fogalmának bevezetésével hierarchikus tervezést tesz lehetővé. A modulok a

Verilog HDL építőelemei, amelyek alapelemekből (angolul: *primitive*) és/vagy egyéb alacsonyabb rendű elemekből állnak.

A modul definiálásának módja lehetővé teszi, hogy a modult alkotó elemek összessége több helyen is használható legyen egy projektumon belül. A modulok a környezetükhöz a *port*-okon (bemenetek, kimenetek) keresztül kapcsolódnak. A *port*-oknak köszönhetően a modul belseje rejtve van a környezete számára. A környezetből szemlélve a modulból csak a bemeneti és a kimeneti vonalak (*port*-ok) láthatók. Ez lehetővé teszi, hogy változtatásokat eszközöljünk a modul belső szerkezetén, de közben nem kell változtatni más tervezési egységeket. Ugyanakkor lehetővé válik, hogy feladatot részfeladatokra bontsuk és több tervező párhuzamosan dolgozzon (csapatmunka).

A csapatmunka feltétele, hogy a projektumot hierarchikusan szervezzük meg, definiáljuk az egyes modulok szerepét és *port*-jait. Ezt követően az egyes modulok vagy modulcsoportok tervezését a csapat egyes tagjaira bízzuk.

A 5-7 ábrán a négy bites számlálót és a *T flip-flop*-ot tekinthetjük modulnak.

A *Verilog* hardver leíró nyelvben a modul definíciója a *module* kulcsszóval kezdődik és az *endmodule* kulcsszóval végződik. Minden modulnak egyedi nevet kell adni, hogy meg tudjuk különböztetni más moduloktól. szükség szerint a modul tartalmazza a *port*-ok listáját, amely definiálja a modul kimenő és bemenő vonalait.

A modul definíciója a *Verilog HDL*-ben tehát az 5-1 példa szerint fog alakulni:

5-1 példa: A modulok definiálásánál használatos legfontosabb elemek.

```
module <a_modul_neve> (<port-ok_listája>);

<a_modul_teste>

endmodule
```

A modul testének nevezett központi részben négy különböző elvonatkoztatási szinten adhatók meg a különböző definíciók. Az egyes szinteket szükség szerint használjuk. Ugyanaz a feladat megfogalmazható bármelyik elvonatkoztatási szinten, a különböző szintű leírások keveredhetnek is.

Az elvonatkoztatási (absztrakciós) szintek a következők:

- **Viselkedési- vagy algoritmikus szint (angolul: *behavioral* or *algorithmic level*)**:
  Ez képezi a legmagasabb szintű elvonatkoztatást a *Verilog HDL*-ben. Ezen a szinten a súlypont a feledat elvégzésére alkalmas algoritmus kialakítása, a hardvermegoldással nem foglalkozunk (feltételezzük, hogy a logikai szintézist megfelelő szoftver fogja elvégezni). A tervezés nagyon hasonlít a *C* szoftvernyelvben történő programozáshoz).
- **Adatfolyam szint (angolul: *dataflow level*)**:
  Ezen a szinten az egyes regiszterek közötti adatáramlást- és a regiszterekben történő feldolgozást definiálva végezzük a tervezést.
- **Logikai kapuk szintje (angolul: *gate level*)**:
  A tervezendő modulokat logikai kapuk megfelelő megválasztásáva és összekötésével valósítjuk meg.
- **Kapcsoló szint (angolul: s*witch level*)**:
  Ez az elvonatkoztatás legalacsonyabb szintje, amelyet a *Verilog* támogat. A modulokat kapcsolókból (tranzisztorok) és memória elemekből építjük fel, megfelelő kötések alkalmazásával.

A *Verilog HDL* lehetővé teszi a négy szint egy projektumon belül történő alkalmazását. Leggyakoribb a két felső szint kombinációja, ezt az *RTL (angolul: register transfer level)* kifejezéssel jelölik a szakirodalomban.

Magasabb szintű elvonatkoztatás esetén a tervezés rugalmasabb és kevésbé függünk a megvalósítást végző technológiától. A kapcsoló szinthez közelítve a rugalmasság csökken és jobban függünk a rendelkezésre álló hardver eszközöktől. Ilyenkor kis változtatás a feladat leírásában nagy változásokhoz vezethet a megvalósításban. A helyzet a *C* nyelvben és az assembly nyelvben történő programozáshoz hasonlítható. Könnyebb a magasabb szintű programnyelvekben (pl. *C* nyelv) programozni, mint az assembly nyelvben, a *C* nyelven írott szoftver könnyebben átvihető egyik gépről a másikra, míg az assembly nyelven írott szoftver rendszerint géptől függő, nehezen vihető át másik gépre.

5.3. Instanciák

A modul tulajdonképpen egy minta, amely alapjána az azonos szerkezetű konkrét objektumokat megvalósítjuk. A modul alkalmazása során a *Verilog HDL* a minta alapján létrehoz egy konkrét objektumot. Minden objektumnak egyedi nevet adunk, definiáljuk a bementi és kimeneti vonalait, az esetleges paramétereket és változókat. A konkrét objektumnak minta alapján történő létrehozását itt instanciálásnak (angolul: *instantiation*) nevezzük, a létrehozott objektum neve instancia (angolul: *instance*).

Az 5-4 ábrán bemutatott négy bites számláló hardver nyelvi leírása a *T flip-flop* négy instanciáját tartalmazza. A továbbiakban minden *T flip-flop* egy *D flip-flop*-ot, kettő *ÉS* kaput, egy logikai invertert és egy *VAGY* kaput instanciál. Az 5-2 példa a négy bites számláló hardver nyelvi moduljának definícióját mutatja be. Minden instanciának egyedi nevet kell adni. A leírásban szereplő kettős dőlt vonal (//) egysoros megjegyzést (kommentár) vezet be. A kommentárnak nincs kihatása a tervezendő áramkörre, csak a tervező számára tartalmaz információkat.

5-2 példa: Modul instanciálása.

```
// Feltételezzük, hogy a T flip-flop-nak megfelelő modult már korábban definiáltuk.

module szamlalo(q, clock, reset);
output [3:0] q;                    // a négy kimeneti bit deklarálása
input clock, reset;                // egy bites vezérlőjelek deklarálása
reg t=1'b1;

T_FF tff0(q[0], t, clock, reset);  // A T_FF modul instanciálása tff0 név alatt.
T_FF tff1(q[1], t, q[0], reset);   // A T_FF modul instanciálása tff1 név alatt.
T_FF tff2(q[2], t, q[1], reset);   // A T_FF modul instanciálása tff2 név alatt.
T_FF tff3(q[3], t, q[2], reset);   // A T_FF modul instanciálása tff3 név alatt.

endmodule
```

A *Verilog HDL* nem teszi lehetővé modul definiálását másik modul definíción belül. Ez helyett a szükséges modult instanciálni kell.

Nem szabad összekeverni a modulok definiálását a modulok instanciálásával. A következő analógiával élhetünk: a ház építésénél a tervrajz a *Verilog* modul definíciójának felel meg, az építés folyamata analóg az instanciálással, maga a kész ház az instancia.

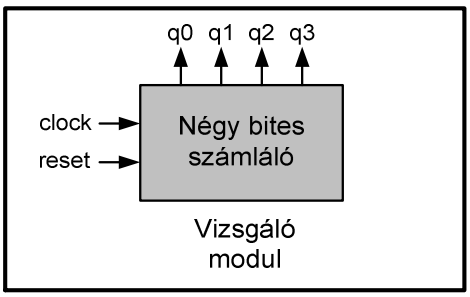
5.4. Szimulációk

Mielőtt megfelelő hardver eszközökkel (*PLD*) megvalósítanánk és valós körülmények között kivizsgálnánk a hardver nyelvi leírásnak megfelelő logikai áramkört, számítógépes szimulációval ellenőriznünk kell annak helyességét. A leírás működőképességét rendszerint az adott bemeneti jelekre adott válaszjeleket vizsgálva derítjük ki. Tudatában kell lennünk, hogy az ilyen szúrópróbás vizsgálódás annyira ad biztos eredményt, amennyire jól tudjuk megválasztani a

bemeneti jeleket. A matematikusok próbáloznak szisztematikus módszerek kidolgozásán, de ma ez még gyerekcipőben jár.

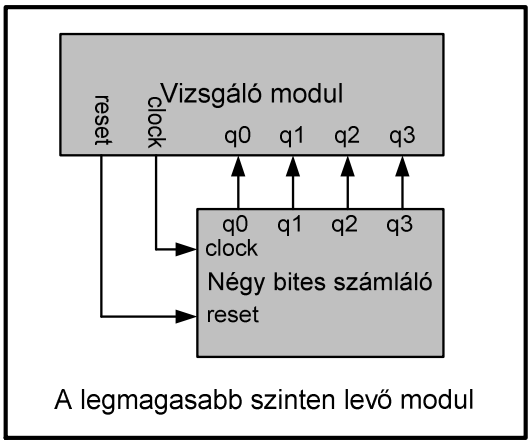
A bemeneti jeleket generáló és a kimeneti jeleket fogadó egység neve vizsgáló tömb (angolul: *stimulus block* vagy *test bench*). A vizsgáló tömb maga is egy hardver nyelvi modul. Ügyelni kell, hogy ne keverjük a vizsgáló tömböt magával a megvalósítandó feladattal. A vizsgáló tömböt két féle képpen alkalmazhatjuk a vizsgálat tárgyára:

- A vizsgáló tömb instanciálja a vizsgálat tárgyát képező modult, bementi jeleket biztosít számára és fogadja annak kimeneti jeleit. A tervezési hierarchiában a vizsgáló blokk a legfelső *Verilog* modul. Ezt az esetet a 5-8 ábrán mutattuk be. A vizsgálat tárgyát a 5-3 ábrán bemutatott négy bites számláló képezi.



5-8 ábra: A vizsgáló tömb (modul) instanciálja a feladatot megvalósító modult.

- A tervezett (vizsgálandó) modult és a vizsgáló modult egy, a hierarchiában legmagasabb szinten levő, üres modulon (angolul: *dummy block*) belül instanciáljuk A vizsgáló modul a vizsgált modullal a *port*-okon keresztül kommunikál. A legfelső modul szerepe csak az említett két blokk instanciálása. Ezt az elrendezést a 5-9 ábra szemlélteti.



5-9 ábra: Egy üres modul instanciálja a vizsgáló modult és a tervezett modult (négy bites számláló).

6. Alapfogalmak a *Verilog HDL*-ben

Ebben a fejezetben a *Verilog* hardver leíró nyelv alapfogalmaival és elemi szabályaival ismerkedünk. Az ismeretanyag meglehetősen száraz, de a következő fejezetek megértéséhez és követéséhez nélkülözhetetlen.

6.1. Nyelvi szabályok

A *Verilog HDL*-ben használatos alapvető nyelvi szabályok meglehetősen hasonlítanak a *C* nyelv szabályaira. A *Verilog* nyelvi szerkezetek jelölések (angolul: *token*) sorozatából állnak. Ezek a jelölések a következők lehetnek: megjegyzések (kommentár), műveleti jelek, számok, jelsorok (angolul: *string*), azonosítók (angolul: *identifier*) és kulcsszavak (angolul: *keyword*).

A *Verilog* nyelvben megkülönböztetjük a kis és a nagy betűket (angolul: *case sensitive*). Például a *szamlalo* és a *Szamlalo* elnevezésű azonosítók nem ugyanazt a változót nevezik meg, mivel egyiknél az első betű kicsi, a másiknál nagy. Minden kulcsszót kis betűvel írunk a *Verilog HDL*-ben.

6.1.1. Üres helyek

A betűközt, tabulátort és új sort a *Verilog HDL*-ben közös néven üres helyeknek nevezzük. Ezeket az üres helyeket a nyelvi szerkezetek értelmezése során figyelmen kívül hagyjuk, csak elválasztó jelekként értelmezzük őket két jelölés között. Ezzel ellentétben, a jelsorok értelmezésénél az üres helyeket is figyelembe vesszük.

6.1.2. Megjegyzések (kommentárok)

A különböző megjegyzések a szoftvernyelvekben elengedhetetlenek a megírt program dokumentálása és a későbbi elemzés megkönnyítése érdekében. Ugyanez érvényes a hardver leíró nyelvekre is. Alapvető szabály, hogy minden olyan utasításhoz vagy nagyobb nyelvi egységhez kommentárt kell fűzni, amelynek megértésében nehézségek adódhatnak egy hét vagy egy hónap után. A kételkedőknek tudniuk kell, hogy a kommentárok írása nem csökkenti a programozó intelligenciáját.

A kommentároka két féle képpen írhatjuk (6-1 példa):

- **Egysoros kommentár**. A „//“ jel utasítja a hardver nyelvi leírást értelmező szoftvert, hogy az ettől a jelől a sor végéig található jeleket át kell ugrania.
- **Többsoros kommentár**. A több soros megjegyzéseket a „**/***“ jellel kezdjük és a „***/**“ jellel fejezzük be.

6-1 példa: Egy soros és több soros kommentár írása.

```
a = b + c;              // Ez egy egysoros kommentár.
/*Ez egy több soros
kommentár*/
```

6.1.3. Műveleti jelek

A műveleti jeleket (angolul: *operator*) három csoportba osztjuk: vannak unáris-, bináris- és ternáris műveleti jelek (6-2 példa). Az unáris műveleti jelek egy változóra vonatkoznak és közvetlenül a változó elé írandók. A bináris műveleti jeleket két változó közé írjuk. A ternáris műveleti jelek két külön jelből állnak, amelyek három változót választanak el egymástól.

6-2 példa: Unáris, bináris és ternáris műveleti jelek.

```
a = ~b;                 // A ~ jel egy unáris műveleti jel, b a változó.
a = b && c;             // A && jel egy bináris műveleti jel, b és c a változók.
a = b ? c : d;          // A ?: jelkombináció egy ternáris műveleti jel, b, c és
                        // d a változók.
```

6.1.4. Számok írása

A számértékekeket két módon írhatjuk a *Verilog HDL*-ben, mint méretezett számokat és mint méret nélküli számokat:

- **Méretezett számok (angolul: s*ized numbers*)**:

  A méretezett számokat a következő módon adjuk meg:

  <méret>'<a számrendszer alapja><szám>

A **<méret>** egy decimális szám, amely az illető szám írásához használatos bitek számát adja meg.

A *Verilog HDL*-ben támogatott számrendszerek a tizes (decimális) rendszer (jele **'d** vagy **'D**) a kettes (bináris) rendszer (jele **'b** vagy **'B**), a nyolcas (oktális) rendszer (jele **'o** vagy **'O**) és a tizenhatos (hexadecimális) számrendszer (jele **'h** vagy **'H**).

A <**szám**>-ot a következő számjegyek segítségével írjuk: *0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f,* illetve értelemszerűen ennek a halmaznak az adott számrendszerben használatos részhalmazával. A számjegyek írásánál egyaránt használhatók kis és nagy betűk (*a, b, c, d, e, f, vagy A, B, C, D, E, F) (6-3 példa)*.

6-3 példa: Méretezett számok írása.

```
3'b101          // három bites bináris szám
16'hfa3e        // tizenhat bites hexadecimális szám
16'HFA3e        // ugyanaz, mint a fenti szám, csak egyes számjegyeket nagy betűvel
                // írtunk.
```

- **Méret nélküli számok (angolul: *unsized numbers*):**

Ha nem adjuk meg a szám írásához használatos bitek számát (hiányzik a <**méret**> paraméter), a *HDL* leírást értelmező szoftver (szimulátor, szintetizáló szoftver) fogja eldönteni a szám hosszát az alkalmazott operációs rendszer vagy a számítógép regisztereinek hossza alapján (de nem lehet kevesebb harminckét bitnél).

Ha hiányzik a <**számrendszer alapja**> paraméter, megegyezés szerint az illető számot decimális számként kell értelmezni (6-4 példa).

6-4 példa: Számok írása méret nélkül.

```
'o7651          // Harminckét bites oktális szám.
'hAB3           // Harminckét bites hexadecimális szám.
4556            // Harminckét bites decimális szám.
```

- ***x* és *z* értékek a számjegyek között**

A *Verilog HDL*-ben a számjegyek írásánál két különleges betűjelet is alkalmazunk: az *x* jel ismeretlen értéket jelöl, a *z* jel nagyimpedanciás állapotot (6-5 példa). A valós digitális áramkörök modellezéséhez elengedhetetlenek ezek a jelölések.

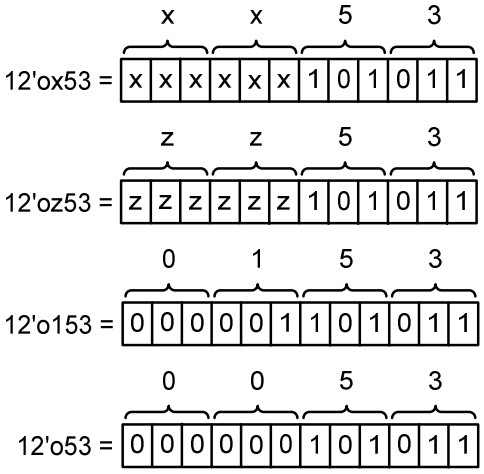
6-5 példa: Az x és z értékek a számjegyek között.

```
12'h13x         // Tizenkét bites hexadecimális szám, a négy legkisebb
                // helyi értékű bit értéke ismeretlen.
6'hx            // Ismeretlen értékű hat bites szám.
32'bz           // Harninckét bites bináris szám. Minden bit
                // nagyimpedanciás állapotban van.
```

A tizenhatos számrendszerben írott számoknál az *x* illetve a *z* jelek négy bitet helyettesítenek, a nyolcas rendszerben három bitet, a kettes rendszerben egy bitet.

Fontos megemlíteni, mi történik, ha a számérték nem tölti ki **<méret>** paraméterrel megadott hosszúságú regisztert. Ilyenkor, ha a definiált legnagyobb helyi értékű bit értéke *0, x* vagy *z*, a meg nem határozott nagyobb helyi értékű bitek is *0, x* illetve *z* értéket kapnak. Így könnyen kitölthető az egész regiszter *0, x* illetve *z* értékekkel. Ha a legnagyobb helyi értékű megadott bit értéke *1*, akkor a felette levő határozatlan bitek *0* értéket kapnak.

A 6-1 ábrán olyan esteket adtunk meg, ahol a szám mérete nagyobb a megadott számértéknek megfelelő bitek számánál.



6-1 ábra: Példa olyan számokra, amelyeknél a méret nagyobb a megadott bitek számánál.

- **Negatív számok írása**

A negatív számokat úgy jelöljük, hogy a **<méret>** paraméter elé minusz jelet teszünk. Nem szabad minusz jelet tenni a számrendszert jelölő paraméter és a számérték közé (6-6 példa).

A feldolgozás (szimuláció, szintézis) során a megfelelő szoftver a negatív számok írására a kettes komplemenst használja.

6-6 példa: A negatív számok írásmódja.

```
-8'hA1          // Nyolc bites negatív szám.
8'h-A1          // A negatív számok téves írásmódja.
```

- **Alsó összekötő vonal**

Az alsó összekötő vonal (_) használható a jobb áttekinthetőség érdekében a számértékek írásánál, bármelyik helyi értékek között, de nem használható az első számjegy előtt (6-7 példa). A *Verilog HDL* leírás értelmezésére az alsó összekötő vonalnak nincs kihatása.

6-7 példa: Az alsó összekötő vonal használata.

```
12'b1100_0101_1111   // Az alsó összekötő vonalat a jobb áttekinthetőség
                     // érdekében használjuk.
```

- **Kérdőjel**

A számok írásánál a nagyimpedanciás állapot jelölésére használhatunk a *z* jel helyett kérdőjelet (*?*) is (6-8 példa). Ugyancsak kérdőjelet használunk bizonyos ***casex*** és ***casez*** kifejezésekben, így javítva a *HDL* leírás áttekinthetőségét. Ezekkel az utasításokkal később fogunk részletesen foglalkozni.

6-8 példa: A nagyimpedanciás állapot jelölése kérdőjellel .

```
4'b01??          // Ugyanazt jelenti mint a 4'b01zz kifejezés.
```

6.1.5. Jelsorok

A jelsorok (angolul: *string*) két idézőjel közé foglalt jelek összessége. Egy jelsort kötelezően egy sorba kell írni, nem eszközölhető meghosszabbítás több soron keresztül (6-9 példa).

6-9 példa: A jelsorok írása.

```
"Ez egy jelsor."
```

6.1.6. Azonosítók

Az azonosítók (angolul: *identifier*) objektumok nevei a *Verilog HDL*-ben, amelyekkel egyértelműen hivatkozni lehet az egyes objektumokra. Az azonosítók betűkből, számokból, alsó összekötő vonalból (_) és dollár jelből ($) állhatnak. Az azonosítók megválasztásánál és írásásnál különbséget teszünk a kis és a nagy betűk között. Az azonosító nem kezdődhet számmal vagy dollár jellel. (A dollár jelet mint első jelet a rendszerfüggvények nevében használjuk).

6.1.7. Kulcsszavak

A kulcsszavak (angolul: *key word*) az azonosítók egy különleges fajtáját képezik, amelyeket a nyelvi szerkezetek definiálására használunk A kulcsszavakat kötelezően kis betűkkel kell írni (6-10 példa).

6-10 példa: Kulcsszavak használata.

```
reg szam;       // A reg kulcsszó; a szam egy adat azonosítója.
input Input;    // Az input  kulcsszó;  az Input viszont egy adat azonosítója.
                // A Verilog HDL-ben megkülönböztetjük a kis és a nagy betűket
                // ezért az input nem ugyanazt jelenti mint az Input.
```

6.2. Az adatok és adathordozók típusai

Ebben a fejezetben a *Verilog HDL*-ben használatos adathordozók típusokat tárgyaljuk. A definiálásra használatos kulcsszavakat vezetjük be és az egyes adathordozó típusokra vonatkozó szabályokat ismertetjük.

6.2.1. A *Verilog HDL*-ben használatos logikai értékek

Amint azt a 6.1.4. pontban a számok írásánál letárgyaltuk, a *Verilog HDL*-ben a számok egyes bitjei négy értéket vehetnek fel (6-1 táblázat). Ez általában is érvényes a különböző típusú adatok bitjeire is.

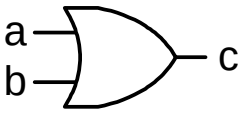
6-1 táblázat: A Verilog HDL által támogatott logikai értékek.

Jelölés	Logikai állapot
0	logikai nulla
1	logikai egyes
x	ismeretlen állapot
z	nagyimpedanciás állapot

6.2.2. Csomópontok, vezetékek

A *Verilog HDL*-ben az alapvető adathordozó típus a különböző hardver elemeket összekötő csomópontra (angolul: *net*) vagy vezetékre vonatkozó logikai állapot. Mint a valós áramkörökben, itt is a csomópont logikai állapotát az adott csomóponthoz kötött logikai áramkör kimenete határozza meg.

A 6-2 ábrán a *VAGY* logikai kapu kimenete a *net* típusú *c* csomóponthoz van kötve. A *c* változó logikai értékét minden pillanatban a *VAGY* kapu kimenete határozza meg.



6-2 ábra: Példa a csomópont típusú adatokra.

A *net* típusú adathordozót leggyakrabban a ***wire*** kulcsszóval deklaráljuk (6-11 példa). Rendszerint egy bites adatról van szó, kivéve, ha vektorként deklaráljuk. Az irodalomban a ***wire*** fogalmát gyakran a *net*-tel azonosítják. Megegyezés szerint, a ***wire*** kulcsszóval deklarált adathordozó nagyimpedanciás logikai állapotot mutat, ha nincs összekötve valamilyen logikai áramkör kimenetével.

6-11 példa: A net típusú adathordozók deklarálása a wire kulcsszóval.

```
wire c;          //A csomópont (net) típusú c adathordozót deklaráló nyelvi
                 // szerkezet (6-2 ábra).
wire a, b;       // A csomópont (net) típusú a és b adathordozókat deklaráló
                 // nyelvi szerkezet (6-2 ábra).
wire d = 1'b0;   // A csomópont (net) típusú d  adathordozót deklaráljuk és
                 // állandóan nulla logikai értéket rendelünk hozzá.
```

Ha egy adathoz állandó logikai értéket rendelünk (mint *d* esetében), nem tanácsos ezt a csomópontot valamilyen digitális kimenethez kötni. Ha ez mégis megtörténne, ütközni fognak a logikai értékek, ha az adott kimeneten logikai egyes akar megjelenni.

Fontos megjegyezni, hogy a *net* adathordozó típus, de nem kulcsszó. Ugyanebbe az adathordozó típusba tartoznak a ***wire*** mellett a ***wand***, ***wor***, ***tri***, ***triand***, ***trior***, ***trireg*** kulcsszavakkal deklarált adathordozók, ezeket ritkábban használjuk.

6.2.3. Regiszterek

A regiszterek logikai állapotokat képesek megjegyezni. A memorizálás mindaddig tart, amíg új értéket nem adunk a regiszternek.

A *Verilog HDL*-ben használatos regiszter fogalmát nem kell azonosítani a *flip-flop*-okból felépített hardveres regiszterekkel, amelyek az órajel felfutó vagy lefutó élénél kapnak új tartalmat. A *Verilog HDL*-ben a regiszter egy olyan típusú adathordozó, amelynek értéke egy korábbi hozzárendelés eredménye. Ellentétben a *net* típusú adathordozóval, a regiszternek nincs szüksége megfelelő digitális kimenetre, hogy a nagyimpedanciás állapottól eltérő állapotba legyen vezérelve. Hasonlóan, a *Verilog* regisztereknek nincs órajel bementük, amellyel az állapotváltozást szinkronizálhatnánk. A *Verilog* regiszterek tartalma bármely pillanatban megváltoztatható megfelelő hozzárendeléssel.

A regiszter típusú adathordozókat ***reg*** kulcsszóval deklaráljuk. Alapértelmezésben a regiszter logikai állapota határozatlan (*x* érték), mindaddig ez van érvényben, amíg megfelelő hozzárendeléssel nem adunk konkrét értéket a regiszternek. A 6-12 példában regiszter típusú adathordozók deklarálását láthatjuk.

6-12 példa: Regiszter típusú adathordozók deklarálása a reg kulcsszóval.

```
reg reset;              // A reset olyan adat, amely adott értéken marad.
initial                 // Az initial  nyelvi szerkezetet később fogjuk
begin                   // megmagyarázni.
   reset = 1'b1;        // A reset  nevű regiszternek logikai egyes értéket adunk.
   #100 reset = 1'b0;   // Száz időegység után a reset  regiszter értékét
end                     // visszaállítjuk logikai nullára.
```

6.2.4. Vektorok

A *net* és a ***reg*** típusú adatok vektorokként is deklarálhatók (ha több mint egy bites adathordozók szükségesek). Ha nem definiáljuk a bitek számát, alapértelmezés szerint egy bites adattal (skalár) dolgozunk (6-13 példa).

6-13 példa: Vektorok deklarálása.

```
wire kimenet;                 // A kimenet net típusú skalár adat.
wire [7:0] led_kijelzo;       // Egy 8 bites  net típusú adat vektor.
wire [15:0] ledA, ledB;       // Két 16 bites  net típusú adat vektor.
reg orajel;                   // Regiszter típusú skalár adat.
reg [0:255] mem;              // Regiszter típusú 256 bites adat vektor.
```

A vektorok hossza megadható [*nagyobb_szám* : *kisebb_szám*] vagy [*kisebb_szám* : *nagyobb_szám*] formában is. A bal oldali szám mindig a vektor legnagyobb helyi értékű bitjét definiálja. A fenti példában a *ledB* vektorban a legnagyobb helyi értékű bit a 15-ös számú, míg a *mem* regiszter típusú vektorban a legnagobb helyi értékű bit a 0-dik bit.

A teljes vektor helyett az egyes kifejezésekben alkalmazhatjuk a vektor bizonyos bitjeit vagy bitcsoportjait. Az előző példában definált vektorok bitjeinek és bitcsoportjainak használatát a 6-14 példában láthatjuk.

6-14 példa: Vektorok bitjeinek és bitcsoportjainak használata.

```
ledB [7];                     // A ledB adat 7-es számú bitje.
ledA[1:0];                    // A ledB adat két legkisebb helyi értékű bitje.
                              // Nem használható a ledA[0:1] írásmód, mert a legnagyobb
                              // helyi értékű bit mindig a bal oldalon kell, hogy  legyen.
mem [0:1];                    // A mem regiszter típusú adat vektor két
                              // legnagyobb helyi értékű bitje.
```

6.2.5. Egész számok

Az egész számok (angolul: *integer*) általános rendeltetésű regiszter jellegű adatok. Az egész szám típusú adathordozók definálására külön kulcsszó létezik, ez az ***integer***. Ugyanezekre az adatokra használhatnánk a ***reg*** kulcsszót is, de az a szokás, hogy a számlálással kapcsolatos regisztereket ***integer*** kulcsszóval deklaráljuk (6-15 példa). Alapértelmezésben az egész szám típusú adat bitjeinek száma az alkalmazott hardvernél jellemző regiszter hosszával egyenlő, de legalább 32 bit hosszúságú. A ***reg*** típusú adathordozókhoz csak pozitív egész számok rendelhetők, míg az integerként deklarált adatok felvehetnek negatív értéket is.

6-15 példa: Egész szám típusú adatok deklarálása.

```
integer szamlalo;             // Az integer típusú, szamlalo elnevezésű adat deklarálása.
initial                       // Ezt a nyelvi szerkezetet később magyarázzuk.
   szamlalo = -1;             // A szamlalo elnevezésű adat kezdő értéke -1.
```

6.2.6. Valós számok

A valós számok is regiszter jellegű adatok a *Verilog HDL*-ben. Deklarálásuk a ***real*** kulcsszóval történik. A ***real*** típus adathordozókhoz az értékek hozzárendelhetők decimális alakban (pl. 3.14) vagy exponenciális alakban (pl. 3e2 = 300) (6-16 példa). Ügyeljünk arra, hogy a Verilog HDL-ben nem vesszüt, hanem pontot használunk a tört rész elválasztására.

A *real* típusú adatok tárolására szolgáló regiszter hossza az alkalmazott hardver lehetőségeitől függ. Alapértelmezésben a a ***real*** típusú adat értéke 0. Ha ***real*** típusú adatot rendelünk hozzá az ***integer*** típusú adathordozóhoz, kerekítés történik a legközelebbi egész számra.

6-16 példa: A real típusú adatok deklarálása és használata.

```
real alfa;              // Az alfa nevű, real  típusú adat deklarálása.
initial
begin
    alfa = 4e10;        // Az alfa adathordozóhoz exponenciális alakba felírt értéket
                        // rendelünk hozzá.
    alfa = 5.31;
end
integer i;              // Az i adat integer típusú.
    i = alfa;           // Az i adat értéke 5 lesz a kerekítés következtében.
```

6.2.7. Sorok

A *Verilog* leírásokban képezhetők egydimenziós sorok (angolul: *array*) a ***reg*** és az ***integer*** típusú adatokból Nem képezhatők sorok a ***real*** típusú adatokból és nem képezhetők többdimenziós sorok. A sorok deklarálása a következő formában történik: <a_sor_neve>[kisebb_szám : nagyobb_szám] vagy <a_sor_neve>[nagyobb_szám : kisebb_szám] (6-17 példa).

6-17 példa: Sorok deklarálása.

```
integer szamok[0:7];        //Nyolc tagú, integer típusú adatokat tartalmazó sor.
reg mem[31:0];              // Harminckét egybites adatból képezett sor.
reg [3:0] port [0:7];       // Nyolc elemet tartalmazó sor, minden elem négy
                            // bites.
integer matrix [4:0] [4:0]; // Nem megengedett deklaráció: a Verilog HDL nem
                            // támogatja többdimenziós sorok deklarálását.
szamok[4]                   // A szamok sor negyedik eleme.
port[5]                     // A port sor ötös számú eleme.
```

Nem szabad összekeverni a vektorok és a sorok fogalmát. A vektor egy adat, amelyben a bitek száma *n*, míg a sor több adatból álló halmaz, amelyben az egyes adatok egy- vagy több bitesek.

6.2.8. Memóriák

A *PLD*-kre alapuló digitális tervezésnél gyakran szükséges regisztereket illetve memóriaegységeket (*RAM*, *ROM*) megvalósítani. A memóriákat a *Verilog HDL*-ben regiszter típusú adatok soraként deklaráljuk (6-18 példa). Az egyes memória elemeket szavaknak nevezzük (angolul: *word*). A szavak lehetnek egy vagy *n* bitesek. Nem szabad összetéveszteni *n* darab egy bites regisztert egy darab *n* bites regiszterrel.

6-18 példa: Memóriák deklarálása regiszerek soraként.

```
reg mem1bit[0:1023];        // A mem1bit memória 1024 darab egy bites
                            // szóból alkotott sor.
reg [7:0] membyte [0:1023]; // A membyte memória 1024 darab nyolc bites
                            // szóból alkotott sor.
membyte[521]                // A membyte név alatt deklarált memória 521-es
                            // számú eleme.
```

6.2.9. Paraméterek

A *Verilog HDL* lehetővé teszi konstansok definiálását a modulon belül a ***parameter*** kulcsszóval (6-19 példa). A paraméterek nem használhatók adatokként.

6-19 példa: Paraméterek definiálása.

```
parameter diodak_szama = 5; // A diodak_szama nevű adatot konstansként
                            //deklaráltuk és 5-ös értéket adtunk neki.
```

Tanácsos a *Verilog HDL* modulok definiálásakor paramétereket alkalmazni. Ha közvetlenül számértékeket írunk, nagyobb valószínűséggel követünk el hibát. A paraméterek használata áttekinthetőbbé teszi a hardver nyelvi leírásokat és megkönnyíti az esetleges változtatásokat.

6.2.10. A *$stop* és a *$finish* rendszerfüggvények

A ***$stop*** rendszerfüggvény ideiglenesen leállítja a hardver nyelvi leírásban megadott szimulációt, így menet közben vizsgálhatók a megfelelő jelek. A *$stop* rendszerfüggvény írásának módját a 6-20 példában láthatjuk.

6-20 példa: A $stop rendszerfüggvény.

```
$stop;
```

A ***$finish*** rendszerfüggvény hatására a szimuláció megszakad (6-21 példa).

6-21 példa: A $finish rendszerfüggvény.

```
$finish;
```

6.2.11. A ‘*define* utasítás

A **‘define** utasítás segítségével szöveg *makro*-kat definiálhatunk, amit az értemező szoftver fog felhasználni (behelyettesíteni). A definiálás módját a 6-22 példában láthatjuk.

6-22 példa: Szöveg makro-k definiálása.

```
’ define <a_makro_neve> <a_makro_szövege>
```

A *makro* definiálásánál és alkalmazásánál mindig jelen van a ’ jel. A *makro* célja, hogy meggyorsítsa a *HDL* leírás elkészítését (mivel hosszú szövegeket a *makro* rövid nevével helyettesíthetünk) és javítsa az áttekinthetőséget. Az értelmezés során a szoftver a *makro* neve helyett mindenhol a *makro* szövegét helyettesíti be (6-23 példa).

6-23 példa: A makro-k használata.

```
‘define S $stop;          // A ‘S jelet helyére mindenhol a $stop szöveget kell beírni.

‘define WORD_REG reg [31:0]
// A fenti makro definíciót alkalmazva egy 32 bites regiszter deklarálása a következő
// módon történhet:
‘WORD_REG reg32;          // A reg32 nevű adat egy 32 bites regiszter típusú adat.
```

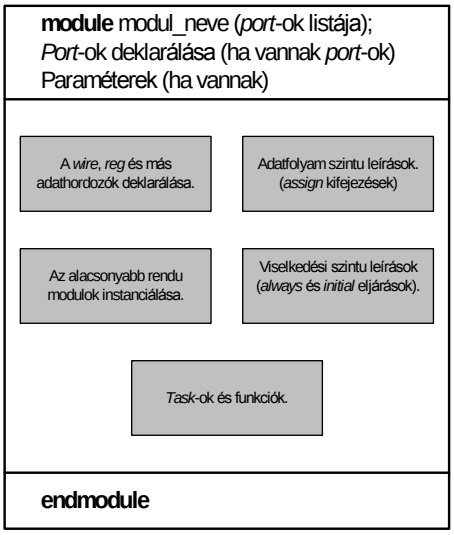
7. Modulok és *port*-ok

Az előző két fejezetben megismerkedtünk a hierarchikus tervezés fogalmával, valamint a *Verilog HDL* alapvető nyelvi szabályaival és az adatok és adathordozók típusaival. Most a korábban bevezetett modulokkal és *port*-okkal foglalkozunk részletesebben.

7.1. Modulok

Az 5.2 szakaszban elmondtuk, hogy a modulok a *Verilog* nyelv építőelemei, amelyek lehetővé teszik a hierarchikus tervezést. Ott csak a modulok definiálásával és instanciálásával foglalkoztunk, a belső szerkezetre nem tértünk ki. Ez a szakasz a belső szerkezetet részletezi.

A modul egy ASCII karakterokból kialakított szöveges leírás. Több különböző részből áll, ahogyan az a 7-1 ábrán látható.



7-1 ábra: A modul alkotó elemei.

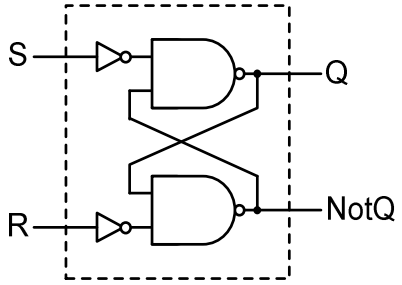
A modul definiálása mindig a ***module*** kulcsszóval kezdődik. Először is megadjuk a modul nevét, a *port*-ok elnevezéseit, deklaráljuk a *port*-ok jellemzőit és paramétereket vezethetünk be. Természetesen a *port*-ok elnevezései és a deklarálásuk csak akkor jelenik meg, ha a tervezendő modul kommunikál más modulokkal.

A modul testét képező központi részben öt különböző egység foglalhat helyet. Ezek: az adat típusok deklarálása, adatfolyam szintű kifejezések, alacsonyabb rangú modulok instanciálása, viselkedési szintű leírások és *Verilog* funkciók. A felsorolt elemeket bármilyen sorrendben írhatjuk a modul testén belül.

A modul definiálása mindig az ***endmodule*** kulcsszóval fejeződik be. Csak a ***module*** kulcsszó, a modul neve és az ***endmodule*** kulcsszó használata kötelező, bármely más elem illetve elemek kihagyhatók a modul definiálásakor.

Egy ASCII fájlban több modul definiálása is elvégezhető. A fájlon belül a modulok sorrendje tetszőleges, nincs összefüggésben a modulok közötti hierarchiával.

Egy példán keresztül szemléltetjük a modul különböző részeit egy *RS latch*-et definiáló modul esetére. A 7-2 ábrán láthatjuk a *latch* logikai rajzát az *S* és *R* bemenetekkel és a *Q* és *NotQ* kimenetekkel. A 7-1 példában megadtuk az *RS latch*-et definiáló modult, és egy vizsgáló modult (*Stimulus* név alatt), amellyel a *latch* működése ellenőrizhető.



7-2 ábra: Az RS latch logikai rajza.

7-1 példa: Az RS latch-et leíró modul és a vizsgáló modul.

```
//A modul neve és a port-ok elnevezései
module RS_Latch(Q, NotQ, R, S);

// A port-ok deklarálása
output Q, NotQ;
input R, S;

// Alacsonyabb rendű modulok instanciálása. Ez esetben két NEM-ÉS kaput
// instanciálunk, amelyek alapelemek (angolul: primitive) a Verilog HDL-ben.
nand n1(Q, ~S, NotQ);
nand n2(NotQ, ~R, Q);

// A modult kötelezően a következő kulcsszóval zárjuk le:
endmodule
```

```
// A modul neve és a port-ok elnevezései
module Stimulus;
// Nincsennek port-ok, mert a vizsgáló modul a legmagasabb rendű modul.

// Az adatok típusának deklarálása.
wire q, notq;
reg set, reset;

// A fent definiált alacsonyabb rendű modul instanciálása.
RS_FlipFlop f1(q, notq, reset, set);

// Egy viselkedési szintű (angolul: behavioral) leírás a set és a reset jelek
// időfüggvényének megadására.
initial
begin
      set = 0; reset = 0;
      #5 reset = 1;
      #5 reset = 0;
      #5 set = 1;
      #5 $finish;
end

// A modult kötelezően a következő kulcsszóval zárjuk le:
endmodule
```

A fenti példában megfigyelhetünk néhány jellegzetességet:

- Az *RS latch*-et leíró modulban nincsennek jelen az összes alkotó elemek, amit feltüntettünk a 7-1 ábrán. Nem deklaráltunk semmilyen adat típust, nincs semmilyen adatfolyam szintű kifejezés (***assign***), nincs viselkedési szintű leírás (***always*** vagy ***initial***), mégis helyes a modul definíciója.
- Az *RS latch* viselkedését vizsgáló modulban (*Stimulus*) deklaráltunk adat típusokat (***wire***, ***reg***), a leírás tartalmaz egy viselkedési szintű leírást, nincsennek viszont *port*-ok és nincs adatfolyam szintű leírás.

Még egyszer hangsúlyozzuk, hogy csak a ***module*** kulcsszó, a modul neve és az ***endmodule*** kulcsszó használata kötelező minden modulban, a többi elemeket szükség szerint és tetszőleges sorrendben írva alkalmazzuk.

7.2. Port-ok

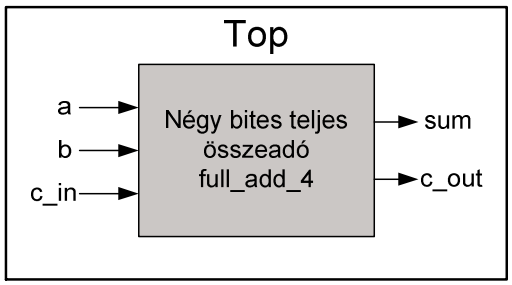
A *port*-ok a modul csatlakozási pontjai, rajtuk keresztül valósul meg a kommunikáció másik modulokkal. A *Verilog HDL* leírásokban a *port*-ok olyanok, mint az integrált áramköröknél a kivezetések. A kommunikáció a modullal csak a *port*-okon keresztül történhet. A modul belső szerkezete kívülről láthatatlan. Ez nagy rugalmasságot biztosít a tervezéskor, mert megváltoztathatjuk a modul felépítését anélkül, hogy ez kihatna a más modulokkal való kapcsolatra. A *port*-okat nevezik még termináloknak is (angolul: *terminal*).

7.2.1. A *port*-ok listája

A *port*-ok elnevezéseinek felsorolása a modulok definiálásakor nem kötelező. Ha a modul nem cserél adatokat a környezetével (más modulokkal), akkor nincs szükség a *port*-ok listájára.

A 7-3 ábrán látható szerkezetben a *Top* elnevezésű modulnak nincs *port* listája, az általa instanciált *full_add_4* nevű modulnak viszont van *port* listája. A megfelelő modul definíciók első sorát a 7-2 példában adtuk meg.

7-3 ábra: Kimeneti és bemeneti port-ok a Top és a full_add_4 modulok esetében.



7-2 példa: A port-ok listája

```
module full_add_4(sum, c_out, a, b, c_in);     // Port listával rendelkező modul.
module Top;                                    // Port lista nélküli modul.
```

A *Top* nevű modul van a hierarchia legmagasabb fokán, ezért nem fogad bemenő jeleket és nem ad kimenő jeleket magasabb szintű moduloknak, így nem tartalmaz *port* listát. A legmagasabb szintű modul csak a *HDL* leírást értelmező szoftverrel tart kapcsolatot, ez viszont nem jelenik meg magában a modul definíciójában.

7.2.2. A *port*-ok deklarálása

A *port* listában szereplő minden *port*-ot a modulon belül deklarálni kell. A deklarálás során megadjuk a *port* irányát: van kimenő-, bemenő- és kétirányú *port* (7-1 táblázat).

7-1 táblázat: A port-ok lehetséges irányai.

A *port* típusa	Irány
input	bemenet
output	kimenet
inout	kétirányú *port*

A 7-3 példában látható a *port*-ok deklarációja a 7-3 ábrán említett négy bites teljes összeadó esetére.

7-3 példa: A port-ok deklarálása

```
module full_add_4(sum, c_out, a, b, c_in);

// A port-ok deklarálásának kezdete
output [3:0] sum;
output c_out;
```

```
input [3:0] a, b;
input c_in;
// A port-ok deklarálásának vége.

//-------------------------------------
// A modul teste.
//-------------------------------------
endmodule
```

Ha egy *port*-ot kimenő-, bemenő- vagy kétirányú *port*-ként definiáltunk, alapértelmezésben a *port*-hoz tartozó adat típusa ***wire***. Amennyiben a ***wire*** adattípus megfelel, elegendő a *port* deklarálása az ***input***, ***output***, ***inout*** kulcsszavakkal, nincs szükség külön deklarálni az adat típusát. Gyakran a kimenő *port*-ok meg kell, hogy jegyezzenek bizonyos értékeket, ilyenkor a *port*-hoz tartozó adatot ***reg*** típusúra kell deklarálni.

A fenti példában minden *port*-hoz ***wire*** típusú adathordozó tartozik. A 7-4 példában megadott modulban (*D flip-flop*) a *q port*-hoz tartozó adat típusa ***reg***.

7-4 példa: A D flip-flop-ot leíró modul port-jainak deklarálása.

```
module DFF(q, d, clock, reset);

output q;               // A q port a kimenetként történő deklarálás folytán
                        // wire típusú lesz.
reg q;                  // Mivel a kimeneti értéket memorizálni kell, a port-hoz
                        // tartozó adat típusát új deklarációval meg kell
                        // változtatni reg típusúra.
input d, clock, reset;
//-------------------------------------
// A modul teste.
//-------------------------------------
endmodule
```

Fontos megjegyezni, hogy a bemeneti és a kétirányú *port*-okhoz tartozó adatokat nem deklarálhatjuk ***reg*** típusúként, mivel a ***reg*** típusú adatok értékeket memorizálnak, ami ellentétben áll a bemeneti *port*-ok szerepével (adatok közvetítése a külvilágból a modul belseje felé).

7.2.3. A *port*-ok összekötésének szabályai

A *port*-ot felfoghatjuk úgy, mint egy objektumot, amelynek két része van és ez a két rész össze van kapcsolva egymással. Az első rész a modul belsejéhez tartozik, míg a második rész a külvilághoz. Ez a felfogás megkönnyíti a modulok instanciálásánál érvényes szabályok megértését. A *HDL* leírást értelmező szoftver hibát jelez, ha nem tartjuk be ezeket a szabályokat. A szabályok a következők (7-4 ábra):

- **Bemenő *port*-ok**

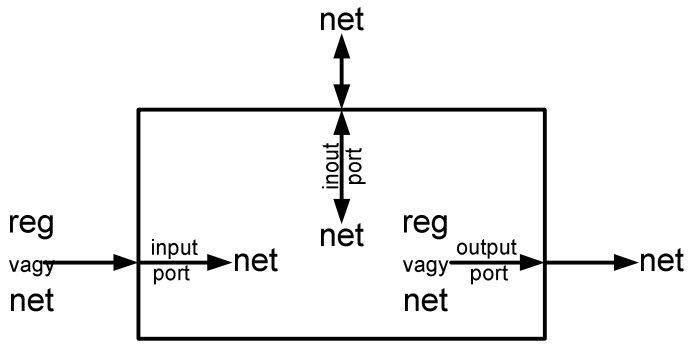
A bemenő *port*-okhoz mindig *net* (***wire***, ***wand***...) adattípus tartozik. Ezek a *port*-ok a környező modulok ***reg*** és *net* típusú adatait szállító *port*-ok-hoz köthetők.

- **Kimenő *port*-ok**

A modul kimenő *port*-jaihoz tartozó adatok lehetnek ***reg*** és *net* típusúak. Ezek a *port*-ok a csakis a környező modulok *net* típusú adatait szállító *port*-jaival köthetők össze.

- **Kétirányú *port*-ok**

A kétirányú *port*-okhoz csak *net* típusú adatokat rendelhetünk hozzá és csakis a szomszédos modulok *net* típusú adatait szállító *port*-okkal köthetők össze.



7-4 ábra: A por-tok összekötésének szabályai.

- **A bitek számának egyezése**

A *Verilog HDL* lehetővé teszi, hogy összekössünk olyan *port*-okat, amelyeknél a bitek száma nem azonos. Mindamellett ilyen esetben a leírást értelmező szoftver figyelmeztetni fog az eltérésre.

- **Össze nem kötött *port*-ok**

A *Verilog HDL* engedélyezi, hogy bizonyos *port*-ok szabadon maradjanak (ne kössük őket össze más *port*-okkal). Ilyen *port*-okon keresztül például információkat nyerhetünk a modul működéséről a kivizsgálás során, a hardveres megvalósításból viszont ezeket a *port*-okat ki fogjuk hagyni.

A 7-5 példában a 7-3 ábrán bevezettett *Top* nevű vizsgáló modul instanciálja a *full_add_4* nevű modult. A *port*-ok összekötése a két modul között többségében szabályos, a *SUM* adat viszont nem deklarálható ***reg*** típusúra a *Top* nevű modulban, mivel az instanciált modul kimeneti *port*-jához van rendelve, az ott jelentkező értékeket kell követnie, nem szabad megtartania korábbi értékeket.

7-5 példa: Nem szabályos kötést tartalmazó példa.

```
module Top;

// A Top modult és a full_add_4  modult összekötő port-ok deklarálása.
reg [3:0] A, B;
reg C_IN;
reg [3:0] SUM;              // Ez hibás deklaráció. Helyesen: wire [3:0] SUM;
wire C_OUT;

      // A full_add_4  modul instanciálása az egyedi fa0 instancia név alatt.

      full_add_4 fa0(SUM, C_OUT, A, B, C_IN);

      // Az instanciálás során egy szabályellenes összekötés történt.
      // A full_add_4  modul reg típusú SUM adatát hordozó port-ot összekötöttük
      // a Top nevű modul SUM adatával, amely szintén reg típusú.

      //-------------------------------------
      // A vizsgáló rész hardver nyelvi leírása.
      //-------------------------------------

endmodule
```

7.2.4. A modul *port*-jainak összekötése külső jelekkel

A *port* listában feltüntetett *port*-ok összekötése a külvilággal (más modulok *port*-jaival és belső adataival, jeleivel) két módon történhet. A két módszer nem keverhető.

- **Összekötés rendezett lista (angolul: *ordered list*) alapján**

A rendezett listán keresztül történő összekötés a könnyebben megérthető módszer a kezdő tervező számára. Az összekötendő adatokat azonos sorrendben adjuk meg a modul instanciálásakor és a modul definiálásakor. Vegyük újra a négy bites összeadó példáját a 7-3 ábráról.

A *HDL* leírást, amelyben a *full_add_4* modul *port*-jait a *Top* nevű modul megfelelő adataival rendezett lista alapján kötjük össze, a 7-6 példában láthatjuk. A *Top* modul *SUM, C_OUT, A, B* és *C_IN* nevű adatai ugyanolyan sorrendben jelennek meg a *full_add_4* modul instanciálásakor, mint a *sum, c_out, a, b* i *c_in port*-ok a *full_add_4* modul definiálásakor.

Fontos megjegyezni, hogy az adatok nevei és a *port*-ok elnevezései a rendezett lista alapján történő összekötésnél nem kell, hogy azonosak legyenek. A 7-6 példában a *sum* és a *SUM* két különböző azonosítónak számít, mivel a *Verilog HDL* megkülönbözteti a kis és a nagy betűket.

7-6 példa: Összekötés rendezett lista alapján.

```
module full_add_4(sum, c_out, a, b, c_in);

output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in;
        //-------------------------------------
        // A modul teste.
        //-------------------------------------
endmodule
```

```
module Top;

reg [3:0] A, B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;

        // A full_add_4 modul instanciálása az egyedi fa0 instancia név alatt.
        // Az adatokat a port-okkal a rendezett listában elfoglalt helyük alapján
        // azonosítjuk.

        full_add_4 fa0(SUM, C_OUT, A, B, C_IN);

        //-------------------------------------
        // A vizsgáló rész hardver nyelvi leírása.
        //-------------------------------------

endmodule
```

- ***Port*-ok összekötése a neveik alapján**

A nagy tervezési munkák során szükség lehet ötven vagy annál több *port*-ot tartalmazó modulra is. A *port*-ok sorrendjének követése ilyen esetben nehézkes és nagy a tévedés valószínűsége. Ennek a gondnak a megoldása érdekében a *Verilog HDL* lehetővé teszi a név alapján történő összekötést. Ez esetben a *port*-ok sorrendje lényegtelen. A 7-7 példában a korábban bevezetett négy bites összeadó *port*-jainak összekötését a *Top* modul megfelelő adataival a nevek alapján végeztük.

7-7 példa: Összekötés a port-ok neve alapján a 7-6 példa esetében.

```
full_add_4 fa0(.c_out(C_OUT), .sum(SUM), .b(B), .c_in(C_IN), .a(A));
```

Fontos megjegyezni, hogy nem szükséges minden egyes *port*-ot felsorolni a név alapján történő összekötéskor. A *port*-ok, amelyek feleslegesek az adott alkalmazásban, kihagyhatók. Ha a fenti példában a *c_out port* feleleges, az instanciálás a 7-8 példában leírt módon történhet.

7-8 példa: Nem teljes összakötés a port-ok neve alapján a 7-6 példa esetében ha a c_out port felesleges.

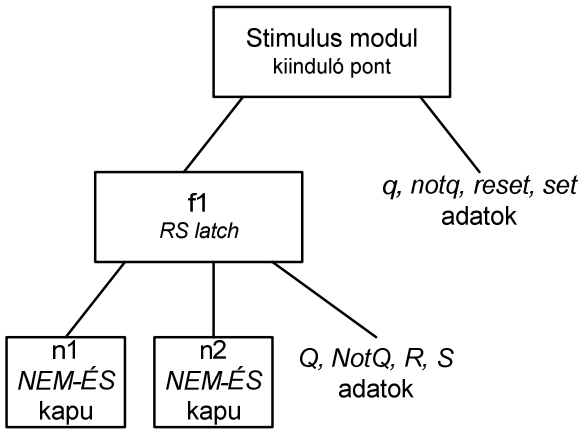
```
full_add_4 fa0(.sum(SUM), .b(B), .c_in(C_IN), .a(A));
```

7.3. Hierarchikus elnevezések

Az 5. fejezetben megismerkedtünk a *Verilog HDL* által támogatott hierarchikus tervezés fogalmával. A tervezés minden szintjén az egyes modul instanciákat, adatokat (jeleket) megfelelő azonosítókkal kell ellátni. A hierarchikus elnevezések (angolul: *hierarchical name*) lehetővé teszik, hogy a tervezés különböző szintjein előforduló azonosítókat egyedi névvel lássuk el. A hierarchikus elnevezés pontokkal elválasztott azonosítók sorából áll. Az elnevezést úgy alakítjuk ki, hogy minden egyes azonosítót pontosan meg lehessen címezni a *HDL leírás* bármelyik részéből.

A legmagasabb szintű modul a kiindulópont a hierarchikus nevek kialakításánál. A hierarchikus név a kiindulópont és az azonosító közötti elérési út (angolul: *path*) pontos leírását tartalmazza. A 7-1 példában leírt *RS latch*-en keresztül szemléltetjük a hierarchikus nevek képzésének módját. A fennálló hierarchikus szerkezetet a 7-5 ábrán láthatjuk.

A megadott tervben a hierarchia legmagasabb fokán a szimulációt leíró *Stimulus* nevű modul áll. Az ebben a modulban definiált azonosítók a *q, notq, reset* és *set*. A *Stimulus* modul instanciálja az *f1* modult, amit *RS_latch* név alatt definiáltunk. Az *f1* nevű instancia instanciálja az *n1* és *n2* elnevezésű *VAGY* kapukat. Az *f1* instanciában jelentkező adatok a *Q, NotQ, R* és *S*.



7-5 ábra: Az RS latch szimulációjára kialakított hierarchikus szerkezet.

A hierarchikus nevet úgy kapjuk, hogy kiinduló pont azonosítója mellé sorjában leírjuk az egyes alacsonyabb szintű instanciák neveit a kiválasztott azonosító felé vezető úton.

A 7-9 példában megadtuk a 7-5 ábrán előforduló azonosítók hierarchikus neveit. Ne felejtsük el, hogy az elnevezés egyes elemeit ponttal el kell választani egymástól.

7-9 példa: Hierarchikus elnevezések.

```
Stimulus.q          Stimulus.f1.Q
Stimulus.notq       Stimulus.f1.NotQ      Stimulus.f1
Stimulus.reset      Stimulus.f1.R         Stimulus.f1.n1
Stimulus.set        Stimulus.f1.S         Stimulus.f1.n2
```

8. *HDL* leírás logikai kapukkal

Az 5., 6. és 7. fejezetben megismerkedtünk a *Verilog HDL*-en alapuló digitális tervezés alapjaival: a tervezés menetével, a nyelvi szabályokkal, az adat típusokkal és a modulok szerkezetével. Megemlítettük, hogy a hardver nyelvi leírás négy lehetséges szinten történhet. Ebben a fejezetben a logikai kapuk szintjén (angolul: *gate level*) történő leírással foglalkozunk részletesen.

Ez a második elvonatkoztatási szint, a tervezések többsége ezen a szinten vagy ennél magasabb elvonatkoztatási szinten történik. A legalacsonyabb-, kapcsoló szint (angolul: *switch level*) gyakorlatilag csak az alkatrészgyártók tervezői számára ad kedvező lehetőségeket: pontosan végig lehet követni a késéseket és jól kiértékelhető a megvalósítás hatékonysága.

A logikai kapuk szintjén történő tervezés elsősorban azért fontos mert általa lehet hatékonyan átmenteni sok régi, bevált áramkört az *SSI* és *MSI* áramkörökkel történő tervezés világából a *PLD*-k világába. Másrészt, a hagyományos digitális tervezésben otthonosan mozgó tervező a kapuk szintjén történő hardver nyelvi leírásokra tér át legkönnyebben, mivel az egyes áramköri elemeknek egy az egyben megfelelő nyelvi szerkezetekkel történik a leírás. Ez az oka, hogy elsőként a logikai kapuk szintjén történő leírásokkal ismerkedünk meg.

8.1. A kapuk fajtái

A *Verilog HDL* támogatja az egyszerű logikai áramkörökből történő digitális tervezést. A logikai áramkörök előre definiált alapelemként (angolul: *primitive*) vannak jelen a hardver leíró nyelvben. Használatuk ugyanúgy instanciálással történik, mint a modulok esetében, azzal a különbséggel, hogy nem kell őket definiálni, mert a hardver leíró nyelv már tartalmazza ezeket a definíciókat. A logikai áramkörök két csoportja áll rendelkezésre:

- különböző *ÉS* kapuk és *VAGY* kapuk,
- illesztő áramkörök.

8.1.1. *ÉS* kapuk és *VAGY* kapuk

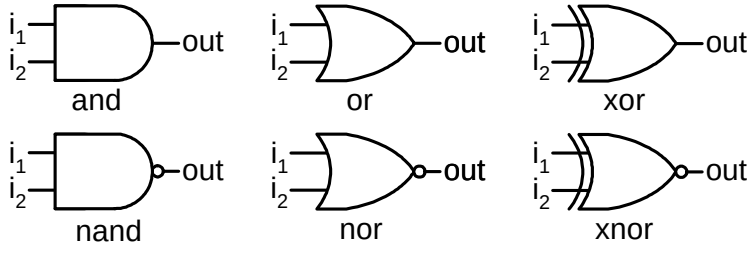
Az *ÉS* illetve a *VAGY* kapuknak egy egybites (skalár) kimenetük és kettő vagy több egybites bemenetük van. Instanciáláskor a *port* listában az első elem a kimenet, a többi *port* bemenet. A hardver leíró nyelvben definiált logikai áramkörök működési elve ugyanaz, mint a fizikailag megvalósított áramköröké: figyelik a bementeket és amint változás történik, kiszámítják és létrehozzák a kimeneten az új logikai szintet.

A *Verilog HDL*-ben definiált *ÉS* és *VAGY* kapuk modulneveit a 8-1 táblázatban láthatjuk. Ezeket a neveket mindig csak ugyanezen a módon írhatjuk, ahogyan a kulcsszavakat is.

8-1 táblázat: A Verilog HDL-ben definiált ÉS és VAGY kapuk modulnevei.

Modulnév	*Funkció*
and	*ÉS* kapu
nand	*NEM-ÉS* kapu
or	*VAGY* kapu
nor	*NEM-VAGY* kapu
xor	kizáró *VAGY* kapu
xnor	kizáró *NEM-VAGY* kapu

A 8-1 ábrán megadtuk az egyes kapuk rajzjeleit két bemenet esetére. A bemeneteket *i1*-gyel és *i2*-vel jelöltük, a kimeneti *port* neve *out*.



8-1 ábra: A Verilog HDL-ben definiált logikai kapuk rajzjelei és modulnevei.

A *Verilog HDL*-ben definiált logikai kapuk instanciálását a 8-1 példában szemléltetjük. Minden egyes logikai kapu kimenő *port*-ját (*out*) az *OUT* csomóponthoz kötöttük, az *in1* és *in2* bemenő *port*-okat pedig a *IN1* és *IN2* csomópontokhoz. A kimenetek összekötése egy modulon belül nem szokásos, de itt most ettől eltekintünk.

A 8-1 példában az első hat instancia kétbemenetű logikai kapuknak felel meg, a hetedik instancia egy hárombemenetű *NEM-ÉS* kapu. A *Verilog HDL*-ben az *ÉS* és a *VAGY* kapuknak nem csak két bemenetük lehet, hanem tetszőleges számú.

A példában a nyolcadik instanciának nem adtunk nevet. Logikai kapuk esetében ez is szabályos instanciálásnak számít: tetszőleges számú kaput instanciálhatunk, anélkül, hogy egynek is instancia nevet adnánk.

8-1 példa: Különböző ÉS és VAGY kapuk instanciálása.

```
wire OUT, IN1, IN2, IN3;

// Kétbemenetű logikai kapuk instanciálása instancia névvel.
and a1(OUT, IN1, IN2);
nand na1(OUT, IN1, IN2);
or or1(OUT, IN1, IN2);
nor nor1(OUT, IN1, IN2);
xor x1(OUT, IN1, IN2);
xnor nx1(OUT, IN1, IN2);

// Három bemenetű NEM-ÉS kapu instanciálása.
nand na1_3inp(OUT, IN1, IN2, IN3);

// ÉS kapu instanciálása instancia név nélkül (engedélyezett módszer).
and (OUT, IN1, IN2);
```

A kimeneti értékek számítása a *Verilog* hardver leíró nyelvben definiált logikai kapuknál megfelelő kombinációs táblázatok alapján történik. A 8-2 ábrán ezeket a táblázatokat adtuk meg kétbemenetű kapuk esetére. Több bemenetű kapuknál ugyanezeket a táblázatokat kell több lépésben (iteratív módon) alkalmazni.

and	i1			
	0	1	x	z
i2 0	0	0	0	0
i2 1	0	1	x	x
i2 x	0	x	x	x
i2 z	0	x	x	x

nand	i1			
	0	1	x	z
i2 0	1	1	1	1
i2 1	1	0	x	x
i2 x	1	x	x	x
i2 z	1	x	x	x

or	i1			
	0	1	x	z
i2 0	0	1	x	x
i2 1	1	1	1	1
i2 x	x	1	x	x
i2 z	x	1	x	x

nor	i1			
	0	1	x	z
i2 0	1	0	x	x
i2 1	0	0	0	0
i2 x	x	0	x	x
i2 z	x	0	x	x

xor	i1			
	0	1	x	z
i2 0	0	1	x	x
i2 1	1	0	x	x
i2 x	x	x	x	x
i2 z	x	x	x	x

xnor	i1			
	0	1	x	z
i2 0	1	0	x	x
i2 1	0	1	x	x
i2 x	x	x	x	x
i2 z	x	x	x	x

8-2 ábra: A Verilog HDL-ben definiált ÉS és VAGY kapuk kombinációs táblázatai.

A szokványos kombinációs táblázatokkal ellentétben itt megadtuk a kapu viselkedését ismeretlen bemeneti logikai szint illetve nagyimpedanciás bemenet esetére is. A fizikai megvalósítás esetén is megtörténik, hogy a logikai kapu bemenetén ismeretlen logikai szint vagy nagyimpedanciás állapot jelenik meg.

Minden ilyen esetben a logikai kapu a kimeneten vagy magas vagy alacsony logikai szintet fog létrehozni (köztes érték nem jellemző), csak sok esetben nem lehet tudni, hogy melyiket. A hardver leíró nyelvben alkalmazott táblázatok maximálisan követik a fizikailag megvalósítható áramkörök viselkedését: ott ahol tudni lehet a kimeneti logikai szintet, a táblázatban *0* vagy *1* szerepel, ahol viszont nem, ott *x* (ismeretlen, határozatlan) értéket írtunk.

8.1.2. Illesztő áramkörök

Ezeknek a logikai áramköröknek egy egybites (skalár) bemenő *port*-juk van és egy vagy több egybites kimenő *port*-juk. A *port* lista utolsó eleme a bemenet, a többi kimenet. A továbbiakban az egyszerűség kedvéért csak az egy kimenentes logikai elemekkel foglalkozunk. A több kimenet használata akkor célszerű, ha több kimeneti vonalat kell meghajtani.

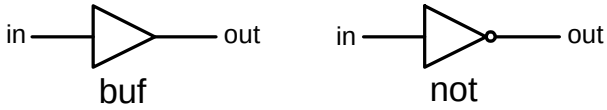
A *Verilog HDL*-ben két fajta (neminvertáló és invetáló) illesztő elem van definiálva. Ezen illesztő elemek modulneveit a 8-2 táblázatban láthatjuk. Ezeket a neveket mindig csak ugyanezen a módon írhatjuk, ahogyan a kulcsszavakat is.

8-2 táblázat: A Verilog HDL-ben definiált illesztő elemek modulnevei.

Modulnév	*Funkció*
buf	neminvertáló illesztő
not	invertáló illesztő

A 8-3 ábrán megadtuk az egyes illesztők rajzjeleit egy kimenet esetére. A bemeneti *port* neve *in*, a kimeneti *port* neve *out*.

8-3 ábra: Neminvertáló és invertáló illesztő rajzjelei és modulnevei.



A 8-2 példában láthatjuk a *Verilog HDL*-ben definiált illesztő elemek instanciálásának módját. Ismételten hagsúlyozzuk, hogy ezeknél az illesztőknél lehet több kimenet is, de csak egy bemenet lehetséges és ez a *port* listában az utolsó.

8-2 példa: A különböző illesztő elemek instanciálása.

```
// Egy kimenetű neminvertáló és invertáló elemek instanciálása.
buf b1(OUT, IN);
not n1(OUT, IN);

// Több kimenetű elem instanciálása.
buf b1_3out(OUT1, OUT2, OUT3, IN)

// Invertáló illesztő instanciálása instancia név nélkül (engedélyezett módszer).
not(OUT1, OUT2, IN);
```

Egy kimenet esetére a *Verilog HDL*-ben definált illesztő áramkörök viselkedése a 8-4 ábrán megadott kombinációs táblázatokkal jellemezhető.

8-4 ábra: Az illesztő áramkörök kombinációs táblázatai.

buf	
in	out
0	0
1	1
x	x
z	x

not	
in	out
0	1
1	0
x	x
z	x

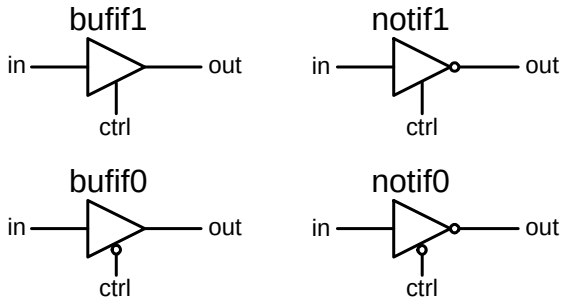
8.1.3. Háromállapotú illesztők

A hardver megoldásoknál (2.1.2 pont) használatos-, vezérlőbemenettel ellátott háromállapotú illesztőknek megfelelő logikai elemeket is definiáltak a *Verilog HDL*-ben. A különböző változatok modulneveit és funkcióit a 8-3 táblázatban összegeztük. Ezeket a modulneveket is változatlan formában kell használni.

Modulnév	*Funkció*
bufif1	neminvertáló illesztő, engedélyezés logikai egyessel
bufif0	neminvertáló illesztő, engedélyezés logikai nullával
notif1	invertáló illesztő, engedélyezés logikai egyessel
notif0	invertáló illesztő, engedélyezés logikai nullával

8-2 táblázat: A Verilog HDL-ben definiált háromállapotú illesztő elemek modulnevei.

Engedélyezés esetén ezek az illesztő elemek közönséges illesztőként működnek, engedélyezés híjján viszont a kimenetükön nagyimpedanciás állapot jelenik meg. Az egyes elemekre használatos rajzjeleket a 8-5 ábrán adtuk meg.



8-5 ábra: Három állapotú illesztő elemek rajzjelei.

A 8-3 példa a háromállapotú illesztők instanciálásának módját mutatja. Az első *port* a kimenet, a második a jel bemenet a harmadik a vezérlő bemenet. Az egyszerűség kedvéért itt is elhagyható az instancia név.

8-3 példa: A háromállapotú illesztők instanciálásának módjai.

```
// Neminvertáló illesztők instanciálása.
bufif1 b1(out, in, control);      // instancia névvel
bufif0 (out, in, control);        // instancia név nélkül

// Invertáló illesztők instanciálása.
notif1 n1(out, in, control);      // instancia névvel
notif0 (out, in, control);        // instancia név nélkül
```

A 8-6 ábrán a háromállapotú illesztők viselkedését leíró kombinációs táblázatok láthatók. A táblázatokban figyelembe vettünk minden eshetőséget a bementeket illetően (ismeretlen logikai szint és nagyimpedanciás állapot).

bufif1		ctrl			
		0	1	x	z
in	0	z	0	x	x
	1	z	1	x	x
	x	z	x	x	x
	z	z	x	x	x

notif1		ctrl			
		0	1	x	z
in	0	z	1	x	x
	1	z	0	x	x
	x	z	x	x	x
	z	z	x	x	x

bufif0		ctrl			
		0	1	x	z
in	0	0	z	x	x
	1	1	z	x	x
	x	x	z	x	x
	z	x	z	x	x

notif0		ctrl			
		0	1	x	z
in	0	1	z	x	x
	1	0	z	x	x
	x	x	z	x	x
	z	x	z	x	x

8-6 ábra: A Verilog HDL-ben definiált háromállapotú illesztő elemek kombinációs táblázatai.

A *Verilog HDL*-ben definiált háromállapotú illesztők szerepe azonos a megfelelő hardverelemek szerepével: elsősorban akkor alkalmazzuk őket, ha több tervezendő áramkör kimenete ugyanazt az átviteli vonalat kell, hogy használja különböző pillanatokban (időmultiplex). Ilyenkor a jeleket háromállapotú illesztőkkel csatoljuk az átviteli vonalra és az illesztők kimenetei közül egyidőben mindig csak egyet engedélyezünk. Így elkerüljük az egyes kimenetek ütközését.

Ütközés alatt azt értjük hogy az egyes kimenetek különböző logikai szinteket próbálnak létrehozni a közös vonalon. Ilyenkor az egyik kimenet nagy áramot ad, a másik nagy áramot nyel és nagy veszteség (melegedés) lép fel, esetenként az alkatrészek tönkre is mehetnek. Szintén nem elhanyagolható következmény, hogy ütközés esetén a kialakuló feszültségszint rendszerint nem szabályos logikai szintnek felel meg, ami az áramkör működésében bizonytalanságot okoz.

8.2. Példa a logikai kapuk szintjén történő tervezésre: négy bites teljes összeadó

A korábbiakban megismerkedtünk (4.2.1 pont) az egy bites félösszeadó és teljes összeadó fogalmával és láttuk, hogyan lehet teljes összeadók kaszkád kötésével több bites összeadókat kiépíteni. Ebben a szakaszban egy négy bites teljes összeadó hardver nyelvi leírását fogjuk elkészíteni.

A megfelelő *port* listát már megadtuk a 7-2 példában. A tervezéshez kizárólag logikai kapukat fogunk használni. A tervezés végén szerkesztünk egy vizsgáló modult, amellyel ellenőrizni tudjuk az összeadó modul működését. Felülről lefelé haladó hierarchikus tervezést fogunk alkalmazni: a négy bites összeadót négy darab egy bites teljes összeadóra bontjuk.

Az egy bites teljes összeadó kombinációs táblázatát a 8-7 ábrán láthatjuk. A jelölések a következők:

- *c_in (carry input)**:*** átviteli bit a bemeneten,
- *a, b:* a bemenetet képező egy bites bináris számok,
- *sum:* az egy bites *a* és *b* számok összege,
- *c_out (carry out)**:*** átviteli bit a kimeneten.

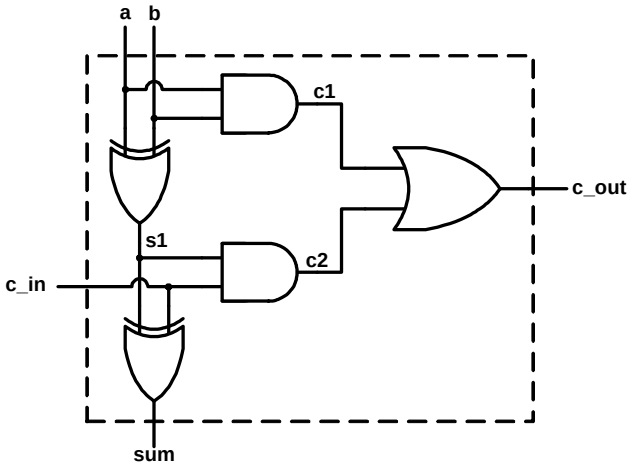
c_in	a	b	sum	c_out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

8-7 ábra: Az egy bites teljes összeadó kombinációs táblázata.

Az egyes kimeneti változók logikai egyenletei a következők:

$$\text{sum} = (a \oplus b \oplus c\_in)$$
$$c\_out = a \cdot b + c\_in \cdot (a \oplus b)$$

Az egyenletek alapján kialakított logikai hálózatot a 8-8 ábrán láthatjuk.



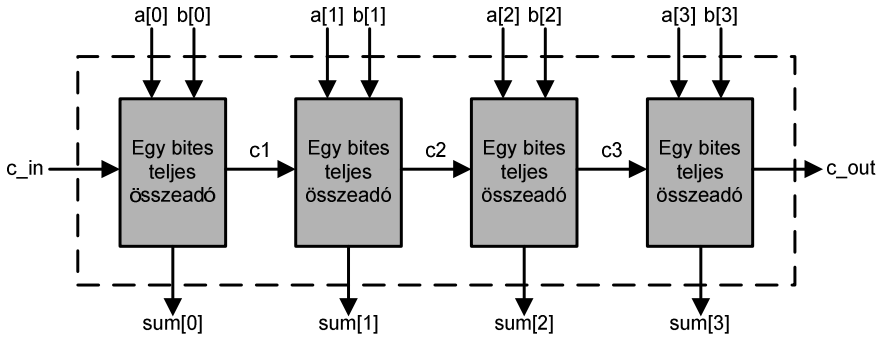
8-8 ábra: Az egy bites teljes összeadó logikai rajza.

A 8-4 példában a fenti logikai rajz alapján megadtuk az egy bites teljes összeadó hardver nyelvi leírását logikai kapuk szintjén. A logikai kapuk instanciálásakor nem használtunk instancia neveket. A belső csomópontok elnevezései azonosak a logikai rajzon alkalmazott elnevezésekkel.

8-4 példa: Az egy bites teljes összeadó Verilog HDL leírása.

```
module FullAdder1bit(sum, c_out, a, b, c_in);

// A port-ok deklarálása.
output sum, c_out;
input a, b, c_in;

// A belső csomópontok deklarálása.
wire s1, c1, c2;

// A logikai kapuk instanciálása (név nélkül).
xor (s1, a, b);
xor (sum, s1, c_in);

and (c1, a, b);
and (c2, s1, c_in);

or (c_out, c1, c2);

endmodule
```

A négy bites összeadót négy darab egy bites teljes összeadó kaszkád kötésével kapjuk a 8-9 ábra szerint.



8-9 ábra: A négy bites összeadó logikai rajza.

A logikai rajz alapján alakítottuk ki a 8-5 példában megadott *Verilog HDL* leírást. A modul a megfelelő deklarációk mellett a 8-4 példában tervezett egy bites teljes összeadó négy instanciáját tartalmazza.

8-5 példa: A négy bites összeadó Verilog HDL leírása.

```
module FullAdder4_bit(sum, c_out, a, b, c_in);

// A port-ok deklarálása.
output [3:0] sum;
output c_out;

input [3:0] a, b;
input c_in;

// A belső csomópontok deklarálása.
wire c1, c2, c3;

// A négy darab egy bites teljes összeadó instanciálása.
FullAdder1bit fa0(sum[0], c1, a[0], b[0], c_in);
```

```
FullAdder1bit fa1(sum[1], c2, a[1], b[1], c1);
FullAdder1bit fa2(sum[2], c3, a[2], b[2], c2);
FullAdder1bit fa3(sum[3], c_out, a[3], b[3], c3);

endmodule
```

Fontos megemlíteni, hogy több helyen is azonos neveket használtunk különböző adatokra. Például, az egy bites teljes összeadónál alkalmazott *sum* elnevezés az egy bites összeget jelöli, míg ugyanez a név a négy bites összeadónál négy bites vektorként van deklarálva. Az egyes modulok definíciójában alkalmazott nevek csak az adott modul számára hozzáférhetők, így nem történik keveredés. A modulban szereplő adatok kívülről csak a teljes hierarchikus név megadása mellett érhetők el.

Az egy bites teljes összeadók instanciálásakor minden instanciának egyedi nevet adtunk, ugyanez nem volt szükséges a logikai kapuk instanciálásakor ennek a modulnak a definiálásakor (8-4 példa).

A tervezés befejeztével meg kell győződnünk, hogy helyes-e az általunk kialakított hardver nyelvi leírás. Szükséges egy vizsgáló modul megszerkesztése, amely bemenő jeleket biztosít az összeadó modul számára és fogadja annak kimenő jeleit. A kimenő jelek elemzésével dönti el a tervező, hogy a modul szabályosan működik-e vagy sem.

A 8-6 példában adtuk meg a *Stimulus* nevű vizsgáló modul hardver nyelvi leírását.

8-6 példa: Az összeadó vizsgálatára kialakított Stimulus nevű modul leírása.

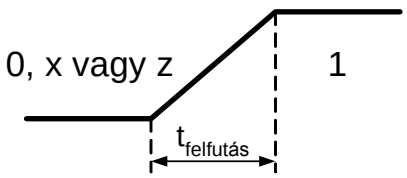
```
module Stimulus;

// A Stimulus modulban szereplő adatok deklarálása.
reg [3:0] A, B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;

// A négy bites összeadó instanciálása
FullAdder4_bit FA1(SUM, C_OUT, A, B, C_IN);

// Az összedó bemenő jeleinek definiálása.
initial
begin
        A = 4'd0;     B = 4'd0;     C_IN = 1'b0;

        #5      A = 4'd3;     B = 4'd4;
        #5      A = 4'd2;     B = 4'd5;
        #5      A = 4'd9;     B = 4'd9;
        #5      A = 4'd10;    B = 4'd15;
        #5      B = 4'd5;     C_IN = 1'b1;
end

endmodule
```

8.3. Késések

A logikai kapuk eddigi ismertetése során úgy tekintettük, hogy a kapuk működésében nem jelentkezik késés. A valós logikai áramköröknél mindig késik a kimenet megváltozása a bemeneti változáshoz képest. Ezt a hardver nyelvi leírásokban is figyelembe kell venni, hogy a tervezés és a szimuláció során hiteles eredményeket kapjunk. A *Verilog HDL*-ben definiált logikai kapuknál három féle késés definiálható, a következőkben ezekkel foglalkozunk.

- **Felfutási késés**

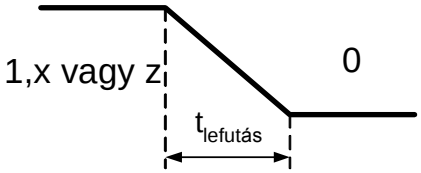
A felfutási késés (angolul: *rise delay*) az az idő, amely szükséges, hogy a logikai kapu kimeneti jelszintje *0, x* vagy *z* értékről logikai egyesre emelkedjen (8-10 ábra).



8-10 ábra: A felfutási késés értelmezése.

- **Lefutási késés**

A lefutási késés (angolul: *fall delay*)) az az idő, amely szükséges, hogy a logikai kapu kimeneti jelszintje *1, x* vagy *z* értékről logikai nulla szintre essen (8-11 ábra).



8-11 ábra: A lefutási késés értelmezése.

- **Kikapcsolási késés**

A kikapcsolási késés (angolul: *turn-off delay*) az az idő, amely szükséges, hogy a logikai áramkör kimenete *0, 1* vagy *x* értékről nagyimpedanciás állapotba (*z*) kerüljön.

Az ismert érték (0 vagy 1) utáni ismeretlen érték (*x*) megjelenéséhez szükséges időt nem definiálják külön paraméterrel a *Verilog HDL*-ben, hanem a fenti három késés közül a legkisebbet használja az értelmező- illetve a szimulációs szoftver.

A fent definiált késéseket három módon adhatjuk meg a *Verilog HDL*-ben.

- Egy értéket adunk meg és ezt az értéket használjuk minden egyes késésre.
- Két értéket adunk meg, a felfutási és a lefutási késést, a kikapcsolási késésre a két érték közül a kisebbet használjuk.
- Három értéket adunk meg: ezek sorrendben a felfutási, a lefutási és a kikapcsolási késés.

Ha semmilyen érték sincs megadva, az alapértelmezés szerint nulla értékeket kell venni (nincs késés). A késések megadására szolgáló nyelvi szerkezeteket a 8-7 példa definiálja.

8-7 példa: A késések megadására szolgáló nyelvi szerkezetek.

```
// Minden késési idő egyenlő a delay_time paraméter értékével.
and #(delay_time) a1(out1, in1, in2);

// A felfutási és a lefutási időre különböző értékeket adunk meg.
or #(rise_value, fall_value) o1(out2, in1, in2);

// A felfutási- a lefutási- és a kikapcsolási időre is különböző értékeket adunk meg.
bufif1 #(rise_val, fall_value, turn_off_value) b1(out3, in, control);
```

A 8-8 példában konkrét eseteket láthatunk a késések megadására. Fontos megemlíteni, hogy a megadott számértékek relatív értékek, nem kötődnek valamilyen konkrét mértékegységhez.

8-8 példa: Konkrét késéseket tartalmazó logikai kapuk instanciálása.

```
and #(5) a1(out1, in1, in2);          // Minden átmenet 5 időegységet vesz igénybe.
and #(4,6) a2(out2, in1, in2);        // felfutási idő = 4, lefutási idő = 6.
bufif0 #(3,4,5) b1(out3, in1, in2);   // felfutási idő = 3, lefutási idő = 4,
                                      // kikapcsolási idő = 5
```

8.3.1. Mininimális-, tipikus- és maximális értékek

A technika jelenlegi állása szerint nem lehetséges két integrált áramkör legyártása, amelyeknek azonosak lennének a jellemzőik. Ez miatt az egyes logikai kapuk késései is bizonyos tartományban variálnak. A minimális-, tipikus- és maximális késések bevezetésével lehetővé válik, a tervezett áramkör kivizsgálása minden várható esetre, függetlenül attól, hogy melyik konkrét logikai kaput fogjuk beépíteni.

Az egyes értékek értelme a következő:

- A minimális késési idő az a legrövidebb idő a bemeneti változás után, amelyet követően leghamarabb megjelenhet a kimeneti változás.
- A tipikus késési idő a késési idő várható (jellemző) értéke.
- A maximális késési idő az a leghosszabb idő a bemeneti változás után, amelyet követően biztosan meg fog történni a kimeneti változás.

A *Verilog HDL* lehetővé teszi minimális-, tipikus- és maximális késések definiálását egy konkrét időérték (pl. a felfutási késés) helyett. A szimuláció során utasítjuk a megfelelő szoftvert, hogy melyik értékeket vegye figyelembe, melyik értékek alapján rajzolja meg a diagramokat (ilyenkor minden instanciált kapunál pl. a minimális értékkel számol).

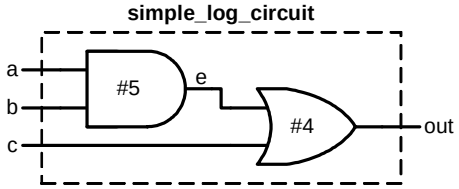
Ha egy késési időre csak egy értéket adunk meg, azt tipikus értéknek tekintjük. A 8-9 példában láthatjuk a minimális-, tipikus- és maximális késések megadásának módját.

8-9 példa: Minimális, tipikus és maximális késések definiálása.

```
and #(4:5:6) a1(out1, i1, i2);
// A  fenti esetben a felfutási-, lefutási- és kikapcsolási idők egyenlők:
// Min  = 4, Tip = 5, Max = 6.

and #(3:4:5, 5:6:7) a2(out2, in1, in2);
// A felfutási idők:          Min = 3, Tip = 4, Max = 5
// A lefutási idők:           Min = 6, Tip = 7, Max = 8
// A kikapcsolási idők        Min = 3, Tip = 4, Max = 5
// Emlékeztetünk, hogy kikapcsolási időként ez esetben a felfutási és a lefutási idők közül a
// kisebbet használnánk, mivel maga a kikapcsolási idő nincs megadva, ÉS kapunál azonban nincs
// kikapcsolási idő.

and #(2:3:4, 3:4:5, 4:5:6) a3(out3, in1, in2);
// Felfutási idők:            Min = 2, Tip = 3, Max = 4
// Lefutási idők:             Min = 3, Tip = 4, Max = 5
// Kikapcsolási idők::        Min = 4, Tip = 5, Max = 6
```

8.3.2. Példa késésekre

Tételezzük fel, hogy adott egy *simple_log_circuit* nevű modul, amely az *out=ab+c* logikai függvényt valósítja meg. A logikai kapukkal történő kivitelezés a 8-11 ábra szerint történhet. A 8-10 példában láthatjuk a megfelelő hardver nyelvi leírást a késésekkel.



8-12 ábra: A simple_log_circuit modulnak megfelelő logikai rajz.

8-10 példa: A simple_log_circuit modul hardver nyelvi leírása.

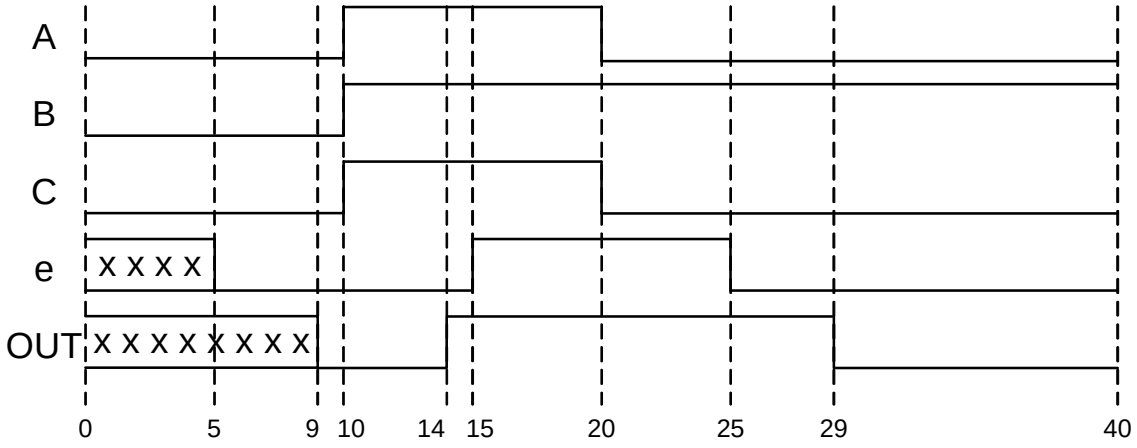
```
module simple_log_circuit(out, a, b, c);

// A port-ok deklarálása.
output out;
input a, b, c;

// Belső csomópontok (vezetékek) deklarálása.
wire e;

// A kombinációs hálózat megvalósításához szükséges logikai kapuk instanciálása.
and #(5) a1(e, a, b);       // 5 időegységnyi késés az ÉS logikai kapunál.
or #(4) o1(out, e, c);      // 4 időegységnyi késés a VAGY logikai kapunál.

endmodule
```

A modul kivizsgálására megírjuk a s*timulus* nevű vizsgáló modult (8-11 példa).

8-11 példa: A simple_log_circuit nevű modul kivizsgálására kialakított stimulus nevű modul.

```
module stimulus;

// A belső adathordozók deklarálása.
reg A, B, C;
wire OUT;

// A simple_log_circuit nevű modul instanciálása.
simple_log_circuit slc1(OUT, A, B, C);

// A bemeneti jelek definiálása a simple_log_circuit nevű modul részére.
// Az alkalmazott nyelvi szerkezeteket később tárgyaljuk.
// A szimuláció negyven időegység után fejeződik be.
initial
begin
            A = 1'b0;    B = 1'b0;    C = 1'b0;
      #10   A = 1'b1;    B = 1'b1;    C = 1'b1;
      #10   A = 1'b1;    B = 1'b0;    C = 1'b0;
      #20   $finish;
end

endmodule
```

A 8-13 ábrán láthatók a szimulációval kapott idődiagramok. Az *E* és az *OUT* változók logikai állapota a megfelelő késések letelte előtt nem kiértékelhető, így a szimulátor a diagram kezdetén *x* értéket jelölt be.



8-13 ábra: *A simple_log_circuit modul szimulációjával kapott idődiagramok.*

9. Adatfolyam szintű *HDL* leírás

A logikai kapuk szintjén kialakítandó *HDL* leírás akkor célszerű, ha egyszerű digitális áramköröket kívánunk létrehozni, amelyeknek a logikai rajza esetleg már korábbi hagyományos tervezésből ismert, vagy könnyen kialakítható. Ilyenkor a tervező számára a legegyszerűbb, hogy egy kapu szintű *Verilog HDL* leírásban instanciálja és összekösse a megfelelő logikai kapukat.

Összetettebb áramkörök tervezésekor, amikor nagy számú logikai kapuból áll össze a tervezett áramkör, a kapuszintű tervezés nem hatékony. A megoldás abban keresendő, hogy a tervezést egy magasabb elvonatkoztatási szintre emeljük, ahol a tervező az adatokon végrehajtandó műveletekre összpontosíthat, és nem köti le magát a logikai árakörökkel történő konkrét kivitelezéssel.

A *Verilog HDL*-ben az adatfolyam szintű tervezés egy magasabb elvonatkoztatási szinten folyik, ahol a hangsúly az adatok megfelelő áramlásán és feldolgozásán van, ahelyett, hogy az egyes logikai kapuk instanciálásával és összekötözésével foglalkoznánk.

A technológia fejlődésével mind nagyobb számú logikai kaput sikerül egy áramkörbe integrálni, ami megnövelte az adatfolyam szintű tervezés népszerűségét. Ma már egyetlen tervezéssel foglalkozó cég sem engedeti meg magának azt a fényűzést, hogy mérnökei idejét logikai kapuk szintjén történő tervezésre pazarolja.

Természetesen továbbra is minden digitális áramkör logikai kapukból épül fel, de ma a konkrét áramkör kialakítását a *HDL* leírás alapján megfelelő tervezői szoftverek végzik. Ezt a folyamatot (gépi) logikai szintézisnek nevezzük. A logikai szintézis annak köszönhetően vált népszerűvé, hogy hatékony tervezői szoftverek állnak rendelkezésre, úgy a *PLD* gyártók, mint független szoftverházak részéről. Ez lehetővé teszi, hogy a tervezők az áramkörben megvalósítandó adatáramlás optimalizálására összpontosítsanak, ne a hardveres részletkérdésekkel terheljék magukat.

Ma a legnagyobb tervezési rugalmasságot az itt ismertetendő adatfolyam szintű leírás és következő fejezetben sorra kerülő viselkedési szintű *HDL* leírás együttes alkalmazása biztosítja. A két leírási módszert közös néven a szakirodalomban *RTL* (*register transfer level*) tervezésnek nevezik.

9.1. Folyamatos hozzárendelés

A folyamatos hozzárendelés az alapvető módja az adatfolyam szinten, hogy valamely *net* típusú adathordozónak logikai értéket adjunk. A folyamatos hozzárendelés ilyen értelemben logikai kapukat helyettesít, de annál magasabb szintű absztrakciót tesz lehetővé, tetszőleges összetettségű kombinációs hálózatot ír le. Minden folyamatos hozzárendelés az ***assign*** kulcsszóval kezdődik. A megfelelő nyelvi szerkezetet a 9-1 példában láthatjuk.

9-1 példa: A folyamatos hozzárendelést végző nyelvi szerkezet.

```
assign <késés> <hozzárendelés>;
```

A késések megadása a folyamatos hozzárendeléseknél nem kötelező. Ha alkalmazzuk, akkor azt az időt adjuk meg, amelynek el kell telnie, hogy a hozzárendelés megtörténjen.

A folyamtos hozzárendelés jellemzői a következők:

- A kifejezés tartalmaz egy egyenlőségjelet. Az egyenlőségjel bal oldalán *net* típusú, skalár vagy vektor jellegű adathordozónak kell állnia, esetleg alkalmazható összecsatolt skalár vagy vektor jellegű, *net* típusú adathordozó. Az összecsatolásról 9.4.8 pont alatt lesz szó. Az egyenlőségjel bal oldalán nem állhat ***reg*** típusú adathordozó.
- A folyamatos hozzárendelések mindig aktívak. Bármilyen változás történik az egyenlőségjel jobb oldalán, azonnal kiértékelődik a bal oldal új értéke és azonnal, vagy az előírt késéssel, hozzárendelődik a megfelelő adathordozóhoz.

- Az egyenlőségjel jobb oldalán szereplő adathordozók lehetnek ***reg*** és *net* típusú adathordozók, valamint *Verilog HDL* függvények (ezekkel nem foglalkozunk ebben a jegyzetben. A *net* és a ***reg*** típusú adathordozók lehetnek skalár vagy vektor jellegűek.

A 9-2 példában folyamatos hozzárendelésekre láthatunk példákat. A *&*, ^, |, *{*, *}* és + műveleti jelek jelentését a 9.4 szakaszban fogjuk megmagyarázni. Most összpontosítsunk csak az ***assign*** nyelvi szerkezetre.

9.2 példa: Példák folyamatos hozzárendelésekre.

```
// Folyamatos hozzárendelés.  Az out, in1 és in2 adathordozók net típusúak.
assign out = in1 & in2;

// Folyamatos hozzárendelés net típusú adathordozó vektorokhoz.
// Az addr1_bits és addr2_bits adathordozók 16 bites reg típusú vektorok.
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];

// A folyamatos hozzárendelésben net típusú skalár és vektor adathordozót csatoltunk össze.
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

A következő szakaszban a *net* típusú adathordozókhoz történő folyamatos hozzárendelések írásának rövidebb módját tárgyaljuk.

9.1.1. Implicit folyamatos hozzárendelés

Ahelyett, hogy külön kifejezésben deklarálnánk a *net* típusú adathordozót és másik kifejezésben történne a folyamatos hozzárendelés, ezt egy lépésben is elvégezhetjük. A 9-3 példában láthatjuk a szabályos és a rövidített eljárás használatát.

9-3 példa: Szabályos és implicit folyamatos hozzárendelés.

```
// A folyamatos hozzárendelés írásának szabályos módja (először deklaráljuk az adathordozót)
wire out;
assign out = in1 + in2;

// Ugyanazt az eredményt érjük el implicit folyamatos hozzárendeléssel.
wire out = in1 + in2;
```

9.2. Késések a hozzárendelésekben

A folyamatos hozzárendelésekben alkalmazott késések meghatározzák, hogy mennyi idő alatt érvényesül az egyenlőségjel jobb oldalán történt változás az egyenlőségjel bal oldalán. A késések megadásának három módja létezik:

- késések a szabályos (***assign*** kulcsszóval végzett) hozzárendelésekben,
- késések az implicit hozzárendelésekben,
- a *net* típusú adathordozók deklarálásakor bevezetett késések.

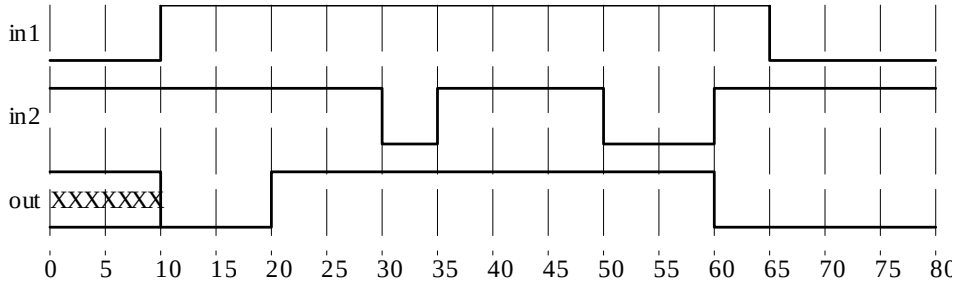
9.2.1. Késések a szabályos hozzárendelésekben

A késések megadásának első módja a szabályos hozzárendelésekben történő késés-magadás. A késési időre vonatkozó számértéket ez esetben az ***assign*** kulcsszó után írjuk. A 9-4 példában (*ÉS* logikai függvény megvalósítása) ezt a módot mutatjuk be.

9.4 példa: Késés megadása a szabályos hozzárendelésekben.

```
assign #10 out = in1 & in2;
```

A 9-4 példában az *in1* és *in2* bemeneti adathordozók logikai állapotának megváltozását követően az *out* adathordozónál a változás *10* időegység késéssel jelentkezik. A 9-1 ábrán láthatók a 9-4 pédában megadott kifejezésben szereplő adathordozók jeleinek időDiagramjai.



9-1 ábra: A 9-4 példában szereplő adathordozók időDiagramjai.

A diagramról a következők olvashatók le:

- A *10*-es időponttól kezdve ugyan a kifejezés jobb oldalán mindkét adathordozó logikai egyesen van, de a hatásuk csak *10* időegység múlva, a *20*-as időpontban jelentkezik, összhangban a 9-4 példában definált késéssel.
- Az *in2* adathordozón ugyan a *30*-tól *35* időegységig terjedő szakaszon logikai nulla jelenik meg, de ez nem okoz változást a kimeneten, mert a logikai nulla időtartama túl rövid (rövidebb a késésként megadott tíz időegységnél).
- A kifejezés jobb oldalán szereplő mindkét adathordozón logikai egyes van jelen a *60*-tól *65* időegységig terjedő szakaszon, de sem ekkor, sem a késés letelte után nem jelenik meg a kifejezés bal oldalán szereplő *out* változónál logikai egyes, mert a jobb oldalon megjelenő egyesek túl rövid ideig vannak jelen (a késésnél rövidebb ideig).
- Az *out* diagram kezdeti szakaszán (a késés letelte előtt) ismeretlen (*x*) érték van bejelölve, mert ezen a szakaszon nem kiértékelhető a kifejezés bal oldala (a szemlélt időszak előtti, számunkra ismeretlen értékektől függ)

A jelenséget, hogy a rövid ideig tartó logikai állapotok a folyamatos hozzárendelésekben nem okoznak változást az ***assign*** kifejezés bal oldalán, inerciális (tehetetlenségi) késésnek nevezzük. A logikai kapuk szintjén végzett tervezésnél is találkoztunk hasonló jelenséggel (8.3 szakasz).

9.2.2. Késések az implicit hozzárendelésekben

Említettük (9.1.1 pont), hogy a folyamatos hozzárendelések megadhatók implicit formában is, az adathordozók deklarálása közben. Ugyanitt késést is definiálhatunk. Erre vonatkozik a 9-5 példa.

9.5 példa: Az implitit hozzárendelésben megadott késés.

```
wire #10 out = in1 & in2;

// A fenti kifejezés hatása ugyanaz, mintha külön deklarálnánk az adathordozókat és
// külön definiálnánk a folyamatos hozzárendelést a megfelelő késéssel.
wire out;
assign #10 out = in1 & in2;
```

9.2.3. Késések megadása a *net* típusú adathordozók deklarálásakor

Lehet magára a *net* típusú adathordozókra késést megadni, minden hozzárendelés nélkül. Ha a késést a *net* típusú adathordozóra adjuk meg, minden esedékes változás ezen az adathordozón a definált késés leteltével jelenik meg. A deklaráláskor megadott késésre a 9-6 példában látható a megfelelő nyelvi szerkezet. Ezt a fajta késés megadást használhatjuk a logikai kapuk szintjén történő tervezésnél is.

9.6 példa: Késés definiálása a net típusú adathordozó deklarálásakor.

```
wire #10 out;
assign out = in1 & in2;

// A fenti kifejezés értelme megegyezik a lentivel:
wire out;
assign #10 out = in1 & in2;
```

Miután röviden megtárgyaltuk a folyamatos hozzárendelésekre vonatkozó alapvető szabályokat és a késéseket, most áttérünk a hozzárendelésekben alkalmazott kifejezések, műveleti jelek és operandusok tárgyalására.

9.3. Kifejezések, műveleti jelek és operandusok

Az adatfolyam szintű leírás logikai kapuk instanciálása helyett a digitális áramkört matematikai kifejezésekkel írja le. Ezek a kifejezések és a bennük előforduló operandusok és műveleti jelek képezik az adatfolyam szintű tervezés alapját.

9.3.1. Kifejezések

A kifejezések olyan hardver nyelvi szerkezetek, amelyek a megfelelő jelfeldolgozás érdekében operandusokat és műveleti jeleket kombinálnak (9-7 példa).

9-7 példa: Különböző operandusokat és műveleti jeleket tatalmazó három kifejezés.

```
a ^ b
addr1[7:4] + addr2[7:4]
in1 | in2
```

9.3.2. Operandusok

Operandusként általánosságban alkalmazhatjuk a 6.2 szakaszban deklarált adathordozók bármelyik típusát. Egyes kifejezések csak bizonyos típusú adathordozókat fogadnak el. Tehát az operandus lehet egy konstans, egész szám (***integer***), valós szám (***real***), *net*, ***reg***, vektor, vagy vektor része (egy vagy több bit) vagy függvényérték (mint említettük, a *Verilog HDL* függvényekkel nem foglalkozunk ebben a jegyzetben). A 9-8 példában az operandusok használatát szemléltetjük.

9.8 példa: A különböző típusú operandusok használata.

```
integer count, final_count;
final_count = count + 1;              // A count adathordozó integer típusú

real a, b, c;
c = a – b;                            // Az a és b adathordozók real típusúak.

reg [7:0] reg1, reg2;
reg [3:0] reg_out;
reg_out = reg1 [3:0] ^ reg2[3:0];     // A reg1[3:0] és reg2[3:0] operandusok a reg1 és a
                                      //reg2 vektor jellegű változók részeit képezik.
```

9.3.3. Műveleti jelek

Az operandusokon műveleteket végzünk, hogy a kívánt eredményhez jussunk. A műveleteket műveleti jelekkel definiáljuk. A *Verilog HDL*-ben számos műveleti jel áll rendelkezésünkre (9-9 példa). A 9.4 szakaszban a műveleti jelekkel ismerkedünk meg részletesen.

9-9 példa: A műveleti jelek alkalmazása különböző kifejezésekben.

```
d1 && d2    // A && egy bináris műveleti jel d1 és d2 operandusokkal.
!a[0]       // A ! egy unáris műveleti jel a[0] operandussal.
C >> 2      // A >> egy bináris műveleti jel C és 2 operandusokkal.
```

9.4. A műveleti jelek fajtái

A *Verilog HDL*-ben nagy számú különböző műveleti jel van definiálva. Vannak aritmetikai-, logikai-, viszonyító-, redukáló-, eltolást végző-, összecsatoló- és feltételes műveletek. Némelyek hasonlítanak a *C* nyelvben alkalmazott műveletekre. Minden műveletet meghatározott jellel jelölünk. A 9-1 táblázatban megadtuk a műveleti jeleket, megfelelő osztályokba sorolva.

A művelet osztálya	A műveleti jel	Művelet	Az operandusok száma
Aritmetikai	*	szorzás	2
	/	osztás	2
	+	összeadás	2
	-	kivonás	2
	%	osztás modul szerint	2
Logikai	!	logikai tagadás	1
	&&	logiki ÉS	2
	||	logikai VAGY	2
Viszonyító	>	nagyobb mint	2
	<	kisebb mint	2
	> =	nagyobb vagy egyenlő	2
	< =	kisebb vagy egyenlő	2
Egyenlőségi	= =	egyenlőség	2
	!=	egyenlőtlenség	2
	= = =	case egyenlőség	2
	!= =	case egyenlőtlenség	2
Bitre vonatkozó	~	bit tagadás	1
	&	bit ÉS	2
	|	bit VAGY	2
	^	bit kizáró VAGY	2
	^~ ili ~^	bit kizáró NEM-VAGY	2
Redukáló	&	redukáló ÉS	1
	~&	redukáló NEM-ÉS	1
	|	redukáló VAGY	1
	~|	redukáló NEM-VAGY	1
	^	redukáló kizáró VAGY	1
	^~ ili ~^	redukáló kizáró NEM-VAGY	1
Eltoló	>>	eltolás jobbra	2
	<<	eltolás balra	2
Összecsatolás és többszörözés	{ }	összecsatolás	tetszőleges számú
	{ { } }	többszörözés	tetszőleges számú
Feltételes	? :	feltételes	3

9-1 táblázat: Az összes műveleti jel megfelelő osztályokba sorolva.

9.4.1. Aritmetikai műveleti jelek

Az aritmetikai (számítási) műveletek vonatkozhatnak egy operandusra (unáris műveletek) vagy két operandusra (bináris műveletek).

Bináris műveleti jelek

A bináris műveletek közé soroljuk a szorzást (jele *****), osztást (jele /), összeadást (jele **+**), kivonást (jele **-**) i modul szerinti osztást (jele **%**). A 9-10 példában a bináris műveleti jelek alkalmazását láthatjuk.

9-10 ábra: A két operandusra vonatkozó (bináris) aritmetikai műveleti jelek használata.

```
A = 4'b0011; B = 4'b0100;       // A és B reg típusú vektorok.
D = 6; E = 4;                   // D és E integer típusú adathordozók.

A * B               // A és B szorzása. Az eredmény  4'b1100.
D / E               // D osztása E-vel. A hányados 1. A maradékot figyelmen kívül hagyjuk.
A + B               // A és B összeadása.  Az összeg 4'b0111.
B – A               // A kivonása B-ből.  A különbség 4'b0001.

// Ha az operandus egy vagy több bitje ismeretlen értékű (x), az aritmetikai művelet
// eredménye teljes egészében ismeretlennek tekintendő (minden bit x).
in1 = 4'b101x;
in2 = 4'b1011;

sum = in1 + in2;    // Az összeg  4'bx lesz.

// A modul szerinti osztási művelet eredménye a maradék.
13 % 3          // Az eredmény 1.
16 % 4          // Az eredmény 0.
-7 % 2          // Az eredmény -1, mert az osztandó előjele negatív.
7 % -2          // Az eredmény 1, mert  az osztandó előjele pozitív.
```

Unáris műveleti jelek

A + és a – műveleti jeleket használhatjuk unáris értelemben is (9-11 példa), ekkor az operandus előjelét határozzák meg. Az unáris értelemben használt + és – műveleti jelek prioritást élveznek a bináris értelemben használt + és – jelekkel szemben.

9-11 példa: Az unáris + és – jelek használata.

```
-4      // Minusz 4.
+5      // Plusz 5.
```

Negatív számok használata csak az ***integer*** és ***real*** típusú adathordozók esetében tanácsolható. A *<méret>'<a számrendszer alapja><szám>* alakú számoknál kerülni kell a negatív előjel használatát, mert a *Verilog HDL* ezeket a számokat kettes komplemens alakra hozza, ami nem várt ereménnyel járhat (9-12 példa).

9-12 példa: Nem várt eredmény <méret>'<a számrendszer alapja><szám> alakú negatív előjelű szám osztásánál.

```
// Negatív előjelű számokhoz inkább az integer és a real típusú adathordozók
// használata tanácsos.
// Integer vagy real típusú adathordozó esetén a várt eredményt kapjuk.
-10 / 5         // Az eredmény -2.

// Kerüljük a <méret>'<a számrendszer alapja><szám> alakot a negatív számok írásánál.
```

```
-'d10 / 5      // Az eredmény: (10 kettes komplemense) / 5 = (2^32 – 10) / 5
               // ahol feltételeztük, hogy az alkalmazott operációs rendszerben a szóhossz 32.
               // Nem várt, téves eredményre jutottunk.
```

9.4.2. Logikai műveleti jelek

A *Verilog HDL*-ben végezhető logikai műveletek az *ÉS* (jele *&&*), a *VAGY* (jele ||) és a *NEM* (jele *!*). A *&&* és a || műveleti jelek két operandussal dolgoznak (bináris műveletek), míg a *!* jel egy operandusra vonatkozik (unáris művelet). A logikai műveletekre a következő szabályok érvényesek:

- A logikai műveletek eredménye mindig egy bit, az érték lehet *0* (hamis), *1* (igaz) és *x* (határozatlan).
- Ha a logikai operandusként használt adat értéke nullától különböző, értékét a logikai műveletekben egyesnek (igaz) vesszük, ha viszont egyenlő nullával, akkor a logikai érték nullának (hamis) tekintjük. Ha az operandus egy vagy több bitje *x* vagy *z*, a logikai értéket x-nek (határozatlan) vesszük.
- A logikai műveletekben az operandusok lehetnek adathordozók vagy kifejezések.

A *HDL* leírás áttekinthetőségének növelése érdekében tanácsos az egyes kifejezéseket zárójelbe tenni. A másik, amely a zárójelek használatából ered, hogy nem kell ügyelnünk a műveletek prioritására.

9-13 példa: A logikai műveleti jelek használata

```
// Alapesetek.
A = 3; B = 0;
A && B                  // Az eredmény: ( 1 && 0 ) = 0.
A || B                  // Az eredmény : (1 || 0 ) = 1.
!A                      // Az eredmény : NEM( 1 ) = 0.
!B                      // Az eredmény : NEM( 0 ) = 1.

// Határozatlan operandusok alkalmazása.
A = 2'b0x; B = 2'b10;
A && B                  // Az eredmény: ( x && 1 ) = x.

// Kifejezések használata operandusként.
(a == 1 ) && ( b == 3 )  // Az eredmény 1 ha mindkét kifejezés logikai értéke 1.
                         // Az erdemény 0, ha legalább az egyik kifejezés logikai értéke 0.
```

9.4.3. Viszonyító műveleti jelek

A viszonyító műveletek a nagyobb mint (jele >), kisebb mint (jele <), nagyobb vagy egyenlő (jele >=), kisebb vagy egyenlő (jele <=). A viszonyító művelet eredménye 1 (igaz) ha a megfelelő állítás helyes és 0 (hamis), ha a műveletben szereplő állítás helyetelen. A viszonyító műveletek esetében is, ha legalább az egyik operandus *x* vagy *z* értékű bitet (biteket) tartalmaz, az eredmény x (határozatlan). Ezek a műveleti jelek ugyanúgy működnek, mint a megfelelő műveleti jelek a *C* szoftver nyelvben. A 9-14 példában a viszonyító műveleti jelek használatát láthatjuk.

9-14 példa: A viszonyító műveleti jelek használata.

```
A = 4; B = 2;
X = 4'b1010; Y = 4'b1011; Z = 4'b1xxx;

A <= B;      // Az eredmény 0.
A > B;       // Az eredmény 1.
Y <= X;      // Az eredmény 0.
Y < Z;       // Az eredmény x.
```

9.4.4. Egyenlőségi műveleti jelek

A műveleteknek ebbe a csoportjába tartoznak a logikai egyenlőség (jele ==), logikai egyenlőtlenség (jele ***!***=), a *case* egyenlőség (jele ===) és a *case* egyenlőtlenség (jele ***!***==). Az egyenlőtlenségi művelet eredménye *1* (igaz) ha a megfelelő állítás helyes és *0* (hamis), ha a műveletben szereplő állítás helyetelen (9-2 táblázat). Ezek a műveletek során az egyes operandusokat bitenként hasonlítjuk össze. Ha a két operandus szóhossza különböző, a rövidebbet nullákkal töltjük fel a magasabb helyi értékek felől az egyenlőségi műveletek elvégzése előtt.

Kifejezés	Értelmezés	A kapható logikai értékek
a == b	*a* egyenlő *b*-vel, az eredmény határozatlan ha az operandusok bitjei között x vagy z szerepel.	0, 1, x
a != b	*a* nem egyenlő *b*-vel, az eredmény határozatlan ha az operandusok bitjei között *x* vagy *z* szerepel.	0, 1, x
a === b	*a* és *b* egyenlőségét vizsgálva az *x* illetve *z* értékű biteket is figyelembe vesszük.	0, 1
a !== b	*a* és *b* egyenlőtlenségét vizsgálva az *x* illetve *z* értékű biteket is figyelembe vesszük.	0, 1

9-2 táblázat: Az egyenlőségi műveletek értelemzése.

Fontos kihangsúlyozni a logikai egyenlőség (egyenlőtlenség) és a *case* egyenlőség (egyenlőtlenség) közötti különbséget (9-15 példa). A logikai egyenlőség eredményeként *x* értéket kapunk ha az operandusok bitjei között akár egy *x* vagy z érték szerepel, viszont a *case* egyenlőség esetében az x és z értékeket is figyelembe véve, bitenként ellenőrizzük a két operandust és sohasem kapunk határozatlan eredményt.

9-15 példa: Egyenlőségi műveletek alkalmazása.

```
A = 4; B = 3;
X = 4'B1010; Y = 4'b1101;
Z = 4'b1xxz; M = 4'b1xxz; N = 4'b1xxx;

A == B      // Az eredmény 0.
X != Y      // Az eredmény 1.
X == Z      // Az eredmény x.
Z === M     // Az eredmény 1 mert minden bit megegyezik, még az x és z értékűek is.
Z === N     // Az eredmény 0 mert a legkisebb helyi értékű bitek különböznek.
M !== N     // Az eredmény 1.
```

9.4.5. Bitenként végzett logikai műveletek jelei

A bitenként végzett műveletek a *NEM* (jele ~), *ÉS* (jele &), vagy (jele |), kizáró *VAGY* (jele ^), és a kizáró *NEM-VAGY* (jele ^~ vagy ~^) műveletek. Ezek a műveletek az operandus(ok) egyes bitjeivel dolgoznak. A NEM művelet esetén az operandus bitjeit egyenként negáljuk. A többi művelet esetében a két operandus megfelelő bitjein páronként végezzük az előírt műveletet. Ha a két operandus bitjeinek száma különböző, a kisebbik nullákkal egészítődik ki. A 9-16 példában a bitenkénti műveleteket szemléltetjük.

9-16 példa: A bitenkénti műveletek szemléltetése.

```
X = 4'b1010;      Y = 4'b1101 ;      Z = 4'b10x1;

~X          // Bitenkénti NEM művelet. Az eredmény 4'b0101.
X & Y       // Bitenkénti ÉS művelet. Az eredmény 4'b1000.
X | Y       // Bitenkénti VAGY művelet. Az eredmény 4'b1111.
```

```
X ^ Y      // Bitenkénti kizáró VAGY művelet. Az eredmény 4'b0111.
X ^~ Y     // Bitenkénti kizáró nem VAGY művelet. Az eredmény 4'b1000.
X & Z      // Bitenkénti ÉS művelet. Az eredmény 4'b10x0.
```

A bitenkénti műveleteknél érvényes kombinációs táblázatokat a 9-2 ábrán tüntettük fel. Ezeknél a műveleteknél az operandus bitjének esetleges *z* értékét *x*-nek tekintjük (lényegében ezt tettük a logikai kapuk szintjén történő tervezésnél is (8. fejezet).

	0	1	x
0	0	0	x
1	0	1	x
x	x	x	x

Bitenkénti ÉS muvelet

	0	1	x
0	0	1	x
1	1	1	x
x	x	x	x

Bitenkénti VAGY muvelet

A	~A
0	1
1	0
x	x

Bitenkénti NEM muvelet

	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

Bitenkénti kizáró VAGY muvelet

	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

Bitenkénti kizáró NEM-VAGY muvelet

9-2 ábra: A bitenkénti műveletekre vonatkozó kombinációs táblázatok.

Fontos kihangsúlyozni a bitenkénti műveletek *(~, &,* | műveleti jelek) és a logikai műveletek közötti különbséget *(!, &&,* || műveleti jelek). A logikai műveletek az operandust egyetlen logikai változónak tekintik, függetlenül a bitek számától, míg a bitenkénti műveleteknél a művelet az egyes bitekre vonatkozik (9-17 példa).

9-17 példa: A bitenkénti műveletek és a logikai műveletek közötti különbségek.

```
X = 4'b1010;        Y = 4'b0000;

X | Y;      // Bitenkénti VAGY művelet. Az eredmény 4'b1010;
X || Y;     // Logikai VAGY művelet. A kirtékelés a következő egyenlőség szerint
            // történik: 1 || 0 = 1.
```

9.4.6. Redukáló műveleti jelek

A redukáló műveletek az *ÉS* (jele *&*), a *NEM-ÉS* (jele *~&*), a *VAGY* (jele *|*), a NEM-VAGY (jele *~|*), a kizáró VAGY (jele *^*) és a kizáró NEM-VAGY (*~^* vagy *^~*). A redukáló műveleti jeleket mindig csak egy operandusra alkalmazzuk (unáris műveletek). A művelet az operandus egyes bitjeire vonatkozik és eredményként mindig egy bit.

A redukáló műveletek kombinációs táblázatai megegyeznek a bitenkénti műveletek kombinációs táblázataival (9-2 ábra). A két műveletcsoport között a különbség, hogy a redukáló műveleteket egy operandus egymást követű bitjeire alkalmazzuk, míg a bitenkénti műveleteknél az operandus vag operandusok adott helyi értékű bitjén (bitjein) történik az előírt művelet. A redukáló műveletek végzése az operandus legnagyobb helyi értékű bitjénél kezdődik és a legkisebb helyi értékű bitnél fejeződik be. A 9-18 példa ezzel kapcsolatban ad eligazítást.

9-18 példa: Redukáló műveletek szemléltetése.

```
X = 4'b1010;

&X      // A kiértékelés a következő egyenlet szerint történik: 1 & 0 & 1 & 0 = 1'b0.
|X      // A kiértékelés a következő egyenlet szerint történik: 1 | 0 | 1 | 0 = 1'b1.
^X      // A kiértékelés a következő egyenlet szerint történik: 1 ^ 0 ^ 1 ^ 0 = 1'b0.

// A kizáró VAGY és  a kizáró NEM-VAGY műveletek felhasználhatók adott vektor
// párossági vagy páratlansági bitjének képzésére.
```

9.4.7. Eltoló műveleti jelek

Az eltoló műveletek a jobbra tolás (jele >>) és a balra tolás (jele <<). Ezek a műveletek egy vektor jellegű változó bitjeit tolják jobbra vagy balra a megadott számú hellyel. A műveleti jelhez két operandus tartozik: maga a vektor, amit el kell tolni és a szám amely az eltolás mértékét határozza meg. A megüresedő bitek helyére nullák íródnak be, nem kerülnek vissza a vektor másik végén kitolt értékek. Az eltoló műveletek használatát a 9-19 példában láthatjuk.

9-19 példa: Az eltoló műveletek használata.

```
X = 4'b1100;

Y = X >> 1;  // Y = 4'b0110. Az X vektor tartalma egy hellyel jobbra tolódik és
             // a legnagyobb helyi értékű helyre nulla íródik be.
Y = X << 1;  // Y = 4'b1000. Az X vektor tartalma egy hellyel balra tolódik és
             // a legkisebb helyi értékű helyre nulla íródik be.
Y = X << 2;  // Y = 4'b0000. Az X vektor tartalma két hellyel balra tolódik és
             // a két legkisebb helyi értékű helyre nulla íródik be.
```

Az eltoló műveletek számos lehetséges alkalmazása közül kiemeljük a szorzási algoritmusokat (eltolásokra és összeadásokra vezetik vissza a szorzást) és a híradástechnikában alkalmazott kódolási algoritmusokat.

9.4.8. Összecsatoló műveleti jel

Összecsatoló művelettel (jele *{ }*) két vagy több operandust lehet egy operandussá egyesíteni. Az operandusokként használt vektorok hossza előre definálva kell, hogy legyen. Ellenkező esetben lehetetlen volna az eredő vektor hosszának meghatározása.

Az összecsatolást definiáló kifejezésben az egyes összecsatolandó elemeket egy kapcsos zárójelben sorakoztatjuk fel, vesszővel elválasztva őket egymástól. Az összecsatolandó elemek (operandusok) lehetnek *net* vagy ***reg*** típusú skalárok, *net* vagy ***reg*** típusú vektorok vagy ilyen vektorok részei vagy definiált hosszúságú állandók (9-20 példa).

9-20 példa: Összecsatoló műveletek szemléltetése.

```
A = 1'b1;    B = 2'b00;   C = 2'b10;   D = 3'b110;

Y = { B, C };                  // Y = 4'b0010
Y = { A, B, C, D, 3'b001 };    // Y = 11'b10010110001
Y = { A, B[0], C[1] };         // Y = 3'b101
```

9.4.9. Többszöröző műveleti jel

A többszöröző művelet (jele *{ { } }*) két operandussal dolgozik. Az első operandus egy konstans ez határozza meg, hogy hányszor kell egymás után írni a másik operandust (9-21 példa). A többszöröző és az összecsatoló műveletek egy kifejezésbe is kombinálhatók a következő módon:

{ $const_1${$adat_1$}, $const_2${$adat_2$}, ..., $const_n${$adat_n$} }

9-21 példa: Többszöröző műveletek alkalmazása összecsatolással kombinálva.

```
reg A;
reg [1:0] B, C;
A = 1'b1;     B = 2'b00;   C = 2'b10;

Y = { 4{A} };              // Y = 4'b1111
Y = { 4{A}, 2{B} };        // Y = 8'b11110000
Y = { 4{A}, 2{B}, C };     // Y = 10'b1111000010
```

9.4.10. Feltételes műveleti jel

A feltételes művelet (jele ***?:***) három operandussal dolgozik. Az alkalmazás a következő kifejezés szerint történik:

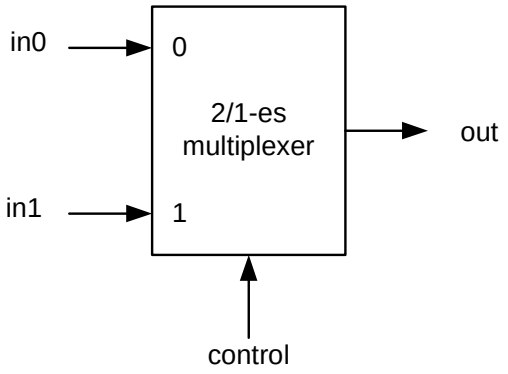
feltétel ? igaz feltételhez kötött kifejezés : hamis feltételhez kötött kifejezés;

A feltételes művelet végzésekor először a *feltétel* logikai értéke határozódik meg. Ha *feltétel* igaz, az erre az esetre esedékes kifejezés értékelődik ki, ha viszont hamis, akkor a hamis feltétel esetén érvényes kifejezés kerül sorra.

Ha a feltétel tetszőleges (*x*) értékű, akkor először is kiértékelődik mindkét kifejezés, majd az így kapott két értékből állítódik elő a művelet végeredménye. A végeredmény kiértékelése ilyenkor bitenként történik: ha a megfelelő helyi értékű bitek a két kifejezésből kapott értékekben megegyeznek, akkor ez az érték kerül az eredmény megfelelő helyi értékű pozíciójára, ha különböznek, akkor az adott helyre *x* érték kerül.

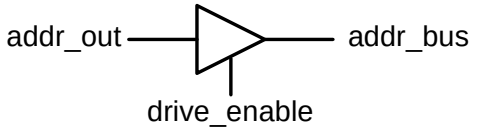
A feltételes műveleteket az adatfolyam szintű tervezéseknél rendszerint feltételes hozzárendelésekre használjuk: a feltétel dönti el, hogy melyik kifejezés értéke rendelődik hozzá az ***assign*** kifejezés bal oldalán levő adathordozóhoz.

A feltételes hozzárendelés úgy működik, mint egy *2/1*-es digitális multiplexer (9-3 ábra, 9-22 példa), így a feltételes művelet multiplexer megvalósítására is használható.



9-3 ábra: Feltételes hozzárendelés hatásának szemléltetése 2/1-es digitális multiplexerrel.

A multiplexer megvalósítása mellett az adatfolyam szintű tervezéseknél a háromállapotú illesztőket is feltételes utasítással valósítjuk meg (9-4 ábra , 9-22 példa).



9-4 ábra: Három állapotú illesztő a 9-22 példában használt adathordozó nevekkel.

9-22 példa: Multiplexer és három állapotú illesztő megvalósítása feltételes utasítással.

```
// 2/1-es multiplexer (9-3 ábra) megvalósítása feltételes utasítással.
assign out = control ? in1 ? in0;

// Három állapotú illesztő (9-4 ábra) megvalósítása feltételes utasítással.
assign addr_bus = drive_enable ? addr_out : 36'bz;
```

A feltételes utasítás használható iteratív módon is. Az utasításban szereplő egyes kifejezések maguk is lehetnek feltételes utasítások. Ezt a lehetőséget egy példán szemléltetjük.

A feladat, hogy két kétbites adatot (*A* és *B*) vizsgáljunk meg, és adjunk a kimenetre logikai egyest, ha mindkét adat páros vagy mindkettő páratlan, ellenkező esetben legyen a válasz logikai nulla. Elegendő a két adat kisebb helyi értékű bitjeit megvizsgálni a 9-23 példában megadott módon két lépésben (iteratív módon) alkalmazott feltételes utasításokkal.

9-23 példa: Iteratív módon alkalmazott feltételes utasítások.

```
reg [1:0] A, B;          // A és B két bites reg típusú adathordozók.

assign out = A[0] ? ( B[0] ? 1'b1 : 1'b0 ) : ( B[0] ? 1'b0 : 1'b1);
```

9.4.11. A műveletek hierarchiája

Ha nem használunk zárójeleket, a kijelölt műveletek elvégzésének sorrendjét a *Verilog HDL* leírást értelező szoftver a 9-2 táblázat alapján határozza meg. A zárójelek használata tanácsos, így külön választhatók az egyes kifejezések és elkerülhetők a műveletek prioritásával kapcsolatos félreértések. Egyedül az unáris műveleteknél nincs gond a prioritással.

Műveletek típusa	Műveleti jelek	Prioritás
unáris műveletek	+ - ! ~	Legmagasabb prioritás
szorzás, osztás, modul szerinti osztás	* / %	
összeadás, kivonás	+ -	
eltolás	<< >>	
viszonyító műveletek	< <= > >=	
egyenlőségi műveletek	== != === !==	
redukáló műveletek	& ~&	
	^ ~^	
	| ~|	
logikai műveletek	&& ||	
feltételes műveletek	? :	Legalacsonyabb prioritás

9-2 táblázat: A műveletek hierarchiája.

10. Viselkedési szintű *HDL* leírás

A digitális áramkörök tervezésére kialakított legmagasabb szintű hardver nyelvi leírás a viselkedési szintű leírás. A magas szintű elvonatkoztatás lehetővé teszi, hogy a tervező ne a részletkérdésekkel foglalkozzon, hanem a tervezendő áramkör viselkedésével. Ezek a leírások nagyon hasonlítanak a *C* szoftvernyelvben kialakított programokra, de mindig tudatában kell lennünk, hogy a leírás célja egy áramkör kialakítása. A *HDL* leírás alapján a szintézist végző szoftver kifejleszt egy áramkört, amelyet megfelelő *PLD* programozásával valósíthatunk meg.

A szimulációt leíró *Verilog HDL* modulok általában olyan viselkedési szintű leírásokat tartalmaznak, amelyek nem kerülnek hardveres megvalósításra, céljuk csupán más modulok vizsgálata.

10.1. Szerkesztett eljárások

A viselkedési szintű leírások két fajta szerkesztett eljárás *(structured procedure)* keretében írhatók: ezeket a megfelelő kulcsszavak alapján ***initial*** és az ***always*** eljárásoknak nevezzük. A viselkedési szintű utasítások csak ezen eljárásokon belül jelenhetnek meg.

Egy *Verilog HDL* modul több szerkesztett eljárást is tartalmazhat. Nem kezdhető új eljárás, amíg az előzőnek az írását be nem fejeztük. Ezek az eljárások a modul szimulációja során mind *t=0* időpontban indulnak és párhuzamosan (konkurrens módon) futnak, ellentétben a szoftver nyelvi leírásokkal, ahol a program végzése lépésről-lépésre történik. Ez a tulajdonság azzal kapcsolatos, hogy a *HDL* leírásnak megfelelő fizikai áramkörben is az áramkör egyes részei egyidőben, egymással párhuzamosan működnek.

10.1.1. Az *initial* eljárás

Az ***initial*** eljárás az ***initial*** kulcsszóval kezdődik, ezt egy egy vagy több hozzárendelést tartalmazó ***initial*** blokk követi. Az ***initial*** blokkban található műveletek végzése *t=0*-ban kezdődik és pontosan egyszer kerülnek végrehajtásra. Ha több ***initial*** eljárást tartalmaz egy modul, mindegyikük *t=0*-ban indul és egymástól függetlenül kerülnek végrehajtásra és fejeződnek be.

Áramkörök tervezésénél az ***initial*** eljárások elsősorban egyszeri beállításokra használhatók. Az esetek többségében az ***initial*** eljárásokat nem áramkörök leírására, hanem szimulációs modulokban alkalmazzuk a megfelelő gerjesztő jelek megadására.

Ha egy ***initial*** eljárás csak egy hozzárendelést tartalmaz, azt közvetlenül az ***initial*** kulcsszó után írjuk, s ezzel az eljárás leírását be is fejeztük. Ha több hozzárendelést tartalmazó ***inital*** blokk követi a kulcsszót, a blokk elé kell írni a ***begin*** kulcsszót, a végére pedig az ***end*** kulcsszót. A 10-1 példában különböző initial eljárásokat láthatunk.

10-1 példa: Több különböző initial eljárást tartalmazó szimulációs modul.

```
module Stimulus;
reg a, b, m;
initial
        m=1'b0;          // Egy hozzárendelés esetén nincs szükség a begin és end
                         // kulcsszavakra.
initial
begin
        #5 a=1'b1;       // Két hozzárendelés esetén már szükséges a begin és end
        #25 b=1'b0;      // kulcsszavak használata. A késésekkel később foglalkozunk.
end
initial
        #50 $finish;     // A szimuláció végét definiáló initial eljárás (egy utasítást
                         // tartalmaz).
endmodule
```

A szimuláció során mind a három ***initial*** eljárás végzése *t=0*-ban kerül sorra. Az egyes hozzárendelések elvégzése a leírásban megadott módon késleltethető. A hozzárendelések tényleges elvégzési ideje a 10-1 táblázatban látható.

idő	hozzárendelés
0	m=1'b0
5	a=1'b1
30	b=1'b0
50	$finish

10-1 táblázat: A 10-1 példában szereplő utasítások elvégzésének tényleges ideje.

10.1.2. Az *always* eljárás

Az ***always*** eljárás leírása az ***always*** kulcsszóval kezdődik, ezt egy egy vagy több hozzárendelést tartalmazó ***always*** blokk követi. Több hozzárendelés esetén itt is szükségesek a ***begin*** és az ***end*** kulcsszavak. Az always blokk hozzárendelései ciklikusan ismétlődnek a szimuláció egész ideje alatt (szimulációs modul esetén), illetve a tervezendő áramkör teljes működési ideje alatt. A 10-2 példában bizonyos áramkör szimulációjához szükséges órajel definiálását végző (***always*** eljárást tartalmazó) modult adtunk meg.

10-2 példa: Órajelet definiáló, always eljárást tartalmazó modul.

```
module clock_gen;
reg clock;
initial
        clock=1'b0;             // Induljon az órajel alacsony logikai szintről.
always
        #10 clock=~clock;       // Minden 10 időegység után  invertáljuk clock logikai
                                // értékét.
initial
        #1000 $finish;          // Az órajel impulzusok végét definiáló initial eljárás.
endmodule
```

10.2. Az eljárásokban szereplő hozzárendelések

Az ***initial*** és ***always*** eljárásokban megadott hozzárendelések új értékeket rendelnek az egyes ***reg***, ***integer*** és ***real*** típusú adathordozókhoz. Az így kapott érték nem változik, amíg újabb hozzárendelést nem végzünk. Ez lényeges eltérés az adatfolyam szintű hozzárendelésekhez képest: ott a *net* típusú adathordozókhoz a hozzárendelés folyamatosan működött. Amint változás történt a bemeneti értékekben, azonnal kiértékelődött és hozzárendelődött az új érték a kimeneti adathordozóhoz.

Az itt tárgyalt hozzárendelések a hozzárendelő jelből és a tőle jobbra elhelyezkedő kifejezésből, valamint a tőle balra elhelyezkedő adathordozóból illetve annak egy bitjéből vagy bitcsoportjából, esetleg összecsatolt adathordozókból áll.

Az eljárásokban szereplő hozzárendelések két típusát különböztetjük meg: léteznek blokkoló- és nem blokkoló hozzárendelések. A közöttük levő különbségeket a következő pontokban tárgyaljuk.

10.2.1. Blokkoló hozzárendelések

A blokkoló hozzárendelések jele az egyenlőségjel (=). Ezek a hozzárendelések abban a sorrendben végződnek el, amilyen sorrendben fel vannak tüntetve az ***initial*** illetve ***always*** blokkban. A soron levő hozzárendelés tehát blokkolja az utána következőket. Két külön eljárásban (***initial*** vagy ***always***) szereplő hozzárendelések nem blokkolják egymást, egymástól függetlenül kerülnek elvégzésre. A 10-3 példában egy blokkoló hozzárendeléseket tartalmazó modul részletet láthatunk,

a 10-2 táblázatban pedig összefoglaltuk, hogy az egyes hozzárendelések milyen sorrendben kerülnek sorra és melyik időpontban végződnek el.

10-3 példa: Blokkoló hozzárendeléseket tartalmazó modul részlet.

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial
begin
        x=1'b0; y=1'b1; x=1'b1;
        count=0;
        reg_a=16'b0; reg_b=reg_a;
        #15 reg_a[2]=1'b1;
        #10 reg_b[15:13]={x,y,z};
        count=count+1;
end
```

10-2 táblázat: A 10-3 példában szereplő hozzárendelések sorrakerülésének sorrendje és elvégzésük tényleges ideje.

A sorrakerülés sorrendje	Az elvégzés időpontja	Hozzárendelés
1	0	x=1'b0
2	0	y=1'b1
3	0	z=1'b1
4	0	count=0
5	0	reg_a=1'b0
6	0	reg_b=reg_a
7	15	reg_a[2]=1'b1
8	25	reg_b[15:13]={x,y,z}
9	25	count=count+1

A hozzárendelésekben megtörténhet, hogy az egyenlőségjel bal oldalán levő adathordozó bitszáma nem egyezik meg a jobb oldalon levő kifejezés kiértékelésekor kapott érték bitjeinek számával. Ha az adathordozó bitszáma a kisebb, a megfelelő számú kisebb helyi értékű bitek rendelődnek az adathordozóhoz, a felesleges nagyobb helyi értékű bitek elvesznek, ellenkező esetben a nem definiált bitek nullákkal töltődnek fel.

10.2.2. Nem blokkoló hozzárendelések

A nem blokkoló hozzárendeléseknél a <= jelet használjuk. Ez egyben a viszonyító művelet jele is, de a *Verilog HDL* leírást értelmező szoftver mindig meg tudja állapítani, hogy viszonyításról, vagy hozzárendelésről van-e szó. Ez esetben a soron levő hozzárendelés nem fagyasztja be a további hozzárendelések végzését. Megismételjük a 10-3 példát, de három hozzárendelést nem blokkoló típusúra változtatunk és figyeljük a következményeket (10-4 példa). A 10-3 táblázatban összefoglaltuk, hogy a 10-4 példa egyes hozzárendelései milyen sorrendben kerülnek sorra és melyik időpontban végződnek el.

10-4 példa: Blokkoló és nem blokkoló hozzárendeléseket tartalmazó modul részlet.

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial
begin
        x=1'b0; y=1'b; x=1'b1;
        count=0;
        reg_a=16'b0; reg_b=reg_a;
        reg_a[2]<= #15 1'b1;
        reg_b[15:13]<= #10 {x,y,z};
        count<=count+1;
```

```
end
```

A sorrakerülés sorrendje	Az elvégzés időpontja	Hozzárendelés
1	0	x=1'b0
2	0	y=1'b1
3	0	z=1'b1
4	0	count=0
5	0	reg_a=1'b0
6	0	reg_b=reg_a
7	15	reg_a[2]<=1'b1
7	10	reg_b[15:13]<={x,y,z}
7	0	count<=count+1

10-3 táblázat: A 10-4 példában szereplő hozzárendelések sorrakerülésének sorrendje és elvégzésük tényleges ideje.

Elmondhatjuk tehát, hogy az első hat hozzárendelés mind *t=0*-ban végződik el, méghozzá a felírás sorrendjében (blokkoló hozzárendelések). Ezt követően, de még mindig *t=0*-ban a maradék három hozzárendelés egyszerre kerül sorra. Ha nincs semmi késés bejelölve az illető hozzárendelés azonnal megtörténik (*t=0*-ban). Ha van késés, akkor a hozzárendelés bal oldalán levő adathordozó a késés letelte után veszi fel a hozzárendelés jobb oldalán szereplő kifejezés *t=0*-ban kiszámolt értékét.

A nem blokkoló hozzárendelésekkel olyan digitális áramköröket szoktunk leírni, amelyekben konkurrens (egyidejű) adatátvitel történik egy közös jel (esemény) hatására. A 10-5 példában három hozzárendelés történik az órajel felfutó élénél. A hozzárendelések időzítését (órajellel és másként) a 10.3 szakaszban fogjuk részletesebben tárgyalni.

10-5 példa: Konkurrens adatátvitel nem blokkoló hozzárendelésekkel.

```
always @ (posedge clock)
begin
        reg1<=#1 in1;
        reg2<=@(negedge clock) in2^in3;
        reg3<=#1 reg1;
end
```

A fenti példában az órajel felfutó élénél a következő eseményekre kerül sor:

- A hozzárendelések jobb oldalain szereplő változók (*in1*, *in2*, *in3*, *reg1*) értékeit megállapítódnak, a megfelelő kifejezések azonnal kiértékelődnek és egy ideiglenes memóriában feljegyződnek.
- A tényleges hozzárendelés akkor történik meg, amikor annak eljön az ideje. Az első és a harmadik hozzárendelés egy időegység késést követően játszódik le (a feltüntetett késés miatt), míg a második hozzárendelés esetében az órajel legközelebbi lefutó élénél történik a hozzárendelés.
- Lényegtelen, hogy a regiszterekbe való beírások (a tényleges hozzárendelések) milyen sorrendben történnek, mert az ideiglenes memória tartalma nem függ a sorrendtől. Pl. a harmadik hozzárendelésben a *reg3* a *reg1* „régi" értékét kapja, annak ellenére, hogy az első hozzárendelésben *reg1* új értéket kapott.

Hogy még jobban megismerkedjünk a nem blokkoló hozzárendelésekkel megvalósított adatátvitel szabályaival, tekintsük először a blokkoló hozzárendeléseket tartalmazó 10-6 példát.

10-6 példa: Versenyfutási jelenséget eredményező HDL leírás blokkoló hozzárendelésekkel.

```
always @ (posedge clock)
a=b;
always @ (posedge clock)
b=a;
```

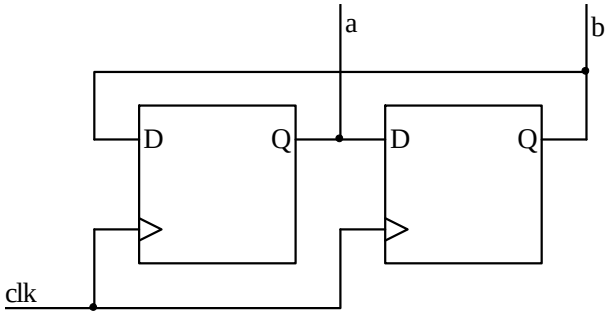
A két ***always*** eljárás egyszerre indul (az órajel minden felfutó élénél). Az eljárásokban szereplő blokkoló hozzárendelések egyikének sincs prioritása (mivel külön eljárásban szerepelnek), ezért az elvégzés sorrendje véletlenszerű lesz. Miután az egyik hozzárendelés megtörténik, mindkét regiszter ugyanazt az értéket fogja tartalmazni, a hátramaradt hozzárendelésnek már nem lesz hatása.

A versenyfutási jelenség megakadályozható nem blokkoló hozzárendelések alkalmazásával (10-7 példa). Itt a hozzárendelések jobb oldalain szereplő adathordozók kezdeti értékei az órajel felfutó élénél feljegyződnek. Ez után végződnek el a hozzárendelések, amelyek során a korábbi értékek kicserélődnek, a hozzárendelések sorrendjétől függetlenül.

10-7 példa: A versenyfutási jelenség kiküszöbölése nem blokkoló hozzárendelésekkel.

```
always @ (posedge clock)
a<=b;
always @ (posedge clock)
b<=a;
```

Ez a *HDL* leírás alapján a 10-1 ábrán bemutatott áramkör szintetizálódik. Az áramkörben az órajel minden felfutó élénél kicserélődik a *flip-flop*-ok tartalma. Az élvezérelt áramkörök alaptulajdonsága, hogy az órajel megfelelő élénél mintavételezik a bemenetet és ezt követően, a mintavételezett értékek alapján elvégzik a megfelelő beírást. Ha ezt követően rövid időn belül a bemenetek megváltoznak, annak nem lesz semmilyen következménye a beírásra.



10-1 ábra: A 10-7 példa HDL leírása alapján szintetizált áramkör.

A versenyfutási jelenség kiküszöbölhető blokkoló hozzárendelések alkalmazásával is, a 10-8 példában megadott módon. A *temp_a* és *temp_b* (ideiglenes) adathordozók bevezetésének köszönhetően feljegyződnek a kiindulási értékek és szabályosan megtörténik a csere.

10-8 példa: A versenyfutási jelenség kiküszöbölése ideiglenes tárolással.

```
always @ (posedge clock)
begin
        temp_a=a;
        temp_b=b;
        a=temp_b;
        b=temp_a;
end
```

10.3. A hozzárendelések időzítése a viselkedési szintű leírásokban

Ha a viselkedési szintű leírásokban nem szerepel semmilyen időzítés, a leírás szimulációja során nem történik előrehaladás az időskálán. A valós áramkörök tervezésénél a tényleges viselkedés pontosabb modellezése megköveteli az időzítések alkalmazását. A szerkesztett eljárásokban (***initial*** és ***always***) az időzítések megadásának három módja létezik:

- késések definiálása;
- élvezérlés;
- szintvezérlés.

10.3.1. Időzítés késések definiálásával

Az eljárásbeli késések definiálásával időt határozunk meg, aminek el kell telnie a hozzárendelés sorrakerülése és végrehajtása között. A késéseket a # jellel vezetjük be, a jelet követi a késés értéke számérték-, azonosító- vagy *min/typ/max* kifejezés formájában.

A késések írásának szabályos módja, hogy a késést a hozzárendelés elé írjuk. Ezt a lehetőséget a 10-9 példában láthatjuk.

10-9 példa: A késések írásának szabályos módja.

```
initial
begin
        #10 y=1'b1;
        #latency z=1'b0;
        #(4,5,6) q=1'b0;
end
```

Az első hozzárendelés a *t=0*-tól számított tíz időegység múlva kerül elvégzésre. Ha a jobb oldalon valamilyen kifejezés szerepel (nem állandó, mint a példában), annak a kiértékelése is ebben a pillanatban történik. Tekintettel a blokkoló hozzárendelések tulajdonságára, csak ez után kerül sorra a másik hozzárendelés elemzése. Mivel ott egy azonosítóval adtuk meg a késést, az azonosító értékének megfelelő további időnek kell eltelni a hozzárendelés elvégzése előtt. A második hozzárendelés befejezése után kerül sorra a harmadik hozzárendelés, ahol a késést *min/typ/max* kifejezéssel adtuk meg.

A késések írásának másik módja a hozzárendelésen belüli késés megadás. Itt a megfelelő késési időt a hozzárendelési jel (= vagy <=) jobb oldalára írjuk. Ebben az esetben a hozzárendelésben szereplő kifejezés kiértékelése a hozzárendelés sorrakerülésének pillanatában történik (nem a késés után, mint a szabályos késés-megadásnál), a hozzárendelés viszont késik. A 10-10 példában az első ***initial*** eljárásban a késést a hozzárendelésen belül adtuk meg, így az összeadás *t=0*-ban történik, de maga a hozzárendelés öt időegységet késik. A második ***initial*** eljárás azt mutatja, hogyan lehet elérni ugyanezt az eredményt egy ideiglenes adathordozó (*temp_xz*) bevezetésével és a késés szabályos megadásával.

10-10 példa: A késések írása a hozzárendelésen belül.

```
initial
y= #5 x+z;

initial
begin
        temp_xz=x+z
        #5 y=temp_xz
end
```

Mint már korábban elmondottuk, az összes ***initial*** és ***always*** eljárások végrehajtása *t=0*-ban indul. Nem lehet tudni, hogy az egyes eljárásokon belüli, *t=0* időben sorrakerülő

hozzárendelések milyen sorrendben végződnek el, így versenyfutási jelenségek léphetnek fel. Nulla értékű késés megadásával sorrendet lehet felállítani.

A 10-11 példában minden hozzárendelés *t=0*-ban végződik el, de az első eljárás hozzárendelései megelőzik a másodikét, tehát a végeredmény *x=y=1* lesz. Általában nem tanácsos ugyanabban a pillanatban egy adathordozó értékét két hozzárendeléssel megváltoztatni a bizonytalan eredmény miatt, de a nulla értékű késéssel a helyzet esetleg pontosítható.

10-11 példa: Nulla értékű késés alkalmazása sorrend felállítása céljából.

```
initial
begin
        x=0;
        y=0;
end

initial
begin
        #0 x=1;
        #0 y=1;
end
```

10.3.2. Időzítés élvezérléssel és szintvezérléssel

A mai digitális rendszerek többsége szinkron hálózat, ezen belül is az élvezérlés a jellemző. Az események az órajel megfelelő élét követően játszódnak le. Az élvezérlést és a szintvezérlést négy módon jelölhetjük a viselkedési szintű hardver nyelvi leírásokban:

- szabályos élvezérlési módszer,
- nevezett eseményre való várakozás,
- több jel egyikének élére való várakozás,
- megfelelő logikai szintre való várakozás.

A szabályos módszerrel élvezérlést írunk le a **@** és a ***posedge*** vagy a ***negedge*** kulcsszavak alkalmazásával. A 10-12 példában az első ***always*** eljárásban a hozzárendelést az órajel mindkét élénél el kell végezni. A második eljárásban a hozzárendelés az órajel felfutó élénél történik, a harmadikban a lefutó élnél. A negyedik eljárásban memorizáljuk *d* értékét abban a pillanatban, amikor a hozzárendelés sorra kerül, de a memorizált érték hozzárendelése a *q* adathordozóhoz az órajel felfutó élénél történik.

10-12 példa: Időzítés szabályos módja (élvezérlés definiálása).

```
always @ (clock)
        q=d;
always @ (posedge clock)
        q=d;
always @ (negedge clock)
        q=d;
always
        q=@(posedge clock) d;
```

Szeretnénk felhívni a figyelmet, hogy a hardver nyelvi leírásokba nincsennek semmilyen kész órajel források, sem kitüntetett elnevezések az órajelekre (a fenti példában a *clock* nem kulcsszó). A leírás szimulációjához szükséges órajelet a tervezőnek kell kialakítania egy modulon belül. Ez a hozzáállás tapasztalható a *HDL* leírás alapján tervezett áramkör hardveres megvalósításánál is: a *PLD* áramkörök nem tartalmaznak belső órajel generátort, az órajelet külön áramkörrel kell biztosítani.

Ha az időzítést nevezett eseményre való várakozással kívánjuk megoldani, először deklarálnunk kell egy eseményt. A kijelölt hozzárendelés akkor végződik el, amikor bekövetkezik az esemény. A deklarációt az ***event*** kulcsszóval eszközöljük, az esemény beindítását a **->** jellel végezzük, az esemény figyelembe vétele (a megfelelő hozzárendelés elvégzése) viszont a **@** jelnél történik.

A 10-13 példában az első sorral végeztük el az esemény deklarálását. Az ezt követő ***always*** eljárás definiálja, hogy mikor jelentkezik a nevezett esemény (hogy mi is valójában az esemény, amire várakozunk). Az itt szereplő ***if*** kulcsszó használatára a következő (10.4) szakaszban térünk ki. A második ***always*** eljárásban szerepel a hozzárendelés, amelyet el kell végezni, ha megtörténik a nevezett esemény.

10-13 példa: Időzítés nevezett eseményre való várakozással.

```
event received_data;

always @(posedge clock)
begin
        if (last_data_packet)
                ->received_data;
end

always @ (received_data)
data_buf={data_pkt[0], data_pkt[1], data_pkt[2], data_pkt[3]};
```

A szabályos módszerrel definiált élvezérlés kiterjeszthető úgy, hogy több jel közül az egyiknek a változását kell figyelembe venni a hozzárendelés elindításához. A leírás hasonló, csak az egyes jeleket az ***or*** kulcsszóval össze kell fűzni (10-14 példa).

10-14 példa: Időzítés több jel egyikének az élére való várakozással.

```
always @(reset or clock or d)
begin
        if (reset)
                q=1'b0;
        else   if(clock)
                q=d;
end
```

Félreértésre adhat okot, hogy a fenti példában ugyan élvezérlést alkalmazunk a hozzárendelés beindításához, mégis a megadott leírás alapján egy szintvezérelt *D latch* áramkör (3.1.1 pont) fog szintetizálódni. Az ***always*** eljárás a *reset*, *clock* és *d* jelek felfutó és lefutó élénél indul, mert a szintvezérelt áramköröknél akkor várható változás a működésben, ha ezen jelek valamelyikénél megváltozott a logikai szint (ez pedig az éleknél történik). Ha tehát változás állt be a három bemenő jel egyikében, az új logikai szintektől függően elvégezzük a *latch* aszinkron reszetjét (*q=1'b0*), transzparenssé tesszük a *latch*-et (*q=d*), vagy a *clock* jel alacsony logikai szintje esetén reteszelt (rögzített) állapotban hagyjuk az áramkört (marad a korábbi kimenet).

Az élvezérlés helyett a *HDL* leírásokban alkalmazhatunk szintvezérlést is. Ezt nem a **@** jellel, hanem a ***wait*** kulcsszóval érjük el. A 10-15 példában a *count* adat inkrementálása csak akkor történik meg minden húsz időegység után, ha a *count_enable* adathordozó magas logikai szintjével ezt engedélyeztük.

10-15 példa: Időzítés logikai szintre való várakozással (szintvezérlés).

```
always
        wait (count_enable) #20 count=count+1;
```

10.4. Feltételhez kötött hozzárendelések

Az eljárásokban az ***if*** és az ***else*** kucsszavakkal definiálható, hogy adott hozzárendelést milyen feltétel mellett kell elvégezni és mikor kell figyelmen kívül hagyni. A leírás módjában három esetet különböztetünk meg.

Az első esetben (10-16 példa) csak az ***if*** kulcsszót használjuk. Ha a *clock* jel alacsony logikai szinten van, megtörténik a hozzárendelés, egyébként nem.

10-16 példa: Igaz feltételhez kötött hozzárendelés.

```
if (!clock)    buffer=data;
```

Ha igaz és hamis feltétel esetén is el kell végezni bizonyos műveletet, akkor a nyelvi szerkezetben az ***if*** és az ***else*** kulcsszavakat használjuk (10-17 példa). Ha a példában a megadott feltétel teljesül, a ***begin*** és ***end*** kulcsszavak közé zárt két hozzárendelést kell elvégezni, ellenkező esetben a ***$display*** utasítás kerül elvégzésre. Ez az utasítás nem áramkört ír le, hanem a leírást szimuláló szoftvert utasítja.

10-17 példa: Igaz és hamis feltételhez kötött hozzárendelések.

```
if (number_queued<MAX_Q_DEPTH)
begin
        data_queued=data;
        number_queued= number_queued+1;
end
else
        $display ("Queue full, Try again");
```

A harmadik eset az, amikor a feltételnek kettőnél több értéke is lehet és minden esetben más hozzárendelést kell elvégezni. A 10-18 példába az *alu_control* paraméter három értékénél (*0, 1* és *2*) megfelelő hozzárendelések kerülnek sorra, ha viszont egyik feltétel sem teljesül, akkor a ***$display*** utasítás érvényesül. A bemutatott nyelvi szerkezetben az ***if – else if – else*** láncolatot használjuk.

10-18 példa: Kettőnél több feltételhez kötött hozzárendelések.

```
if (alu_control==0)
        y=x+z;
else if (alu_control==1)
        y=x-z;
else if (alu_control==2)
        y=x*z;
else
        $diplay ("Invalid alu control signal");
```

10.5. Többfelé történő elágazások

Az előző szakaszban tárgyalt többszörös ***if-else*** szerkezetek írása nagyobb számú feltétel esetén körülményes és nehezen áttekinthető. Ilyen esetekben a ***case*** kulcsszóval kialakítható szerkezet használata tanácsos.

10.5.1. A *case* szerkezetek

A szerkezet kialakításához a ***case, endcase*** és ***default*** kulcsszavakat használjuk. A ***default*** kulcsszó használata nem kötelező, ha mégis alkalmazzuk, egy szerkezetben csak egyszer jelentkezhet. A 10-19 példában az előző példa ***if-else*** szerkezetét ***case*** szerkezetté alakítottuk, így egyszerűbb *HDL* leírást kaptunk.

10-19 példa: A 10-18 példában szereplő HDL leírás case szerkezetre átalakítva.

```
reg [1:0] alu_control;

case (alu_control)
        2'd0: y=x+z;
        2'd1: y=x-z;
        2'd2: y=x*z;
        default: $diplay ("Invalid alu control signal");
endcase
```

A multiplexer áramkörök (2.5 szakasz) *HDL* leírása éppen a ***case*** szerkezettel a legcélszerűbb. A 10-20 példa egy 4/1-es multiplexer leírását mutatja (megadtuk a teljes modult).

10-20 példa: 4/1-es multiplexer HDL leírása case szerkezettel.

```
module mux4_to_1(out, i0, i1, i2, i3, s1, s0);
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0})
        2'd0: out=i0;
        2'd1: out=i1;
        2'd2: out=i2;
        2'd3: out=i3;
        default: $diplay ("Invalid control signals");
endcase
endmodule
```

A példában az ***always*** eljárás mindig aktivizálódik, amikor legalább egy bemeneti adathordozó logikai szintje megváltozik A 10-14 példához hasonlóan itt is szintvezérlésről van szó, de az eljárás akkor indul, amikor a logikai szint megváltozik (tehát az élnél). Az esetek kódjait a két egybites választóbemenet (*s1*, *s0*) összefűzésével kaptuk. A ***default*** kulcsszóval kezdődő sor kihagyható, ez egyébként is nem az áramkör leírására vonatkozik. Annak ellenére, hogy a kimeneti adathordozót regiszter típusúra deklaráltuk, a megadott *HDL* leírás alapján kombinációs hálózat (multiplexer) fog szintetizálódni.

A ***case*** szerkezet egy-egy esetében egy hozzárendelés helyett állhat több is. Ilyenkor a hozzárendeléseket a ***begin*** és az ***end*** kulcsszavak közé kell zárni. Az esetek kódjainak egyes bitjei felvehetnek *x* és *z* értékeket is, ezzel az áramkör viselkedése precízebben leírható.

10.5.2. A *casex* és a *casez* kulcsszavak használata

Elsősorban a leírások rövidítése céljából használhatók a ***casex*** és ***casez*** kulcsszavak. A ***casez*** kulcsszóval használatakor az eset kódjában a *z*-vel jelölt helyeken állhat bármi, a hozzárendelésre ez nem lesz kihatással. A ***casex*** kulcsszó esetén sem az *x* sem a *z* értékeket nem vesszük figyelembe a hozzárendelés végrehajtásakor. A 10-21 példában egy olyan kódert írtunk le, amely a kimeneti kódot az szerint alakítja ki, hogy van-e az adott helyen logikai egyes, a többi bit értéke lényegtelen.

10-21 példa: A casex kulcsszó használata különleges kóder kialakítására.

```
reg[3:0] encoding;
integer state;
casex (encoding)
        4'b1xxx: state=3;
        4'bx1xx: state=2;
        4'bxx1x: state=1;
        4'bxxx1: state=0;
```

```
        default:  state=0;
endcase
```

10.6. Hurkok a viselkedési szintű leírásokban

A viselkedési szintű leírásokban az ***always*** eljárások a hozzárendelések ismételt végzését okozzák a szimuláció ideje alatt, illetve, megvalósított hardvernél, a működés ideje alatt. A *Verilog HDL*-ben definiálható hurkok segítségével a hozzárendelések ismétlése feltételhez köthető. Így az ismétlések száma vagy időpontja beállítható.

Négy kulcsszó létezik a hurkok definiálására: ezek a ***while***, a ***for***, a ***repeat*** és a ***forever***. A hurkok csak a viselkedési szintű leírásokon belül, tehát csakis ***initial*** vagy ***always*** blokkokban jelentkezhetnek.

10.6.1. A *while* típusú hurok

Ezt a hurkot a ***while*** kulcsszóval és a mellette zárójelben álló kifejezéssel vezetjük be. A hurokban szereplő hozzárendelések addig ismétlődnek, amíg a zárójelben levő kifejezés igaz logikai értéket ad. A hozzárendelések egyszer sem történnek meg, ha már kezdettől fogva a kifejezés értéke hamis. Ha a hurkon belül több hozzárendelést kell elvégezni, azokat ***begin*** és ***end*** kulcsszó közé kell zárni. A 10-22 példában megadott modul részlet egy hét bites számlálót definiál ***while*** hurok segítségével.

10-22 példa: A while típusú hurok használata számláló kialakítására.

```
integer count;
initial
begin
        count=0;
        while (count < 127)
        begin
                $display (“count=%d”, count);
                count=count+1;
        end
end
```

A ***$display*** utasítás melletti zárójel azt definiálja, hogy mit kell kiírni a tervezett áramkör szimulációját végző számítógép kijelzőjére. Valahányszor a szimulátor átmegy a hurkon, a kijelzőn megjelenik a *count*= szöveg és a *count* adat pillanatnyi értéke decimális formában.

10.6.2. A *for* típusú hurok

Ezt a hurkot a ***for*** kulcsszóval és a mellette zárójelben álló kifejezéssel vezetjük be. A kifejezés három elemet tartalmaz:

- egy kezdeti feltételt,
- a befejezési feltétel ellenőrzését,
- egy hozzárendelést, amellyel a hurok befejezését ellenőrző értéket módosítjuk.

Az előző példában megadott számláló a ***for*** hurok segítségével egyszerűbben leírható (10-23 példa). El kell azonban mondani, hogy a ***while*** hurok általánosabb, szélesebb körben használható.

10-23 példa: A for típusú hurok használata számláló kialakítására.

```
integer count;
initial
        for (count=0; count<128; count=count+1)
                $display (“count=%d”, count);
```

10.6.3. A *repeat* típusú hurok

Ezt a hurkot a ***repeat*** kulcsszóval és a mellette zárójelben álló számértékkel vezetjük be. A számérték helyett szerepelhet valamilyen azonosító vagy kifejezés is, de ez a hurokba lépés előtt kiértékelődik és a hurok időtartama alatt állandónak tekintődik. A ***repeat*** hurkot tehát akkor alkamazzuk, ha a hurokban megjelenő hozzárendeléseket ismert számszor kell elvégezni. A 10-24 példában a korábbi számláló leírását láthatjuk ***repeat*** hurokkal.

10-24 példa: A repeat típusú hurok használata számláló kialakítására.

```
integer count;
initial
begin
        count=0;
        repeat (128)
        begin
                $display (“count=%d”, count);
                count=count+1;
        end
end
```

10.6.4. A *forever* típusú hurok

Ezt a hurkot a ***forever*** kulcsszóval vezetjük be. A leírás nem tartalmaz feltételt, amely a hurok befejezését okozná, így a hurok elvileg végtelen ideig tart (a szimuláció végéig illetve hardveres megvalósításnál mindaddig, amíg az áramkör táplálást kap). A 10-25 példa órajelet állít elő ***forever*** hurok segítségével.

10-25 példa: A forever típusú hurok használata órajel kialakítására.

```
initial
begin
        clock=1’b0;
        forever #10 clock=~clock;
end
```

10.7. Soros és párhuzamos blokkok

A blokkokokban hozzárendeléseket csoportosítunk, amelyek egy egységet alkotnak. Ezideig már sokszor alkalmaztuk a ***begin*** és ***end*** kulcsszavakat, amelyekkel soros blokkokat hoztunk létre. A soros blokkokban a hozzárendelések egymás után, a felírás sorrendjében kerülnek elvégzésre. A soros blokkok mellett léteznek párhuzamos blokkok, az ezekben szereplő hozzárendelések egyidőben kerülnek sorra (az elvégzés időpontja különbözhet, ha késések is szerepelnek a leírásban.

10.7.1. Soros blokkok

A soros blokkokat a ***begin*** kulcsszóval vezetjük be és az ***end*** kulcsszóval zárjuk le. A két kulcsszó közötti hozzárendelések a felírás sorrendjében végződnek el. Nem indul új hozzárendelés, amíg az előző be nem fejeződött. Az estleges késések relatívak: a megelőző hozzárendelés végrehajtásától számítjuk a soron levő hozzárendelésre vonatkozó késést.

Az eddigi példákban már számos soros blokkot alkalmaztunk. A 10-26 példa egy késés nélküli soros blokkot tartalmaz.. A négy hozzárendelés a felírás sorrendjében, de mind *t=0*-ban végződik el. Így *z* és *w* bitjeit az előzőleg *x*-hez és *y*-hoz rendelt értékekből alakítjuk ki.

10-26 példa: Soros blokk késések nélkül.

```
reg x,y;
reg [1:0] z, w;
```

```
initial
begin
        x=1'b0;
        y=1'b1;
        z={x,y};
        w={y,x};
end
```

A 10-27 példában a soros blokk hozzárendelései (az első kivételével) késéseket tartalmaznak. Mivel a késések összeadódnak, az első hozzárendelés *t=0*-ban végződik el, a második *t=5*-ben, a harmadik *t=15*-ben, a negyedik pedig *t=35*-ben.

10-26 példa: Soros blokk késésekkel.

```
reg x,y;
reg [1:0] z, w;
initial
begin
        x=1'b0;
        #5 y=1'b1;
        #10 z={x,y};
        #20 w={y,x};
end
```

10.7.2. Párhuzamos blokkok

A párhuzamos blokk hozzárendeléseit a ***fork*** kulcsszó előzi meg és a ***join*** kulcsszó zárja le. A korábban a nem blokkoló hozzárendeléseknél említett módon (10.2.2 pont) a hozzárendelések egyidőben kerülnek sorra, de itt nem memorizálódnak az adatok a blokkba való belépés pillanatában.

Az elvégzési időket késések megadásával, eseményre való várakozással, élvezérléssel és szintvezérléssel (10.3.2 pont) befolyásolhatjuk. A késéseket a blokkba való belépés pillanatától számítjuk. A hozzárendelések felírásának sorrendje lényegtelen.

Ha a 10-27 példában megismételtük a korábbi példát, de a soros blokk helyett párhuzamos blokkot alkalmazunk. Az adathordozókhoz végeredményben ugyanazokat az értékeket rendeljük, de a hozzárendelések elvégzésének ideje módosul. A feltüntetett késések abszolút értékek.

10-27 példa: Párhuzamos blokk késésekkel.

```
reg x,y;
reg [1:0] z, w;
initial
fork
        x=1'b0;
        #5 y=1'b1;
        #10 z={x,y};
        #20 w={y,x};
join
```

A párhuzamos blokkok esetében versenyfutási jelenség léphet fel, ha két vagy több hozzárendeléssel próbáljuk módosítani az egy adathordozóhoz rendelt adatot. Ugyanígy gondok jelentkezhetnek, ha ugyanabban a blokkban definiálunk és használunk egy adatot. A 10-28 példában a *z* és *w* értékei bizonytalanok a blokk elhagyásának pillanatában, mert nem tudni, hogy idejében definiálva lettek-e *x* és *y* értékei.

10-28 példa: Párhuzamos blokk késések nélkül (versenyfutási jelenség lép fel).

```
reg x,y;
reg [1:0] z, w;
```

```
initial
fork
        x=1'b0;
        y=1'b1;
        z={x,y};
        w={y,x};
join
```

10.7.3. Kombinált blokkok

A soros és a párhuzamos blokkok kombinálhatók (10-29 példa). A bemutatott esetben a külső (fő) blokk soros, a belső párhuzamos. Értelemszerűen először a soros blokk első hozzárendelése hajtódik végre, ezt követi a párhuzamos blokk két hozzárendelése, majd a soros blokk utolsó hozzárendelése kerül sorra. A párhuzamos blokk két hozzárendelése egyidőben kerül sorra de végrehajtásuk a *t=5* és *t=10* időpontokban történik a bejelölt késések miatt.

10-29 példa: Soros és párhuzamos blokk kombinálása.

```
initial
begin
        x=1'b0;
        fork
                #5 y=1'b1;
                #10 z={x,y};
        join
        #20 w={y,x};
end
```

10.7.4. Nevezett blokkok

A blokkoknak nevet adhatunk. Az így kapott nevezett blokkoknál:

- helyi adathordozókat deklarálhatunk,
- az egyes adathordozókat hierarchikus neveken keresztül elérhetjük,
- a végrehajtást megszakíthatjuk.

A 10-30 példában az *i* adathordozó két blokkban is szerepel, de mivel ezek nevezett blokkok, nem történik keveredés. Az sem gond, hogy az adathordozók típusa nem ugyanaz.

10-30 példa: Nevezett blokkok alkalmazása.

```
module top;

initial
begin: block1
integer i;
.........
end

initial
fork: block2
reg i;
.........
join
endmodule
```

10.7.5. A nevezett blokkok megszakítása

A ***disable*** utasítással a nevezett blokkok végrehajtása megszakítható. Így bizonyos utasítások átugorhatók vagy hurokból ki lehet lépni. A ***while*** hurokból való kilépés módját láthatjuk a 10-31 példában. Ha a *flag* vektor soron levő bitje egyes, a *block1* nevű blokk végrehajtása megszakad. A disable utasítással bármelyik nevezett blokk megszakítható, nem csak az, amelyikben

a megszakító utasítás szerepel. Ez lényeges eltérés a szoftvernyelvek szabályaitól, ahol mindig csak a pillanatnyilag soron levő program/alprogram szakítható meg.

10-31 példa: Nevezett blokk megszakítása.

```
initial
begin
        flag=16'b0010_0000_0000_0000;
        i=0;
        begin: block1
        while (i<16)
                begin
                        if (flag[i])
                        begin
                                $display ("True bit at element number %d", i);
                                disable block1;
                        end
                        i=i+1;
                end
        end
end
```

11. Irodalomjegyzék

[1] Szittya Ottó: Digitális és analóg technika informatikusoknak, I. és II. kötet, LSI Oktatóközpont, Budapest, 1999.

[2] Dejan Živković, Miodrag Popović: Impulsna i digitalna elektronika, Nauka, Belgrád, 1993.

[3] Samir Palnitkar: Verilog HDL, A Guide to Digital Design and Synthesis, Sunsoft Press, 1996.

[4] Michael Ciletti: Advanced Digital Design with the Verilog HDL, Prentice Hall, 2003.

[5] Morris Mano: Digital Design, III. kiadás, Prentice Hall, 2002.

[6] Matijevics István: Digitális Technika, Szabadkai Műszaki Főiskola, Szabadka, 2003.

[7] Janovics Sándor, Tóth Mihály: A logikai tervezés módszerei, Műszaki Könyvkiadó Budapest, 1973.

[8] Arató Péter: A logikai rendszerek tervezése, III. kiadás, Tankönyvkiadó, Budapest, 1990.

[9] Peter Ammon: Kapumátrix áramkörök, Berendezésorientált áramkörök, Műszaki Könyvkiadó, Budapest, 1989.

[10] Radomir Stanković, Milena Stanković: Logičko projektovanje, Nauka, Belgrád, 1991.

[11] Tihomir Aleksić: Logička sinteza digitalnih mreža, Naučna knjiga, Belgrád, 1975.

[12] Ripka Gábor: Felületre szerelhető alkatrészek, Műszaki Könyvkiadó, Budapest, 1992.

[13] Texas Instruments: Logic Selection Guide, Dallas, Texas, 2003.

[14] Fairchild: Logic Selection Guide, 2003.

[15] www.ti.com

[16] www.fairchildsemi.com

[17] www.prenhall.com

[18] www.xilinx.com

[19] www.altera.com

[20] www.atmel.com

[21] www.latticesemiconductor.com

[22] www.simucad.com