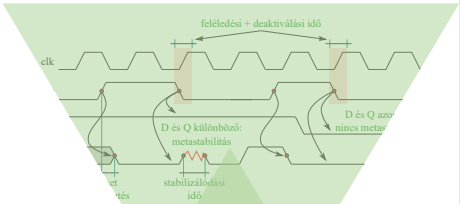
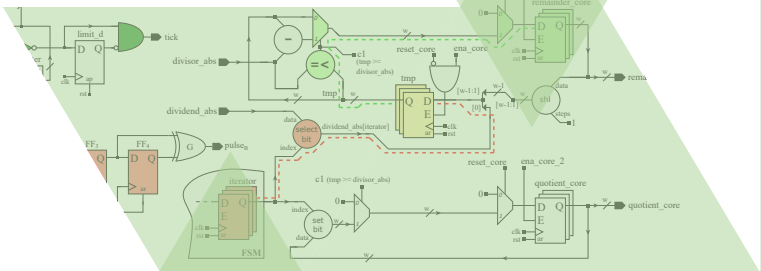
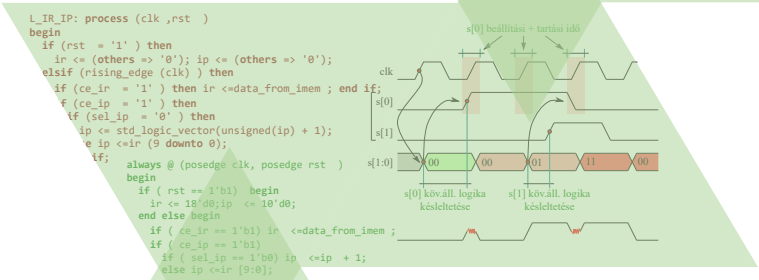


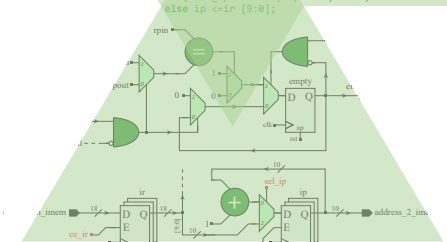
Horváth Péter

DIGITÁLIS RENDSZEREK MODELLEZÉSE ÉS SZINTÉZISE









L'Harmattan

Ez a mű a Creative Commons Nevezd meg! - Ne add el! - Így add tovább! 4.0 Nemzetközi Licenc feltételeinek megfelelően felhasználható.
https://creativecommons.org/licenses/by-nc-sa/4.0/



Digitális rendszerek modellezése és szintézise

Modelling and Synthesis of Digital Systems

Horváth Péter

https://orcid.org/0009-0006-0097-259X

Villamosmérnöki és informatikai tudományok / Electrical engineering, Electronic engineering, Information engineering (10880), Beágyazott rendszerek / Embedded systems (10124)

digitális rendszertervezés, RTL-tervezés, HDL-modellezés, funkcionális verifikáció

digital system design, RTL design, HDL modelling, functional verification

ISBN 978-963-646-446-2 (pdf)

DOI: https://doi.org/10.56037/978-963-646-446-2



https://openaccess.hu/

Horváth Péter

Digitális rendszerek modellezése és szintézise

Horváth Péter

DIGITÁLIS RENDSZEREK MODELLEZÉSE ÉS SZINTÉZISE

L'Harmattan, 2025

A kötet megjelenését támogatták:

Ericsson Magyarország Kft.

BME VIK, Elektronikus Eszközök Tanszéke

Pro Progression Alapítvány

C3S Elektronikai Fejlesztő Kft.

Második, javított kiadás, 2025

© Horváth Péter, 2025
© L'Harmattan, 2025

ISBN 978-963-646-494-3
ISBN 978-963-646-446-2 (pdf)
DOI: https://doi.org/10.56037/978-963-646-446-2

TARTALOM

Köszönetnyilvánítás 9

Bevezetés 11

1. A DIGITÁLIS RENDSZEREK TERVEZÉSÉNEK FOLYAMATA

1.1 Elvonatkoztatási szintek a digitális rendszertervezésben 13
1.2 A tervezési folyamat lépései 16
1.2.1 Iterációk és párhuzamosítás a tervezési folyamatban 24

2. RENDSZERTERV

2.1 Műholdfedélzeti kommunikációs modul rendszerterve 29
2.1.1 Szoftverközpontú megoldás 30
2.1.2 Hardverközpontú megoldás 33

3. VISELKEDÉSI MODELLEZÉS

3.1 A MiniCore mikroprocesszor viselkedési szintű modellje 38
3.2 Alkalmazásspecifikus áramkör viselkedési szintű modellje 49

4. RTL TERVEZÉS, HARDVERLEÍRÓ NYELVEK

4.1 Erőforrás-allokáció és ütemezés 58
4.2 RTL modellek formális leírása 61
4.2.1 Egy kis történelem 62
4.2.2 VHDL vs Verilog 63
4.2.3 Szimulációs és szintézis szemantika 63
4.3 Alapvető adattípusok 65
4.4 A HDL modellek szerkezete, utasítástípusok 66
4.4.1 Folytonos értékadás 67
4.4.2 Példányosítás 70
4.4.3 Folyamat 72
4.4.4 Egyidejűség és sorrendiség a HDL-ekben 76
4.5 RTL építőelemek 87
4.5.1 Regiszterek 87
4.5.2 Multiplexerek 91
4.5.3 Aritmetika 95

4.5.4 Regisztertömbök, memóriák 99
4.5.5 Állapotgépek 104
4.6 Újrafelhasználhatóság, generikus modellek 111
4.7 Számláló-alapú funkcionális egységek 116
4.8 Adatfeldolgozó rendszerek RTL modelljei 126
4.9 A MiniCore mikroprocesszor RTL modellje 136
4.9.1 A műveletvégző egység 139
4.9.2 A vezérlő egység 145

5. FUNKCIONÁLIS VERIFIKÁCIÓ

5.1 A verifikációs környezet 162
5.2 Verifikációs komponensek 164
5.3 Önellenőrző verifikációs környezetek 171
5.4 A gerjesztési minták előállítása 172
5.5 A verifikációs környezetek minősítése 175
5.6 A funkcionális verifikáció szintjei 179
5.7 Verifikációs keretrendszerek 182

6. A DIGITÁLIS ÁRAMKÖRÖK IDŐZÍTÉSI KÉRDÉSEI

6.1 A késleltetések fizikai háttere, metastabilitás 185
6.2 Időzítési kényszerek 193
6.2.1 Az időzítési kényszerek specifikálása 194
6.2.1.1 Órajelfeltétel *(clock constraint)* 194
6.2.1.2 Bemeneti késleltetés *(input delay constraint)* 201
6.2.1.3 Kimeneti késleltetés *(output delay constraint)* 202
6.2.1.4 Időzítési kivételek *(timing exceptions)* 204
6.2.2 Az időzítési kényszerek formális rögzítése 207
6.3 Adatátvitel órajeltartományok határán, reset-szinkronizáció 210
6.3.1 Órajelszinkronizáló áramkörök 211
6.3.2 Reset-szinkronizáció 226

7. A DIGITÁLIS ÁRAMKÖRÖK MEGVALÓSÍTÁSÁNAK TECHNOLÓGIÁI

7.1 Standard cellás integrált áramkörök 233
7.2 FPGA áramkörök 239

8. ÁRAMKÖRSZINTÉZIS

8.1 Manuális áramkörszintézis – példák 249
8.2 Automatizált áramkörszintézis 261
8.2.1 Logikai szintézis 263
8.2.1.1 Kódelemzés és RTL optimalizáció 265
8.2.1.2 Generikus szintézis és technológiafüggetlen logikai optimalizáció 266
8.2.1.3 Technológiai leképezés és technológiafüggő logikai optimalizáció 278
8.2.2 Fizikai szintézis 280
8.2.2.1 Fizikai szintézis standard cellás technológián 281
8.2.2.2 Fizikai szintézis FPGA technológián 287
8.2.3 Szintézis utáni ellenőrzések 289

9. ESETTANULMÁNY

9.1 A MiniCore mikroprocesszor szintézise és optimalizációja 292
9.1.1 Az interfészjelek szinkronizációja a lokális órajelhez 293
9.1.2 Az időzítési kényszerek rögzítése 294
9.1.3 Időzítési optimalizáció (**timing closure**) **296**
9.1.4 „Legyen űrös…” 301
9.1.4.1 A céltechnológia sugárzásállósága 303
9.1.4.2 SEU-védelem 304
9.1.4.3 Nagymegbízhatóságú állapotgépek 306
9.1.4.4 Hamming-2 állapotkódolás 308

Irodalomjegyzék 311

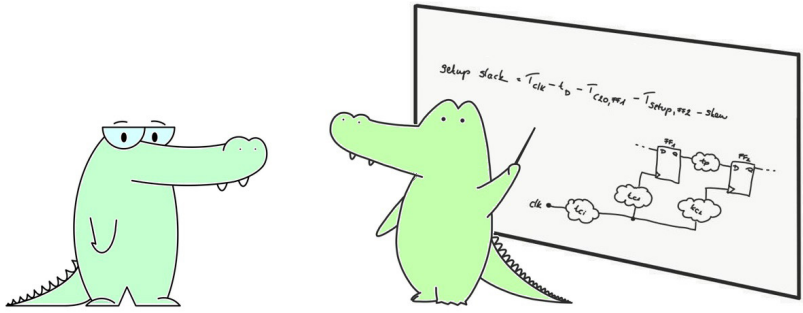
Rövidítésjegyzék 313

Tárgymutató 313

Köszönetnyilvánítás

A kézirat ellenőrzésében és „finomhangolásában” nyújtott segítségért köszönetemet fejezem ki doktori témavezetőmnek, Dr. Hosszú Gábornak, aki a nem is olyan régi idők emlékére vigyázó tekintetét ismét az írásomra vetette; egykori tanítványomnak és jelenlegi kollégámnak, Dr. Jani Lázárnak, aki a kéziratot ugyanazzal a minden apró részletre kiterjedő figyelemmel elemezte, mint ahogy teszi azt az RTL terveimmel azok verifikációja során; valamint jóbarátomnak, Nagy Gergelynek, aki a legviharosabb körülmények között is talált időt arra, hogy megjegyzéseivel javítsa a szöveg érthetőségét és pontosságát.

Dr. Horváth Péter
Budapest, 2020. április



Bevezetés

A félvezető technológia kezdetektől töretlen fejlődését számszerűen általában az egységnyi felületre integrált tranzisztorok számával és azok geometriai méretével jellemezzük. A modern integrált áramkörök összetettsége, számítási teljesítménye és megbízhatósága azonban nem kizárólag az azokat felépítő alkatrészek kedvező tulajdonságainak köszönhető. Bár a gyártástechnológia fejlődése elsősorban az alkatrészek méretének és energiafogyasztásának csökkenésében, valamint a kihozatal növekedésében nyilvánul meg, ami természetesen a „funkció-sűrűség" növelésének lehetőségét rejti, de ennek a lehetőségnek a kiaknázása, tehát az egyre nagyobb, bonyolultabb és kifinomultabb rendszerek tényleges létrehozása csak úgy lehetséges, ha a technológiai fejlődéssel annak közvetlen hasznosítója, a mikroelektronikai tervezés is lépést tart. A mai integrált áramkörökben tranzisztorok milliárdjai alkotnak összefüggő rendszereket, amelyek megtervezése csak hatékony, megbízható, számítógéppel segített tervezési módszerekkel (*Computer-Aided Design*, CAD) lehetséges.

Az ezredforduló óta a tervezési módszereknek az összetettség mellett az egyre inkább felértékelődő piaci szempontokkal – például a rövidülő piacra kerülési idővel és a termék életciklusával –, valamint a gyártási költségek növekedésével is számolniuk kell. Erre az összetett követelményrendszerre a mikroelektronikai tervezés a „Mit tervezzünk?" és a „Hogyan tervezzünk?" kérdésekre adott megfelelő válaszokkal reagált. A tervezés tárgyaivá mostanra az újrafelhasználható, újrakonfigurálható, programozható áramköri részletek váltak, az alkalmazott tervezési módszerek között pedig elterjedtek a magas elvonatkoztatási szintről induló, a lehető legtöbb részfolyamatot automatizáltan végző megoldások.

A modern tervezési módszerek azáltal támogatják az egyre összetettebb rendszerek kifejlesztését illetve csökkentik a tervezésre fordítandó időt, hogy az implementációs részletek időigényes, hibalehetőségekkel terhelt kidolgozását szoftver eszközök segítségével, automatizáltan végzik. Az újrafelhasználhatóság, amely szabványos algoritmusok és interfészek alkalmazását jelenti, a tervezés idejét és költségét is csökkenti. Az újrakonfigurálható, programozható megoldások – tárolt programú mikroprocesszorok és programozható logikai eszközök (*Programmable Logic Device*, PLD) – használata pedig lehetővé teszi a funkcionalitás módosítását a költséges újragyártás nélkül. Ugyancsak a gyártási költségek csökkentését – egyúttal a megbízhatóság növekedését – szolgálja a nagyfokú integráció, a digitális és analóg részegységek, kommunikációs alrendszerek és memóriák egyetlen chipre épülő megvalósítása. Az ilyen kompakt, újrafelhasználható részegységekből összeálló, de egészében véve alkalmazásspecifikus integrált áramköröket „rendszerchipeknek" (*System-on-Chip*, SoC) nevezzük.

Jelen írás az SoC áramkörök digitális részegységeinek tervezési módszereit mutatja be. E tervezési módszerek alapelvei, nevezetesen a hardverleíró nyelvek (*Hardware Description Language*, HDL), az automatizált áramkörszintézis és a hozzá kapcsolódó kiegészítő eljárások már az 1980-as évek végén körvonalazódtak, a '90-es évek során pedig széles körben

el is terjedtek. Bár az ezt követő három évtizedben – a gyártástechnológia látszólag megállíthatatlan fejlődése következtében és a termelékenység növekedésének igénye miatt – a tervezési módszerek finomodtak, hatékonyabbá váltak, sőt, új megközelítések is megjelentek – például a magas szintű szintézis (*High Level Synthesis*, HLS) –, de az alapok lényegében nem változtak, paradigmaváltás a tervezésben egyelőre (2020) nem következett be.



1

A digitális rendszerek tervezésének folyamata

1.1 Elvonatkoztatási szintek a digitális rendszertervezésben

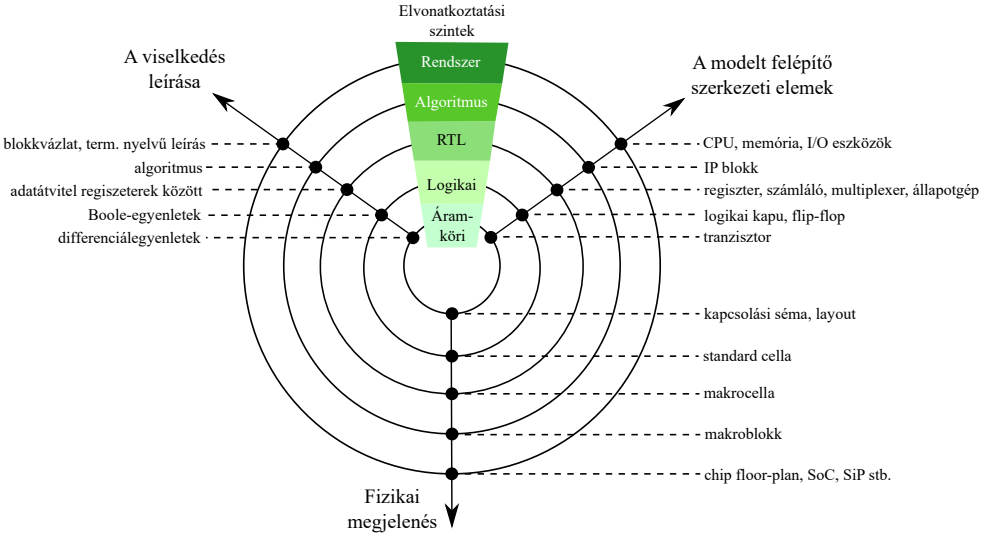
Az információs technológia szakterminológiájában, sőt tulajdonképpen mindennapi életünkben is elég elterjedten használt kifejezés az, hogy „digitális rendszer". Különböző szövegkörnyezetekben, különböző szakemberek számára ez a kifejezés igen eltérő dolgokat jelölhet, ezért mielőtt beleásnánk magunkat a tervezési módszerek tárgyalásába, célszerű rögzítenünk, hogy mit is takar a digitális rendszer fogalma számunkra. Egyszerűen – bár némileg felületesen – fogalmazva azt mondhatjuk, hogy digitális rendszer alatt olyan áramköröket értünk, amelyekben két domináns feszültségszint van jelen, tehát feladatukat bináris, kétértékű jelek továbbítása és manipulálása által valósítják meg.

Ebből a meghatározásból már egyértelmű, hogy áramköröket akarunk tervezni, és ha azt halljuk, hogy „áramkörtervezés", akkor valószínűleg az első kép, ami lelki szemeink előtt megjelenik, az egy papírlapra, ceruzával felvázolt, ellenállásokból, tranzisztorokból, kapacitásokból, műveleti erősítőkből és ezekhez hasonló alkatrészekből álló kapcsolási rajz. A kapcsolási rajz nem más, mint a leendő áramkörnek egy elvont modellje, vagyis egy olyan reprezentációja, ami a tervezett áramkörnek bizonyos tulajdonságait jól leírja, más tulajdonságait pedig elhanyagolja. Egy modell annál elvontabb, más szóval annál absztraktabb, minél kevesebb tulajdonságát adja vissza a modellezett rendszernek, minél kevesebb részletet tartalmaz a konkrét megvalósításra vonatkozóan. Egy kapcsolási rajz például egy olyan modell, amely – ha az egyes komponensek kivezetései közötti feszültség- és áramviszonyokat leíró matematikai összefüggéseket, vagyis az alkatrészek karakterisztikáit is a modell részének tekintjük – kiválóan leírja az áramkör belsejében, adott gerjesztés hatására kialakuló feszültségeket és áramokat, ugyanakkor semmit nem mond az áramkör fizikai méreteiről, mert nem tartalmazza az ehhez szükséges megvalósítási részleteket, például hogy egy adott műveleti erősítőből milyen tokozású kivitelt választunk majd.

Bár bizonyos alkalmazási területeken az áramkörtervezés valóban kezdődhet egy kézzel felrajzolt, alkatrészszintű kapcsolási rajzzal, de az összetett digitális áramkörök – pl. egy mikroprocesszor – megtervezése, pontosan a hihetetlen bonyolultság miatt, nem így történik. A folyamat sokkal elvontabb, absztraktabb, kevesebb megvalósítási részletet tartalmazó modellel, az ún. rendszertervvel kezdődik.[1] A digitális áramkörök tervezése során

[1] Érdekes módon a legelső lépések során a papír–ceruza módszer itt is igen elterjedt. Nem véletlen, hiszen a „tervezés" valójában fejben történik, az elképzeléseink külvilág felé való kommunikálásának pedig egyik leghatékonyabb eszköze – 21. századi technológia ide vagy oda – továbbra is az írás és a rajz.

aztán a rendszerterven kívül egy egész sor, különböző elvontságú modell áll elő, egy részük kézzel elkészítve, más részük gépi úton, CAD szoftverek segítségével, míg végül megszületik egy olyan részletgazdag modell, amely már az áramkör gyártási folyamatának alapjául szolgálhat. A digitális rendszertervezésben használatos elvonatkoztatási szintek és az azokra jellemző modellek tulajdonságait az ún. Gajski–Kuhn-féle Y-diagram (GK-diagram) segítségével szokás szemléltetni (lásd 1-1. ábra).



1-1. ábra. *A Gajski–Kuhn-féle Y-diagram*

A GK-diagram három szempont (nyilak) szerint vizsgálja az egyes absztrakciós szinteket (koncentrikus körök):

- A viselkedés megfogalmazásának formája: Hogyan írja le a modell a bemenetére adott és a kimenetén leolvasott információ közötti transzformációt? Ebből a szempontból tekintve a modell belső felépítése nem képezi a vizsgálat tárgyát. Azt vizsgáljuk, hogy a rendszer mit csinál, de azt nem, hogy hogyan.
- A modellt felépítő szerkezeti elemek: Milyen belső erőforrásokat használ a modell a funkció megvalósításához? Hogyan kapcsolódnak össze ezek a strukturális elemek az összetettebb funkció megvalósítása érdekében?
- Az adott elvonatkoztatási szintre jellemző összetettségű, méretű rendszer fizikai megjelenése: Az egyes szintekre jellemző önálló tervezési egységek összetettsége eltér. Egy önálló rendszerszintű tervezési egység (pl. egy mikroprocesszor) – abból adódóan, hogy kevés megvalósítási részletet tartalmaz – sokkal összetettebb funkciót írhat le, mint egy önálló tervezési egység kapuszinten (pl. egy összeadó áramkör).

A GK-diagram a következő absztrakciós szinteket definiálja:

- **Rendszerszint**. A rendszerszintű modell, vagy rendszerterv egy összetett áramkör nagyvonalú szerkezetét tükröző blokkvázlat. A rendszertervnek két fő célja van: rögzíteni, hogy az áramkör milyen funkcionális egységekből épül fel, és hogy e funkcionális elemek milyen módon kommunikálnak egymással.
- **Algoritmusszint**. Az algoritmusszintű modellek célja, hogy egy-egy funkcionális egység viselkedését formálisan megfogalmazzák. Éppen ezért ezt az elvonatkoztatási szintet viselkedési modellnek is szokás nevezni. A funkció egyértelműségét biztosító formális jelleg gépi nyelvek, például magas szintű programozási nyelvek alkalmazásával érhető el.
- **Regiszter-transzfer szint.** (*Register-Transfer Level*, RTL). A regiszter-transzfer szint elnevezés arra utal, hogy ezen az elvonatkoztatási szinten a cél az áramkört felépítő tároló elemek, flip-flopok és regiszterek közötti adatátvitel és transzformációk meghatározása. Az RTL modell tehát az algoritmusszintű modellnél kevésbé elvont, már olyan megvalósítási részleteket is tartalmaz, hogy egy adott adatfeldolgozási művelet során milyen belső regisztereket, memóriákat és aritmetikai/logikai műveletvégző elemeket használunk, továbbá rögzíti azt is, hogy egy összetett művelet során az egyes részfeladatokat milyen sorrendben, milyen ütemezéssel hajtjuk végre.
- **Logikai, más néven kapuszint**. A kapuszintű modellek az áramkört felépítő logikai kapuk és flip-flopok hálózatát írják le. Ennek megfelelően ezeket a reprezentációkat strukturális modellnek is nevezzük.
- **Áramköri, más néven tranzisztorszint**. Az áramköri szintű modellek azt írják le, hogy egy digitális funkció, például egy logikai kapu vagy egy flip-flop kapuszinten adottnak tekintett viselkedése hogyan valósítható meg tranzisztorok megfelelő összekapcsolásával. Ezen az elvonatkoztatási szinten beszélhetünk tehát kapcsolástechnikákról, amelyek lényegében erre a „hogyan" kérdésre adott válaszok, a rendelkezésre álló alkatrészek megfelelő összekapcsolására vonatkozó szabályrendszerek formájában (pl. CMOS kapcsolástechnika: felhúzó és lehúzó hálózat megvalósítása MOS-tranzisztorok segítségével).

A teljes folyamatot tekintve elmondható, hogy az összetett digitális rendszerek tervezése egy ún. *„top–down"*, vagyis felülről lefelé szerveződő módszer, ahol a „fent" és „lent" az elvonatkoztatás mértékére vonatkozik; elvont, absztrakt modellekből indulunk ki, majd a tervezés során – kézi vagy gépi úton – egyre több megvalósítási részletet dolgozunk ki, az elvonatkoztatás mértéke tehát egyre csökken, befelé haladunk a GK-diagram középpontja felé. A tervezés akkor ér véget, amikor a terv már minden olyan apró részletet tartalmaz, ami lehetővé teszi az áramkör gyártását. Fontos hangsúlyozni, hogy ez a top–down jelleg az elvonatkoztatás mértékére vonatkozik, nem pedig a funkció összetettségére. Ellenkezőleg, ha a funkció összetettségéről beszélünk, akkor inkább a *„bottom-up"* kifejezés állja meg a helyét, hiszen – ahogy a bevezetőben is említettük – a mai modern SoC áramkörökre jellemző összetettség úgy kezelhető, ha már meglévő, egyszerűbb funkciójú áramköri részletekből, ún. IP *(Intellectual Property)*-magokból építjük fel a bonyolultabb funkciójú rendszert.

1.2 A tervezési folyamat lépései

Az alábbi alfejezet az összetett digitális funkciót megvalósító áramörök tervezésének folyamatát, az ún. *„design flow”*-t mutatja be. E fejezet áttekintő jellegű, a tervezési folyamat egyes fázisainak jellemző problémáit és megoldásait a későbbi alfejezetek részletesen tárgyalják.

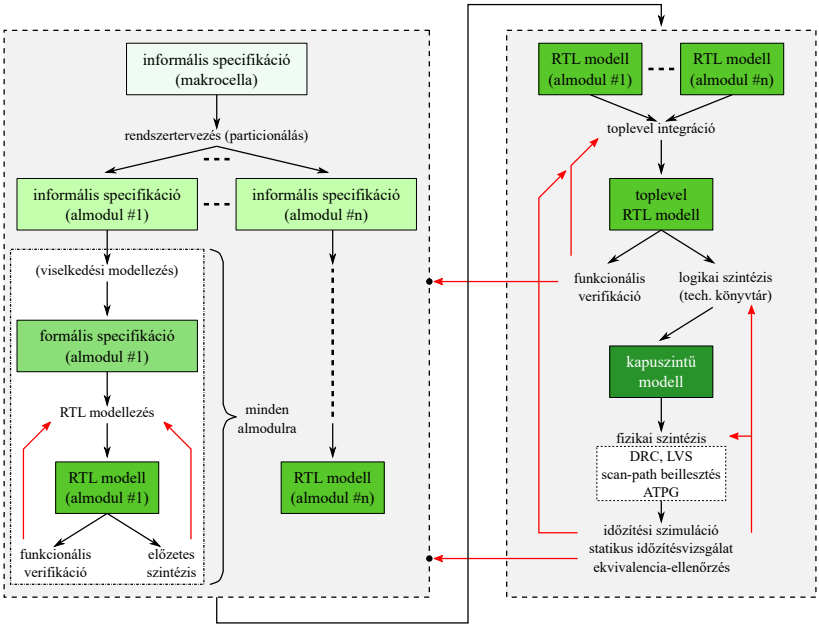
Az, hogy mit tekintünk „összetett digitális funkció”-nak, nézőpont kérdése. Egy mikrokontroller SPI *(Serial Peripheral Interface)* perifériavezérlő áramköre például tekinthető összetettnek, hiszen az elsődleges funkcióján – az SPI protokollnak megfelelő soros/párhuzamos átalakításon – kívül egy szabványos, a processzormaghoz való csatlakoztatást lehetővé tévő busz-interfészt is meg kell valósítani benne, aminek következtében a szükséges flip-flopok és egyéb logikai áramkörök száma több száz, sőt, a busz-interfésztől függően akár több ezer is lehet. A logikai funkción túlmenően, a fizikai tervezés során gondoskodni kell a belső erőforrások megfelelő táp-, órajel- és reset-ellátásáról, figyelembe kell venni az SPI buszjelek lokális órajelhez képesti aszinkronitását, és vizsgálni kell az előálló áramkör minden lokális és külső adatútjának időzítési tulajdonságait, hogy biztosítható legyen az előírt rendszerórajellel való kompatibilitás. Az SPI interfészáramkör tervezése tehát nem egyszerű feladat, mondhatjuk, hogy az áramkör összetett. Képzeljük el azonban egy olyan SoC áramkör tervezését, amelyben több általános célú mikroprocesszormag, grafikus processzor, gyorsítótár-vezérlő áramkör, DMA *(Direct Memory Access)*-vezérlő, megszakításvezérlő, számtalan perifériavezérlő áramkör és több, egyedi funkciójú, ún. alkalmazásspecifikus – jellemzően nagy számítási igényű digitális jelfeldolgozási (*Digital Signal Processing*, DSP) funkciót megvalósító – részáramkör kap helyet. Ebben a szövegösszefüggésben az „összetett” jelzőt aligha használnánk az egyik sötét sarokban megbúvó, apró SPI interfészáramkörre.

Ahogy azt a bevezetőben is említettük, a tervezési folyamat termelékenységének kulcsa az újrafelhasználhatóság, az ún. IP-alapú megközelítés, ami azt jelenti, hogy egy új rendszer megtervezése során igyekszünk felhasználni korábban megtervezett és kipróbált áramköri részleteket, nem próbáljuk meg minden új termék esetén feltalálni a melegvizet, ami által rengeteg időt nyerhetünk, ráadásul az előálló termék megbízhatóbb is lesz, hiszen a már kipróbált áramkörökben kisebb valószínűséggel fordul elő még fel nem fedezett hiba. Az újrafelhasználható modulokból való építkezés azt eredményezi, hogy a digitális rendszerek szerkezete fraktálszerű, az egyes funkciókat megvalósító áramkörök ún. áramköri hierarchiába szerveződnek. A legfelső hierachiaszinten lévő, ún. toplevel modul egyszerűbb részfunkciókat ellátó IP magokból és a közöttük lévű illesztő logikákból *(„glue logic”)* áll, és ugyanez igaz magukra az – alacsonyabb áramköri hierarchiaszinten lévő – IP magokra is.

Az ebben a fejezetben tárgyalt tervezési folyamat az áttekinthetőség kedvéért e hierarchiának csak két szintjét nevezi meg; a „makrocellát” és az azt felépítő „almodulokat”. Maguk az elnevezések kissé önkényesnek és semmitmondónak tűnhetnek (azok is...), aminek az az oka, hogy az áramköri hierarchia különböző szintjeire nincs széleskörűen elterjedt, szabványos megnevezés éppen amiatt, hogy a hierarchia szintjeinek száma és azok összetettsége áramkörről áramkörre változik. Az e fejezetben bemutatott fogalmak könnyebben megérhetők, ha egy konkrét példát képzelünk magunk elé; az állatorvosi ló

céljára egy mikrokontroller tökéletes választás. Ebben az esetben a teljes mikrokontroller egy makrocella, amely kisebb, de önállóan értelmezhető feladatot ellátó almodulokból, például mikroprocesszormagból, utasítás- és adatmemóriából, memóriavezérlőből és perifériavezérlő áramkörökből (pl. UART/SPI/I²C stb.) épül fel.

Azt várhatnánk, hogy a tervezési lépések sorozata, a *design flow* függ attól, hogy mennyire összetett a megvalósítandó funkció, azonban az itt bemutatott, általános tervezési folyamat az összetettségnek igen széles skáláján változtatás nélkül alkalmazható. A fenti példánál maradva, az SPI perifériavezérlő és a teljes mikrokontroller esetén is ugyanazokat a lépéseket kell elvégeznünk, bár az egyes lépésekhez szükséges idő, valamint az elkerülhetetlenül előforduló iterációk, visszalépések, újratervezések száma értelemszerűen nagyon eltérhet. Az 1-2. ábra egy makrocella, tehát egy önálló funkcióval rendelkező, egyszerűbb részáramköröket, almodulokat tartalmazó funkcionális egység tervezési folyamatának lépéseit mutatja be.



1-2. ábra. *Egy makrocella tervezési folyamata*

Informális specifikáció • Az informális specifikáció a makrocella felhasználója szempontjából fogalmaz meg követeléseket, általában valamilyen természetes nyelv segítségével, innen az „informális" jelző. Legfőképpen az interfészek jellemzőit, az elsődleges funkciót és egyéb tulajdonságokat, mint például fogyasztás, számítási teljesítmény stb. rögzíti. Már az informális specifikáció is tartalmazhat a makrocella belső szerkezetére vonatkozó információt, tipikusan blokkdiagramok formájában, amelyek a funkcionalitásra vonatkozó követelmények megértését segítik, és nem feltétlenül tükrözik pontosan a tervezés során megvalósított rendszer valódi szerkezetét. A természetes nyelven megfogalmazott specifikáció

nyilvánvaló hátránya, hogy nem mindig egyértelmű. Ennek a problémának az enyhítése céljából a felhasználói igényeket egy sor követelményként *(user requirement)* fogalmazzuk meg. Az egyes követelményekre az jellemző, hogy önmagukban rövidek – jellemzően egyetlen mondat – és csak egy-egy apró részletét rögzítik a kívánalmaknak. Ha például egy mikroprocesszormag specifikálásáról van szó, akkor az alábbiakhoz hasonló felhasználói követelmények képzelhetők el:[2]

- **REQ-CORE-0011**: *The microprocessor core shall implement a five-stage instruction pipeline.*
    - *Validation method: review*
- **REQ-CORE-0023**: *The microprocessor core shall provide a 1 instruction/clock cycle throughput.*
    - *Note: Assuming that the considered instruction flow does not include branch instructions and memory-reference instructions.*
    - *Validation method: simulation*
- **REQ-CORE-0056**: *The delay between the rising edge of the interrupt line and the execution of the interrupt service routine's first instruction shall not exceed 20 μs.*
    - *Validation method: measurement*

A fenti példákból a felhasználói követelményeknek két további fontos tulajdonsága is látható, nevezetesen az azonosíthatóság és a megkövetelt validációs eljárás.[3] Az egyes felhasználói követelményekhez egyedi azonosítót rendelünk. Az azonosíthatóság egyben nyomonkövethetőséget is jelent; a tervezési folyamat során az egyes tervezői döntések közvetlen összefüggésben lehetnek valamely felhasználói követelménnyel. E tervezői döntéseket dokumentáljuk, és a dokumentáció során az adott tervezői döntést az azt indikáló felhasználói követelményre való hivatkozással támasztjuk alá. Ha például hozunk egy döntést, amely szerint ötfokozatú utasításátlapolást *(pipeline)* megvalósító architektúrát fogunk létrehozni, akkor e döntésünket közvetlenül egy felhasználói követelménnyel igazolhatjuk. Ennek akkor van nagy jelentősége, ha a felhasználói követelmények a tervezési folyamat során megváltoznak. A változás három dolgot jelenthet: új követelmény születik, egy már létező követelmény megváltozik vagy egy követelményt törölnek. Az ilyen esetekben a korábban meghozott döntéseket újra meg kell vizsgálni. A fenti példánál maradva tegyük fel, hogy az ötfokozatú pipeline megvalósításra és az átbocsátóképességre vonatkozó követelményeket törlik. Ekkor azt a tervezői döntést, amely szerint egy bonyolult, nagy számítási kapacitású, ötfokozatú pipeline architektúrát hozunk létre, felül kell vizsgálni. Ha már nincs többé előírva sem a pipeline, sem az 1 utasítás/órajelciklus átbocsátóképesség, akkor egy sokkal egyszerűbb, kisebb – ezáltal olcsóbb –, könnyebben ellenőrizhető, pipeline nélküli architektúrát is készíthetünk. Természetesen ilyenkor mérlegelnünk kell azt is, hogy egy ilyen változtatást érdemes-e végrehajtanunk egy félkész terv rendelkezésre állása esetén, hiszen a változtatás akkor is többletmunkával jár, ha éppenséggel annak eredménye egy

[2] A példaként adott követelményeket szándékosan angol nyelven fogalmaztuk meg, szemléltetve, hogy a specifikáció informális jellegéből adódó nehézségeket az esetleges nyelvi különbségek is tovább mélyíthetik.

[3] Validáció alatt itt a termék felhasználói igényeknek való megfelelésének igazolását értjük.

egyszerűbb áramkör. Ez a többletmunka csak akkor éri meg, ha például az áramköri terv helyességének ellenőrzése, az ún. funkcionális verifikáció során a korábbi megoldáshoz képest jelentős időt nyerünk az áramkör egyszerűségéből adódóan.

A felhasználói követelményekhez rendelt validációs eljárás rögzíti, hogy az adott követelménynek való megfelelést milyen módon kell igazolnunk. A fenti példában a *review* kifejezés arra utal, hogy a terv részletes vizsgálata elegendő a követelménynek való megfelelés igazolásához. Ezt a vizsgálatot általában a megrendelő végzi, de az is lehetséges, hogy magát a vizsgálatot is a tervezést végző fél végzi el, és a vizsgálat eredményét a megrendelővel egyeztetett formában közli, dokumentálja. Egy másik lehetséges, némileg szigorúbbnak tekinthető igazolási mód a szimuláció, amit rendszerint akkor alkalmazunk, ha az adott követelménynek való megfelelés igazolása a terv igen mélyreható, időigényes vizsgálatát igényelné. Ekkor szoftver eszközöket, szimulátorokat használunk a terv megfelelőségének kimutatására, és a szimulációs eredményeket ugyancsak dokumentáljuk. Egy harmadik, a példában említett validációs módszer a mérés, amely a fentiek közül a legszigorúbbnak tekinthető. A megrendelő által fontosnak vélt követelményeknek való megfelelőséget általában méréssel kell igazolni. Kritikus követelmények esetén az is gyakori, hogy a mérési elrendezést és a mérési módszereket is a megrendelő írja elő.

A követelményekkel kapcsolatban már utaltunk rá, hogy a tervezési folyamat megkezdése után is változhatnak. Összetett rendszerek esetén a specifikáció önmagában is egy olyan bonyolult folyamat végeredménye, amelynek során nem ritkán hibák adódhatnak. A specifikáció hibái többnyire belső ellentmondásokként jelennek meg. Tegyük fel például, hogy egy UART *(Universal Asynchronous Receiver Transmitter)* vevőáramkört kell fejlesztenünk az alábbi követelmények figyelembevételével:

- **REQ-UARTRX-0001**: Az UART vonal szimbólumsebessége: 800 kbit/s.
- **REQ-UARTRX-0004**: A megbízhatóság növelése céljából az UART soros/párhuzamos átalakítónak egyenlő időközönként legalább 3 mintát kell vennie egy szimbólumból.
- **REQ-UARTRX-0011**: Az UART soros/párhuzamos átalakító áramkör órajelfrekvenciája nem lehet nagyobb 1 MHz-nél.

A fenti példa egy tipikusnak mondható szituációt szemléltet, amikor is egy követelmény-hármas bármely kiválasztott két eleme meghatározza a harmadik követelményben előírt tulajdonság maximális/minimális értékét vagy megvalósíthatóságát, de a harmadik követelmény ennek ellentmond.[4] További problémaforrás lehet, ha a megrendelő nem rendelkezik kellő kompetenciával a tekintetben, hogy az általa megálmodott rendszer egyáltalán megvalósítható-e, így már a rendszer specifikálása során kénytelen a kivitelező szaktudására támaszkodni. A specifikáció tehát – különösen összetett rendszerek esetén – a megrendelő és a tervező/gyártó közös munkája nyomán, kompromisszumos megoldások egész sorának kidolgozása és kölcsönös elfogadása útján jön létre, a tervezés folyamatával időben átlapoltan.

[4] A helyzet a hétköznapi megrendelő–kivitelező szituációkból is ismerős lehet, amikor is a kivitelező fél álláspontja szerint „Cégünk képes olcsón, jó minőségben és gyorsan dolgozni. A kedves megrendelő e háromból kettőt választhat."

Rendszertervezés • A rendszertervezés során az informális specifikációban megfogalmazott összetett funkciót részfeladatokra bontjuk (particionálás) és definiáljuk az egyes részfeladatokat végrehajtó almodulok közötti kommunikációs interfészeket. A rendszerterv még mindig informális; a legtöbb esetben például az áramkör szerkezetét blokkdiagramok, tehát egyszerű rajzok rögzítik, a kommunikációs interfészekről pedig sok esetben ekkor még csak annyit tudunk, hogy léteznek, de azt nem, hogy pontosan hogyan fogjuk megvalósítani őket. A különböző rendszertervek persze részletesség szempontjából nagyon eltérőek lehetnek. Egy újonnan kifejlesztett mikrovezérlő esetén például a processzormag és a perifériavezérlők közötti interfészekkel kapcsolatban elképzelhető olyan rendszerterv, amely csak annyit állít, hogy ezek az almodulok egy közös buszon fognak kommunikálni, de az is lehet, hogy erre a célra egy már létező, szabványosnak tekinthető megoldást, például Advanced eXtensible Interface-buszt, vagyis AXI-buszt választunk. A gyakorlatban e két véglet nem nevezhető tipikusnak, a kommunikációs interfészekkel kapcsolatban jellemzően olyan paramétereket rögzítünk a rendszertervezés során, mint hogy soros vagy párhuzamos legyen a kapcsolat, szükség van-e a FIFO *(First-In First-Out)* tárolókra két modul között az eltérő működési sebességek miatt, busztopológiát alkalmazunk vagy pont–pont kapcsolatokat, mekkora adatátviteli sebességre van szükség két almodul között, milyen adatkapcsolati protokollt használjunk stb. Ugyancsak a rendszertervezés része az ún. hardver/szoftver-szétválasztás *(hardware/software partitioning)*, ami annak eldöntését jelenti, hogy a definiált részfeladatok közül melyiket valósítjuk meg egy különálló, egyedi funkciót ellátó áramkörként, és melyiket hajtjuk végre egy általános célú mikroprocesszorra megírt program segítségével. Ha a rendszer hardver és szoftver komponenseket is tartalmaz, akkor a rendszertervben ugyancsak rögzíteni kell e komponensek közötti kommunikációs interfészek jellegét (memóriatérkép, mailbox, FIFO stb.).

Viselkedési modellezés, formális specifikáció • A rendszertervezés eszközei a követelményekhez hasonlóan informálisak, vagyis tartalmuk csak egy szubjektív értelmezési folyamat által érthető meg. Ahhoz, hogy elkerüljük a félreértéseket, legalább egy közös szabályrendszerre szükség van, ami rögzíti például, hogy a blokkdiagramon a szürke doboz ezt és ezt, a szaggatott vonal pedig azt „szokta jelenteni". Érezhető, hogy ezek a laza konvenciók egy bizonyos összetettség felett elégtelenek a modellezéshez, ezért – ha a rendszer bonyolultsága megkívánja – a particionált, informális rendszerterv részegységeinek leírására formális nyelvű modell készül, jellemzően valamilyen magas szintű programozási nyelv segítségével. A viselkedési modellt formális specifikációnak is nevezik, hiszen ez az a reprezentáció, ami a következő lépés, az RTL modellezés szempontjából referenciának, specifikációnak tekintendő. A viselkedési leírások elkészítésének egy speciális, a modern SoC áramkörök tervezése során elterjedten használt módja az ún. tranzakciószintű modellezés (*Transaction Level Modeling*, TLM), amelynek lényege, hogy a rendszerkomponensek viselkedésének megfogalmazását és az azok közötti kommunikációs interfészek megvalósításának részleteit egymástól függetleníti. A tranzakciószintű modellezés legelterjedtebb eszköze a SystemC nevű C++ osztálykönyvtár.

RTL tervezés, RTL modell • Az RTL modell fogalmának kissé ködös meghatározása, amely szerint az áramkört felépítő tároló elemek, flip-flopok, regiszterek közötti adatátvitelt és transzformációkat írja le, egy példával tehető szemléletesebbé: Egy algoritmusszintű modellben a belső változókon végzett bonyolult aritmetikai műveleteket a modellező nyelv operátorainak (pl. összeadás, szorzás stb.) és vezérlési szerkezeteinek (pl. elágazás, ciklus stb.) használatával írhatjuk le. Egy RTL modell ennél kevésbé elvont, több megvalósítási részletet tartalmaz, nevezetesen azt, hogy a művelet elvégzése során pontosan hány regisztert használunk a köztes eredmények eltárolásához, hány lépésben, ún. vezérlési ciklusban[5] hajtjuk végre a műveletet és hogy az egyes vezérlési ciklusokban milyen aritmetikai/logikai erőforrásokat veszünk igénybe. Bár az RTL modellek is ábrázolhatók konvencionális, tehát jelentésüket tekintve közös megegyezésen alapuló alkatrész-szimbólumokból (regiszter, összeadó, multiplexer stb.) álló, ún. RTL kapcsolási rajzokkal, de a formális, egyértelmű jelleg megtartása érdekében az RTL modellezés elsődleges eszközei a hardverleíró nyelvek. A HDL leírásokat a klasszikusnak tekinthető design-flow-ban, tehát az esetek túlnyomó részében kézzel készítjük a viselkedési modellt specifikációnak tekintve. Bizonyos alkalmazási területeken használatosak olyan szoftvereszközök is, amelyek egy megfelelően megírt viselkedési modellből automatikusan is képesek RTL modellt előállítani. Ezeket a szoftvereszközöket magas szintű szintéziseszközöknek nevezzük.

Funkcionális verifikáció • Az RTL modell funkcionális helyességét szimulációval ellenőrizzük. A funkcionális szimulációra egyszerű almodulok esetén használható HDL tesztkörnyezet, de összetett, sok almodulból álló makrocelláknál magasabb szintű verifikációs környezetek és módszerek alkalmazása jellemző (pl. *e Reuse Methodology,* eRM; *Unified Verification Methodology,* UVM).

Előzetes szintézis • Az RTL modell már kellően sok megvalósítási részletet tartalmaz ahhoz, hogy belőle automatikus úton, tehát CAD-szoftverek segítségével állítsunk elő kapuszintű reprezentációt. Ezt a folyamatot nevezzük automatizált RTL szintézisnek, vagy áramkörszintézisnek. A regiszter-transzfer szinten alkalmazott HDL-ek szintaktikai sokszínűsége miatt azonban egyazon funkció többféleképpen megfogalmazható, de a különböző nyelvi szerkezeteket a szintéziseszközök nem mindig kezelik egységesen, és a kódolási stílusnak nagy hatása lehet a leírásból szintetizált áramkör tulajdonságaira. Annak eldöntése, hogy az RTL modell valóban azt az áramkört írja-e le formálisan, amelyet a tervező elképzelt (erre az angol nyelvű szakirodalom úgy utal, hogy *RTL intent*), előzetes szintézis útján lehetséges. Az előzetes szintézis eredményéből az áramkör időzítési viszonyaira becslés adható, amelyet az RTL tervező a modell finomítására használhat fel, szükség esetén újratervezve a modell kritikus részleteit. Az előzetes szintézis nem optimalizált áramkört eredményez, amelyet a fejlesztés további fázisaiban nem használnak fel, csupán egy visszacsatolást jelent az RTL tervező számára.

[5] A vezérlési ciklus legegyszerűbb esetben magát az órajelciklust jelenti, de léteznek bonyolultabb, több órajelet tartalmazó megoldások is, amelyekben egy-egy vezérlési ciklus több órajelciklus lehet. Ilyen esetekben az órajel helyett inkább a fázisjel és a fázisjel-séma kifejezések használatosak.

Toplevel integráció • A makrocellát felépítő almodulokat más-más tervező vagy akár komplett tervező csapat készítheti. Ezeket az almodulokat a toplevel integrációnak nevezett lépés során egyetlen, magasabb áramköri hierarchiát képviselő HDL modellbe integráljuk. Az almodulok modelljeinek példányosításán túlmenően jellemzően a toplevel integrációnak a feladata, hogy a különböző órajel-forrásokról járó almodulok közötti, ún. órajel-szinkronizáló megoldásokat szükség esetén beépítse (*Clock Domain Crossing*, CDC). A makrocellát felépítő almodulok automatizált szintézisét vezérlő parancsfájlok (szintézis scriptek) eltérhetnek, ezeket tehát integrálni kell egy globális, makrocella-szintű szintézis parancsfájlba, ami ugyancsak a toplevel integrátor feladata.

Logikai szintézis • A makrocella toplevel HDL modelljéből automatizált logikai szintézis – más néven RTL szintézis – útján egy kapuszintű, a céltechnológia cellakönyvtárának alapelemeit és azok összeköttetéseit leíró, ún. netlistát állítunk elő. A logikai szintézisnek két, jól elkülöníthető részfolyamatát különböztetjük meg:

- **Generikus szintézis**: Az RTL modellből olyan netlista készül, amelynek elemei generikus, általánosan használt logikai elemek: logikai kapuk, flip-flopok és memóriák. Ezek az alapelemek még csak absztrakt modelljei a leendő áramkörben található logikai elemeknek, ami azt jelenti, hogy fizikai paramétereket – késleltetést, geometriai méretet vagy fogyasztást – még nem rendelünk hozzájuk, pusztán logikai viselkedésük ismert. Ezt a modellt technológiafüggetlen kapuszintű modellnek is nevezzük. A tervezőrendszerek lehetővé teszik azt, hogy ezen a modellen logikai szimulációt végezzünk, meggyőződve arról, hogy az automatizált szintézis során előállt netlista valóban ugyanazt a funkciót valósítja meg, mint amit az RTL modell.
- **Technológiai leképezés** *(technology mapping)*: A generikus netlista minden eleméhez hozzárendeljük a céltechnológia cellakönyvtárának egy elemét, amely képes ellátni annak a konkrét generikus logikai elemnek a feladatát. A technológiafüggő kapuszintű modellnek is nevezett netlista elemeihez már rendelhetők fizikai paraméterek, mint késleltetés, fogyasztás, geometriai méret, hiszen a cellakönyvtár elemeinek ezek a tulajdonságai jól ismertek. Egy ezen a modellen elvégzett logikai szimuláció tehát már pontosabb képet festhet a leendő áramkör működéséről, mert a logikai alapelemek késleltetéseit már figyelembe tudja venni. A gyakorlatban azonban ez a fajta szimuláció nem jellemző, mert az alapelemek közötti összeköttetések késleltetésével még nem számol, ami – különösen modern technológiák esetén – túlságosan is jelentős mértékben befolyásolja az összképet ahhoz, semhogy a tisztán kapukésleltésekkel számoló időzítési szimulációnak értelme legyen.

Fizikai szintézis

- Standard cellás ASIC technológia esetén a fizikai szintézis magában foglalja a makrocella alaprajzának (*„floor-plan”*: befoglaló alakzat, táp- és földsínek, cellasorok, huzalozási csatornák stb., lásd 8.2.2. pont), a különböző standard cellák elhelye-

zésének és összeköttetéseiknek megtervezését. A standard cellák elhelyezése és a huzalozás kialakítása automatizált úton, CAD szoftverek segítségével történik, az eljárás neve – nagy meglepetésünkre – elhelyezés és huzalozás, angol szakirodalomban *place&route*.

- Újrakonfigurálható hardver (pl. *Field-Programmable Gate Array*, FPGA vagy *Complex Programmable Logic Device*, CPLD) esetén a hardver már készen van, a felhasználó – jelen esetben a makrocella tervezője – csak a már meglévő hardverelemek konfigurálását végzi. Kijelöljük, hogy a technológiafüggő kapuszintű modell melyik elemét az FPGA/CPLD melyik konkrét komponense fogja megvalósítani, majd a huzalozási erőforrásokat – kapcsolómátrixokat – is felkonfiguráljuk a szükséges logikai kapcsolatok kialakításához. Az újrakonfigurálható hardver-alapú tervezés terminológiája ezt az – egyébként ugyancsak automatizált – konfigurációs folyamatot nevezi elhelyezésnek és huzalozásnak.

Standard cellás technológia esetén a fizikai terv ellenőrzésére a két legalapvetőbb módszer az ún. DRC *(Design Rule Check)* és LVS *(Layout vs. Schematic)* teszt. A DRC, vagyis tervezésiszabály-ellenőrzés a terv gyárthatóságát ellenőrzi, az LVS teszt segítségével pedig meggyőződhetünk róla, hogy a layout-kialakítás parazita elemei nem befolyásolják-e magát az áramköri funkciót. Újrakonfigurálható hardver esetén ezeket az ellenőrzéseket nekünk mint az FPGA vagy CPLD felhasználóinak már nem kell elvégeznünk, hiszen az eszköz kifejlesztője és gyártója már elvégezte.

Scan-path beillesztés, ATPG • Annak érdekében, hogy az áramkör belső állapota hozzáférhető legyen a tesztelés során, a belső regisztereket olyan cellakönyvtári elemekkel kell megvalósítani, amelyek átkapcsolhatók ún. *scan-path* üzemmódba. Scan-path üzemmódban az összes belső regiszter egy vagy több shift-regisztert képez, amely kívülről tölthető (beállítás, *control*) és kiolvasható (megfigyelés, *observe*). A makrocella teszteléséhez szükséges tesztszekvenciák előállítására automatikus módszerek használatosak, ezek gyűjtőneve ATPG *(Automatic Test Pattern Generation)*.

Időzítésvizsgálatok • Az elkészült fizikai terv automatizált úton készült a HDL nyelven megfogalmazott, funkció szempontjából szimulációval ellenőrzött RTL modell alapján, és igen egyszerű áramköri elemek tömkelegéből épül fel; flip-flopokból és logikai kapukból. Az RTL modell szimulációjánál nem vettük figyelembe a leendő áramkör alkatrészeinek olyan tulajdonságait, mint például a késleltetés.[6] A fizikai tervezés után előálló, késleltetéssel bíró logikai kapukból és regiszterekből álló netlistán kétféle vizsgálatot kell elvégeznünk; le kell ellenőriznünk, hogy az alapelemek és a közöttük lévő vezetékezés késleltetése lehetővé teszi-e a célul kitűzött órajel-frekvencia alkalmazását, továbbá meg kell győződnünk róla, hogy az automatizáltan előállított kapuszintű modell ugyanazt a funkciót valósítja-e meg, mint az RTL modell. Az előbbi ellenőrzéshez ún. statikus időzítésvizsgálatot

[6] Az RTL szimulációnak ez nem volt, nem is lehetett célja, hiszen az egy sokkal elvontabb modell, amelyben valójában még „alkatrészekről" sem beszélhetünk.

(*Static Timing Analysis*, STA), az utóbbihoz időzítési szimulációt, más néven dinamikus időzítésvizsgálatot végzünk.

- **STA**: A megvalósított áramkör ismert késleltetéseit egy, a tervező által definiált késleltetési kényszer-rendszer *(constraint)* alapján kiszámított „maximálisan megengedhető" késleltetésekkel hasonlítja össze.
- **Időzítési szimuláció**: Az RTL modellhez készített tesztkörnyezetben az RTL modell helyett a szintetizált netlistát példányosítva logikai szimulációt végzünk, amely a kapu- és vezetékkésleltetéseket is figyelembe veszi.

Az előállított áramkör belső időzítési tulajdonságai nemcsak maguknak az alapelemeknek a késleltetéseitől függenek, hanem az alkatrészek közötti vezetékezés által létrehozott parazita kapacitások okozta többlet késleltetésektől is. A vezetékezés tulajdonságai – vezetékszakaszok ellenállása, vezetékszakaszok közötti parazita kapacitások értéke – standard cellás technológia és újrakonfigurálható hardver esetén is pontosan ismert, tehát az időzítési tulajdonságok mindkét esetben pontosan modellezhetők. Azt a folyamatot, amely során az időzítési információt kinyerjük a rendelkezésre álló modellekből, *back-annotation*-nek nevezzük.

A fenti tervezési folyamat standard cellás alkalmazásspecifikus integrált áramköri (*Application-Specific Integrated Circuit*, ASIC) technológián és újrakonfigurálható hardver esetén gyakorlatilag teljesen megegyezik, kivéve a fizikai tervezés konkrét kivitelezését, ahol értelemszerűen elkerülhetetlen, hogy az alkalmazott CAD-algoritmusokban eltérések legyenek a két technológia fizikai szintű eltéréseiből adódóan. Az időzítési vizsgálatok elvégzése után azonban mindkét esetben elmondható, hogy a tervezési folyamat véget ért; előálltak az alábbiak:

- Egy netlista, vagyis egy adottnak tekintett alapelemkészletből felépülő kapcsolási séma, amely funkcionálisan megfelel az elvárásainknak (leggyakrabban ún. *Electronic Design Interchange Format*, EDIF-fájl vagy kapuszintű HDL modell).
- A belső időzítési tulajdonságokat, vagyis az adatutakat jellemző késleltetési adatokat leíró adatbázis (*Standard Delay Format*, SDF-fájl).
- Standard cellás technológia esetén a pontos layout-rajzolatot leíró adatbázis (*Graphic Database System*, GDSII, vagy *Open Artwork System Interchange Standard*, OASIS formátumú fájl).

A felsorolt információk összessége már lehetővé teszi standard cellás technológia esetén az áramkör gyártását, újrakonfigurálható hardver esetén pedig a konkrét eszköz felkonfigurálásához szükséges programozófájl előállítását.

1.2.1 Iterációk és párhuzamosítás a tervezési folyamatban

A legegyszerűbb esetektől eltekintve a tervezési folyamat szinte mindig tartalmaz iterációs hurkokat, ahogy az az 1-2. ábrán is látszik. Minden olyan eljárás, amely valamilyen

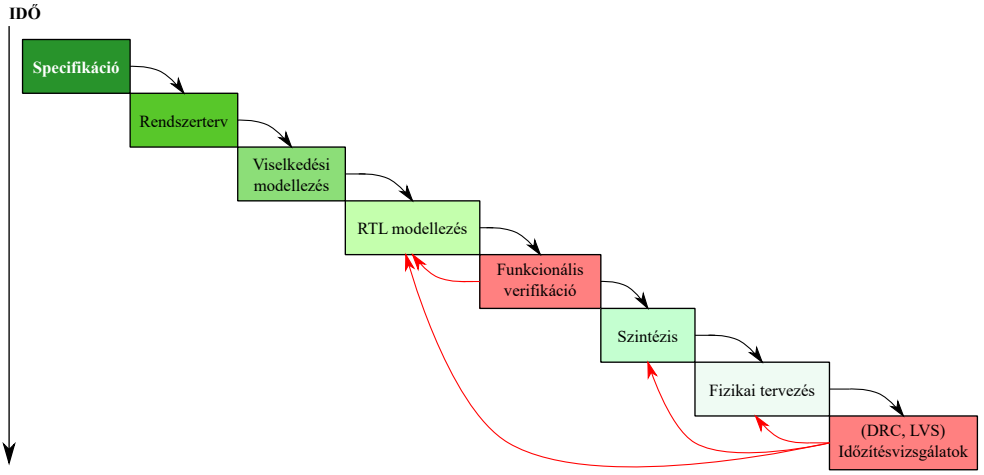
szempont alapján ellenőrzi egy tervezési lépés sikerességét, egyúttal értelemszerűen egy lehetséges iterációs hurkot is jelent, amely során a terv addig elkészült részeit újratervezzük, vagy legalábbis finomítjuk, pontosítjuk, optimalizáljuk *(refinement, optimization)*. Összetett rendszerek esetén a tervezés során felmerülő nehézségek akár a specifikáció módosítását is igényelhetik, ilyenkor természetesen a rendszer minden, addig elkészült reprezentációját frissíteni kell. Ezek az iterációk nyilvánvalóan növelik a fejlesztési időt, ennek ellenére a lehető legtöbb tervezési lépést igyekszünk valamilyen módon önmagában ellenőrizni, vagy legalábbis csökkenteni az iterációs hurok által bezárt tervezési lépések számát. Erre azért van szükség, mert minél kisebb az iterációs hurok, annál könnyebb felderíteni, hogy hol követtünk el hibát. Erre a leglátványosabb példa az almodulok önálló funkcionális verifikációja, ami a szoftverfejlesztésben ismert „*unit test*" megfelelőjének tekinthető. Minden almodult a saját specifikációja alapján ellenőrzünk, mielőtt beépítenénk azt egy magasabb áramköri hierarchiát jelentő makrocellába, így ha a makrocella ellenőrzése során valami nem stimmel, akkor tudhatjuk, hogy a hiba okát nem magukban az almodulokban kell keresnünk – hiszen azok önmagukban már helyesnek bizonyultak a unit test során –, hanem a hibát valószínűleg a toplevel integráció során követtük el (például elfelejtettünk egy összeköttetést megvalósítani két almodul között, nem megfelelően alakítottuk ki a reset-hierarchiát stb.). Sajnos a gyakorlatban az is előfordul, hogy a unit test elvégzése ellenére a toplevel modul verifikációja során mégiscsak az derül ki, hogy egy almodulon belül van a hiba. Ez egy nagyobb iterációs hurkot jelent, hiszen vissza kell nyúlni egészen a problémás almodul RTL modelljéig – sőt, azt is meg kell vizsgálni, hogy az almodul verifikációja során miért nem derült fény arra a bizonyos hibára. Mindez természetesen megnövekedett fejlesztési időt jelent.

Hasonló iterációs hurok forrása lehet az előzetes szintézis eredménye. Ebben a lépésben arról győződhetünk meg, hogy a szintézisszoftvernek sikerült-e felismernie minden, általunk leírt nyelvi szerkezetet, és olyan áramkört valósított-e meg, amit elképzeltünk. A fejlesztő által jól ismert szintézisszoftver persze „nem szokott meglepetést okozni" ebben a lépésben, ennek az ellenőrzésnek leginkább akkor van jelentősége, ha olyan szoftverrel kell dolgoznunk, amivel korábban még nem találkoztunk. Jellemző példa erre a szituációra az FPGA eszközökben található blokk-RAM modulok automatikus „felismerésének" *(inference)* kérdése. Az RTL HDL modell elkészítésénél mindig arra törekszünk, hogy a kód technológia- és gyártófüggetlen legyen, tehát bármilyen technológiára (standard cellás ASIC, FPGA, CPLD), és bármilyen gyártó eszközére (tetszőleges FPGA családra) változtatás nélkül szintetizálható legyen. Ezt úgy érhetjük el, hogy a memória jellegű funkciót nem egy, a gyártó által rendelkezésünkre bocsátott, specifikus almodul példányosításával építjük be a modellbe, hanem megának a memóriának a viselkedését fogalmazzuk meg a HDL nyelvi eszközeivel. Ezt sokféleképpen megtehetjük, és a leírásunk viselkedése nem feltétlenül egyezik meg a konkrét eszközben elérhető blokk-RAM modulok viselkedésével. Ekkor a szintéziseszköz „nem ismeri fel", hogy memóriáról van szó, a funkciót tehát kénytelen tengernyi flip-flop segítségével megvalósítani, ami a beépített blokk-RAM modulokat használó megoldáshoz képest összehasonlíthatatlanul rosszabb, különösen nagyméretű memóriablokkok esetén. Annak érdekében, hogy ne nekünk kelljen „kikísérletezni", milyen erőforrásokat milyen nyelvi szerkezetekkel tudunk megvalósítani, a szintéziseszköz

fejlesztője rendszerint rendelkezésünkre bocsát egy sor HDL példaleírást *(coding rules, coding guideline)*, melyeket útmutatónak tekinthetünk az RTL tervezés során.

Ugyancsak iterációs hurokhoz vezet, ha az időzítésvizsgálatok során kiderül, hogy a célul kitűzött időzítési követelményeknek az automatikus logikai és fizikai szintézis során előállított áramkör nem felel meg. Ebben az esetben is az a célunk, hogy az iterációs hurok a lehető legkisebb legyen, tehát először próbálkozhatunk a fizikai terv finomításával, például az időzítés szempontjából kritikus erőforrások (regiszterek, I/O bufferek) kézi elhelyezésével. Annak ellenére, hogy a fizikai tervezés digitális áramkörök esetén teljesen automatizált, az utólagos, kézi beavatkozás lehetőségét minden tervezőrendszer biztosítja. Ha a fizikai terv ilyen módon való módosításával, optimalizálásával nem érünk el jobb eredményt, akkor kénytelenek vagyunk egy nagyobb iterációs hurkot megvalósítani az RTL modell módosításával, remélve, hogy a logikai és fizikai szintézis újbóli elvégzésével elérjük a kívánt eredményt.

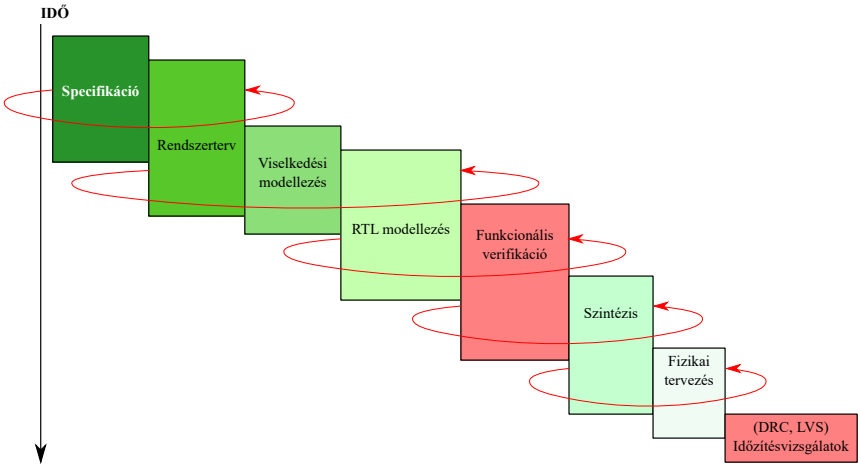
Az 1-2. ábrán látható folyamatábra és az azt követő leírás a fejlesztési folyamat lépéseiről azt sugallja, hogy e lépések szorosan egymást követve vezetnek el standard cellás céltechnológia esetén a gyártásra alkalmas modellhez, programozható hardver esetén pedig egy működő prototípushoz. Ez tulajdonképpen igaz is – leszámítva az előző bekezdésben bemutatott „megtorpanásokat", visszacsatolásokat –, egyszerű rendszerek esetén követhető a tervezési fázisok szigorú egymásutániságát leíró folyamatmodell, amit – az 1-3. ábra alapján belátható módon – vízesésmodellnek nevezünk.



1-3. ábra. *A fejlesztési folyamat vízesésmodellje*

A vízesésmodell fontos tulajdonsága, hogy az n+1-edik tervezési fázis nem kezdődhet el addig, ameddig az azt megelőző, n-edik tervezési fázis be nem fejeződött. Összetett rendszerek esetén ennek a megközelítésnek van egy óriási hátránya: Ha az egyes tervezési fázisok önmagukban is időigényesek, akkor az esetleges visszacsatolások során, amikor az n+1-edik fázisban kiderül, hogy az n-edik fázist – legalább részben – újra kell kezdeni, akkor az n+1-edik fázis nem halad. Összetett funkció esetén ez tulajdonképpen azt jelenti, hogy az előre

nem látható helyzetek, hibák, amelyek részleges újratervezéshez vezetnek, nemcsak hogy gyakrabban fordulnak elő, de a – fejlesztési időben mért – költségük is nagyon nagy. A mai modern, nagy bonyolultságú digitális rendszerek fejlesztése során ezért az egyes tervezési részfeladatokat az 1-4. ábrán látható módon átlapoltan hajtjuk végre.



1-4. ábra. *A fejlesztési folyamat átlapolódó részfeladatai*

E folyamatmodell alapötlete az, hogy minden fázist azonnal elkezdünk, amint a – legalább az elkezdéshez – szükséges információ rendelkezésre áll.

- A specifikáció legelső verziójának a rendelkezésre állásakor elkezdődik a rendszertervezés, de – ahogy azt az informális specifikációról szóló bekezdésben is említettük, a rendszertervezés során felmerülő problémákra adott megoldások nemcsak a rendszertervet magát, hanem a specifikációt is módosíthatják.
- Ha a rendszertervben már egyértelműen azonosíthatók bizonyos részfunkciók, akkor megkezdődhet ezek viselkedési modellezése.
- Ha elkészült egy egyszerű funkció viselkedési modellje, akkor elkezdődhet az RTL modell elkészítése is.
- Az RTL modell jellemzően sok-sok almodulból áll. Ahhoz, hogy elkezdődhessen a funkcionális verifikáció folyamata, nem szükséges, hogy a teljes toplevel RTL modell rendelkezésre álljon, hiszen az almodulok önmagukban is verifikálhatók, sőt verifikálandók.
- Az automatizált áramkörszintézishez szükséges időzítési peremfeltételek és szintézis parancsfájlok előkészítése a teljes RTL modell nélkül is elkezdődhet.
- Az almodulok fizikai tervezése standard cellás ASIC és FPGA esetén is függetleníthető egymástól, tehát egy almodul fizikai terve úgy is elkészíthető, hogy a toplevel modul RTL tervezése még nem fejeződött be (ez FPGA technológián kevésbé jellemző megoldás, de technikailag ott is kivitelezhető).

Általánosságban azt lehet mondani, hogy az n+1-edik tervezési fázis az n-edik tervezési fázis elkezdése után bármikor elkezdődhet, de addig nem fejeződhet be, amíg az n-edik fázis be nem fejeződött. Az ilyen formán történő „előre dolgozás" persze kockázatot is rejt; képzeljük el, hogy az algoritmusszintű modelleken számításiteljesítmény-becslés céljából futtatott tesztek eredményei miatt a rendszerterv (pl. a hardver/szoftver-szétválasztás jellege) megváltozik, miközben bizonyos almodulok esetén már az RTL modellezés is elkezdődött. Ez azt jelenti, hogy minden, előre nem látott változást az összes, éppen futó részfolyamatnak „le kell követnie", ami értelemszerűen többletmunkát jelenthet (ezt reprezentálják a piros nyilak az 1-4. ábrán). A tapasztalat azonban azt mutatja, hogy az összetettségnek azon a szintjén, amely a mai modern digitális rendszereket jellemzi, ez a fajta kockázatvállalás az esetek többségében kifizetődik.



2

Rendszerterv

A rendszertervezés fázisában előállított modell, röviden a rendszerterv egy összetett áramkör szerkezetét tükröző blokkvázlat, és az elképzelt rendszer legfontosabb funkcionális tulajdonságait rögzítő egyéb, nem formális eszközökkel leírt adathalmaz. A „nem formális eszközök" alatt ezen a ponton szinte bármit lehet érteni, amivel tervezői döntések rögzíthetők, beleértve egyszerű szöveges leírásokat, rajzokat, táblázatokat stb. A rendszertervezés során olyan döntések születnek, amelyek igen nagy hatással lehetnek a tervezési folyamat végén előálló termék tulajdonságaira. Egy rendszerszinten meghozott rossz döntés még akkor is eredményezhet használhatatlan végterméket, ha egyébként a rendszertervet követő összes tervezési lépést tökéletesen kivitelezzük. Nem véletlen tehát, hogy az összetett digitális rendszerek fejlesztése során igen nagy hangsúlyt fektetünk a rendszerterv elkészítésére, és a feladatot rendszerint a legtapasztaltabb mérnökökre bízzuk.

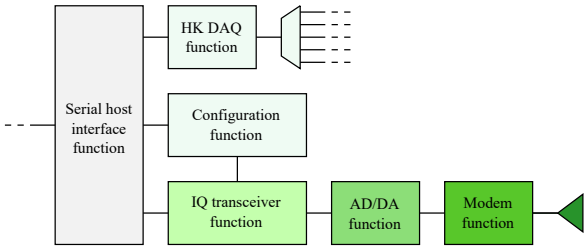
Általánosan használható, formális tervezési módszerek a rendszertervek elkészítéséhez nem jellemzők, ezért a tervezési folyamatnak ezt a lépését egy konkrét példa segítségével szemléltetjük.

2.1 Műholdfedélzeti kommunikációs modul rendszerterve

Tűzzük ki célul egy műholdfedélzeti rendszerekbe modulszerűen beépíthető, általános célú kommunikációs egység megtervezését. Induljunk ki az alábbi követelményjegyzékből:

- **REQ-COM-0001**: A modul legyen alkalmas kétirányú, half-duplex, S-sávú RF *(Radio Frequency)* kapcsolat megvalósítására a műhold és a földi állomás között.
- **REQ-COM-0002**: A modul legyen alkalmas QPSK *(Quadrature Phase-Shift Keying)*-modulált rádiójelek előállítására és vételére.
- **REQ-COM-0003**: A modul biztosítson 300 kSymbol/s maximális szimbólumsebességet.
- **REQ-COM-0004**: A szimbólumsebesség legyen futásidőben csökkenthető a jel–zaj viszony javítása érdekében.
- **REQ-COM-0005**: A modul egy általánosan ismert, kis- és nagyműholdak fedélzetén egyaránt rendelkezésre álló, soros kommunikációs interfészen keresztül legyen elérhető a fedélzeti rendszer központi számítógépe számára.
- **REQ-COM-0006**: A modul végezzen alapvető adatgyűjtést a rendszer állapotára, pl. a szükséges tápfeszültségek jelenlétére és megfelelőségére, a hot-spotok hőmérsékletére stb. vonatkozóan.

A fenti specifikációból kiindulva felrajzolható egy tisztán funkcionális rendszerterv *(concept design)*, aminek jellegzetessége, hogy nem leendő szerkezeti elemekből, hanem „funkciókból" épül fel (lásd 2-1. ábra).



2-1. ábra. *A kommunikációs modul funkcionális rendszerterve*

A fenti rendszerterv az alábbi funkciókat tartalmazza:

- **Serial host interface function**: A modul felhasználójával való kapcsolatért felelős funkcionális elem.
- **HK DAQ** *(Housekeeping Data Acquisition)* **function**: A rendszer állapotának monitorozását, feszültségszintek és hőmérsékletek mérését végző funkcionális elem.
- **Configuration function**: A futásidejű paraméterek beállítását végző funkcionális elem.
- **IQ transceiver function**[7]: A QPSK modulációhoz szükséges IQ minták előállítását végző, és az IQ mintákból az eredeti információtartalmat visszaállító funkcionális elem.
- **AD/DA** *(Analog-Digital/Digital-Analog)* **function**: Az adó ágban a digitális IQ mintákat analóg jellé alakító, a vevő ágban az analóg IQ mintákat digitalizáló funkcionális elem.
- **Modem function**: Az adó ágban az alapsávi mintasorozat vivőfrekvenciára való felkeverését, a vevő ágban a vivőfrekvenciás jel alapsávra való lekeverését végző funkcionális elem.

2.1.1 Szoftverközpontú megoldás

Hardver/szoftver-particionálás. • A hardver/szoftver-szétválasztás során azt határozzuk meg, hogy az egyes funkciók közül melyek azok, amelyeket egy mikroprocesszoron futó szoftver valósít majd meg, és melyek azok, amelyekhez dedikált célhardvert fejlesztünk ki. Általában arra törekszünk, hogy a lehető legtöbb funkciót szoftverként valósítsuk meg, mert így a fejlesztési folyamat nagymértékben párhuzamosíthatóvá válik;

[7] A QPSK moduláció működését itt nem részletezzük. A rendszertervezés problémáinak megértéséhez elegendő azt tudni, hogy az IQ minták előállítása és feldolgozása egy összetett DSP funkció, amelynek számításiteljesítmény-igénye a szimbólumsebességtől függően meglehetősen nagy lehet.

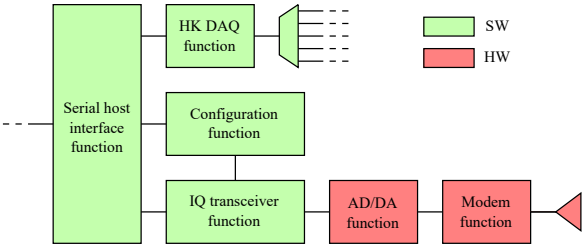
a szoftverkomponensek kifejlesztése és tesztelése a célhardverelemek kifejlesztésével párhuzamosan történhet. A hardverfejlesztés folyamata során tulajdonképpen az összes szoftverkomponens egyetlen hardverelemként, egy processzorként jelenik meg, amelynek hardver vonatkozásai – a tervezési szempontok, amelyeket a hardver (kapcsolási rajz és *Printed Circuit Board*, PCB rajzolat) kifejlesztése során figyelembe kell venni – teljesen függetlenek magától a funkciótól, tehát az időigényes hardvertervezés egyszerűsödik. A leggyakoribb probléma, ami korlátozza egy részfunkció szoftverben való megvalósíthatóságát, az a számításiteljesítmény-igény. Egy dedikált célhardver egy adott feladatot hatékonyabban – kevesebb idő alatt, kisebb átlagos fogyasztás mellett – tud megvalósítani, mint egy általános célú mikroprocesszor. Vegyük sorra a példarendszerünk funkcionális elemeit, és végezzük el a hardver/szoftver-particionálást:

- **Serial host interface function**: Erről a funkcióról kevés információ áll rendelkezésünkre a specifikációban, viszont a követelmények alapján kikövetkeztethetjük, hogy várhatóan lesz-e szükség célhardverre a megvalósítás során. A modul szimbólumsebessége 300 kSymbol/s-ban van maximalizálva, ami azt jelenti, hogy ennél sokkal nagyobb adatátviteli sebességet a felhasználói oldalon sem lesz érdemes megvalósítanunk. A felhasználói soros interfészen egy adatkapcsolati protokollt majd meg kell valósítanunk, ami a hasznos adatátviteli sebességet némileg csökkenteni fogja. Ezen a ponton persze nem ismerjük ezt a protokollt, hiszen a specifikáció nem írja elő, de egy nagyvonalú becslés alapján azt mondhatjuk, hogy 1 Mbit/s adatátviteli sebesség valószínűleg elegendő lesz. Ez a sebesség megvalósítható egy mikrokontroller belső SPI, vagy akár USART/UART interfészével is, célhardver kifejlesztésére nincs szükség, ezt a funkciót tehát megvalósíthatjuk szoftverben.
- **HK DAQ function**: A státuszinformáció összegyűjtése jellemzően a kis számításiteljesítmény-igényű feladatok közé tartozik, ráadásul a mikrokontrollerek belső A/D-átalakítói tökéletesen alkalmasak a feladatra, úgyhogy ezt a funkciót is megvalósíthatjuk szoftverben.
- **Configuration function**: A futásidejű paraméterek beállítása szintén nem igényel nagy számítási kapacitást, hiszen lényegében csak a konfigurálandó funkció vezérlőbitjeinek megfelelő beállításáról van szó, amihez nyilvánvalóan nem szükséges célhardver, tehát ezt is szoftveresen fogjuk megoldani.
- **IQ transceiver function**: Az IQ minták előállítása és feldolgozása már érdekesebb lehet hardver/szoftver-szétválasztás szempontjából. Az ehhez hasonló DSP funkciók számításiteljesítmény-igénye tipikusan a szimbólumsebesség és az alkalmazott modulációs eljárás egyéb paramétereitől függ. A számunkra előírt maximális szimbólumsebesség 300 kSymbol/s, ami – viszonylag egyszerű modulációs sémát feltételezve – egy DSP processzorral még kezelhető, döntsünk tehát ezúttal is a szoftveres megvalósítás mellett.
- **AD/DA function**: A specifikáció csak a szimbólumsebességet írja elő, az A/D- és D/A-átalakítás szempontjából viszont a mintavételi frekvencia a kérdéses paraméter. Minél több mintát veszünk egy szimbólumból, annál jobb jel–zaj-viszonyt érhetünk el, esetünkben tehát a mintavételi frekvencia mindenképpen a Mbit/s,

vagy inkább a 10 Mbit/s tartományba kell hogy essen. Bár a Mbit/s-os mintavételi frekvencia DSP mikrokontrollereknél egyáltalán nem elérhetetlen, helyezkedjünk ebben a kérdésben egy biztonságosabb álláspontra, és bízzuk az A/D- és D/A-átalakítást egy kifejezetten gyors, dedikált hardverelemre. Annál is inkább, mivel hardvertervezési szempontból ez a döntés nem jelent túl nagy problémát, hiszen csupán egyetlen komponens, egy nagysebességű AD/DA-konverter beépítését igényli.

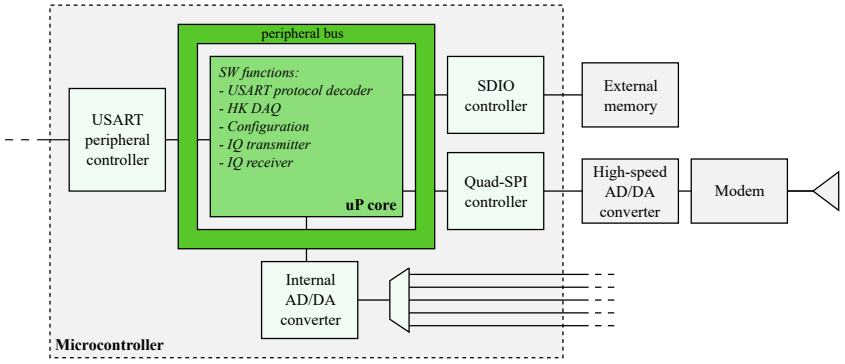
- **Modem function**: A moduláció és demoduláció az S-sávban (2-4 GHz) analóg áramköri elemekkel megvalósítandó feladat, tehát hardver/szoftver-szétválasztásról beszélni ez esetben nincs értelme. A modem funkciót egyértelműen dedikált hardverrel oldjuk meg.

A fenti döntések meghozása után tehát előáll egy olyan, részletesebb funkcionális rendszerterv, amely már a hardver/szoftver-particionálásra vonatkozó megvalósítási részleteket is tartalmazza (lásd 2-2. ábra).



2-2. ábra. *Hardver/szoftver-particionálás a kommunikációs rendszer esetén*

Megvalósítási (implementációs) technológia hozzárendelése a funkciókhoz • A hardver/szoftver-particionálást is tartalmazó funkcionális rendszertervből már előállítható egy még részletesebb, a kommunikációs interfészek és funkcionális elemek alapjául szolgáló konkrét technológiákat is tükröző rendszerterv (lásd 2-3. ábra).



2-3. ábra. *Funkcionális elemek és interfészek implementációs technológiáit leíró rendszerterv*

A hardver/szoftver-szétválasztás elvégzésével már önmagában is csökkent a modellünk elvonatkoztatási szintje, hiszen immár ismertek olyan megvalósítási részletek is, mint hogy milyen hardverelemekre lesz szükség a rendszer felépítéséhez; mikrokontroller, memória, A/D- és D/A-átalakító és a rádiófrekvenciás analóg áramköri blokkok. Az új rendszerterv-ről az is leolvasható, hogy a felhasználói interfész alapja az USART protokoll lesz, a szoftver tárolására SDIO *(Secure Digital Input Output)* interfészen hozzáférhető, külső memóriát fogunk alkalmazni, a nagysebességű A/D- és D/A-konverziót pedig egy Quad-SPI interfésszel rendelkező komponens fogja megvalósítani.

A specifikációban előírt funkcionalitást tehát részfeladatokra bontottuk, elvégeztük a hardver/szoftver particionálást és meghatároztuk az egyes rendszerkomponensek közötti kommunikációs interfészek jellegét. Az eredményül előálló modell tehát már a részletes tervezés alapjául szolgáló, teljes rendszertervnek tekinthető.

2.1.2 Hardverközpontú megoldás

Tegyük fel, hogy a kommunikációs modulunk követelménylistájában két „apró” módosítást végez a megrendelő a fejlesztési folyamatnak viszonylag korai fázisában[8]:

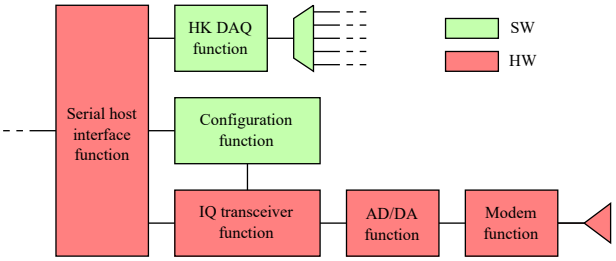
- **REQ-COM-0003**: A modul biztosítson ~~300 kSymbol/s~~ **10 MSymbol/s** maximális szimbólumsebességet.
- **REQ-COM-0005**: A modul ~~egy általánosan ismert, kis- és nagyműholdak fedélzetén egyaránt elérhető, soros kommunikációs~~ **SpaceWire** interfészen keresztül legyen elérhető a fedélzeti rendszer központi számítógépe számára. **A SpaceWire interfész bitsebessége 25 Mbit/s legyen.**

Vizsgáljuk meg, hogy a fenti módosítások befolyásolják-e a hardver/szoftver-particionálás során korábban meghozott döntéseinket, szükség van-e a rendszerterv módosítására.

- A SpaceWire interfész nagyműholdas rendszerekben széleskörűen alkalmazott megoldás, a mikrokontrollerek azonban tipikusan nem tartalmaznak ilyen interfészáramkört, ami azt jelenti, hogy ezt a funkciót célhardver segítségével, egy SpaceWire CODEC IC-vel tudjuk megoldani, amit valamilyen párhuzamos interfészen kapcsolhatunk a központi mikrokontrollerhez.
- A 10 MSymbol/s szimbólumsebesség sajnos nagyobb gondot jelent, ez ugyanis egy processzoron futó szoftverrel aligha kivitelezhető, célhardver pedig – különösen futásidőben konfigurálható kivitelben – nem érhető el például egy *„off-the-shelf”* IC formájában, úgyhogy nem csak hogy át kell térnünk a hardveres megvalósításra, de ezt a hardvert ki is kell fejlesztenünk és implementálnunk valamilyen programozható logikai eszköz – például egy FPGA – segítségével.

[8] Az itt bemutatott példa szándékosan sarkított. A követelményspecifikáció ilyen szintű megváltozása nem nevezhető tipikusnak, de azért elképzelhetetlennek sem.

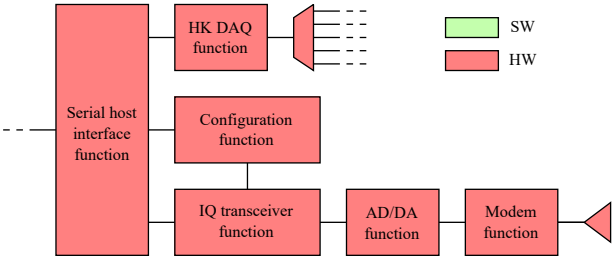
Ezen a ponton tehát úgy tűnik, hogy az eredeti rendszertervünk nem fog megfelelni az új követelményeknek, a korábban szoftveres megvalósításra predesztinált soros interfész-funkció és a QPSK modulációhoz szükséges IQ mintafeldolgozás funkció egyaránt hardverben kell hogy megvalósuljon (lásd 2-4. ábra).



2-4. ábra. *Hardver/szoftver-particionálás az új követelmények fényében – 1*

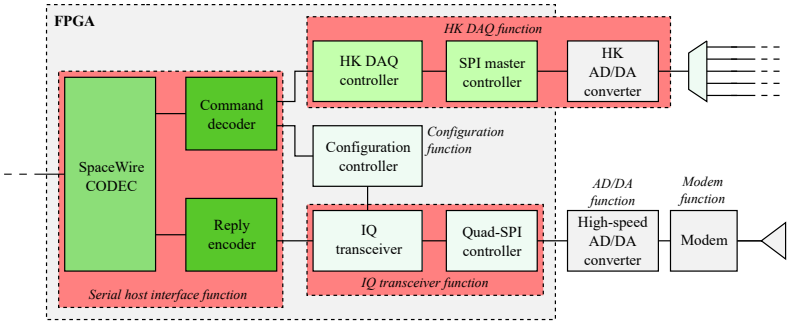
Rendszertervünk várhatóan egy új IC-vel, a SpaceWire CODEC IC-vel és egy FPGA-val fog bővülni, mielőtt azonban elkezdenénk ezt a bővített rendszertervet megrajzolni, álljunk meg egy pillanatra, és gondoljuk át, nincs-e valami kevésbé fájdalmas lehetőség.

Azzal, hogy egy FPGA-t is bevontunk a játszmába, immár rendelkezésünkre áll egy, a mikroprocesszorhoz hasonlóan általános célú eszköz, amelyben tetszőleges digitális funkciót – beleértve a SpaceWire CODEC IC teljes funkcióját – meg tudunk valósítani, bár kétségtelenül igen eltérő módon, mint egy processzor esetén. Immár csak a HK adatgyűjtés és a futásidejű konfigurálást lehetővé tévő funkcióhoz rendeltünk szoftveres megvalósítást. Azonban e két funkció igen egyszerűen kezelhető hardverben is, ha egyéb okokból kifolyólag a rendszerben amúgy is szerepel egy olyan eszköz, nevezetesen az FPGA, amely alkalmas arra, hogy e funkciókat megvalósítsa. Ezt felismerve tehát előáll egy új funkcionális rendszerterv (lásd 2-5. ábra), amelyben immár minden részfunkciót célhardverként írunk le.



2-5. ábra. *Hardver/szoftver-particionálás az új követelmények fényében – 2*

Ebben a megoldásban tehát nincs szoftverkomponens, tehát mikroprocesszorra sincs szükség, ehelyett minden részfeladatot valamilyen dedikált célhardver fog megvalósítani (lásd 2-6. ábra).



2-6. ábra. *Funkcionális elemek és interfészek implementációs technológiáit leíró rendszerterv az új követelmények fényében*

A teljesen hardverközpontú megoldás hátrányának tűnhet, hogy a benne található részfunkciókat megvalósító áramköröket meg kell tervezni, ami időigényes, hibalehetőségekkel terhelt, specifikus szaktudást igénylő folyamat (a jegyzet további fejezetei pontosan ezt a folyamatot tárgyalják részletesen). A rendszertervezés gyakorlata során azonban, amikor a műszaki szempontokon kívül egyéb, például a fejlesztési időre, költségre és emberi erőforrás rendelkezésre állására vonatkozó szempontokat is figyelembe kell vennünk, nem szabad megfeledkeznünk az IP-alapú tervezés korábban említett jellegzetességeiről; a tervezés során nem egy üres papírlappal indulunk, az egyes részfunkciókat megvalósító áramkörök különböző módokon már rendelkezésre állhatnak. A fenti példában a SpaceWire CODEC, az egyszerű SPI és Quad-SPI vezérlők szabványos protokollokat valósítanak meg, ami azt jelenti, hogy ezek az áramkörök tulajdonképpen már rendelkezésre állnak, előre megtervezett és validált komponensként felhasználhatók, ami igencsak lerövidítheti a fejlesztés folyamatát.[9]

A 2-6. ábrán látható rendszerterv már kellő részletességű ahhoz, hogy a részletes tervezés alapját képezze, tehát – a 2-3. ábrán láthatóhoz hasonlóan – teljes értékű rendszertervnek tekinthető.

Mindkét, itt bemutatott rendszertervre igaz, hogy továbbfejleszthető, még több megvalósítási részlet hozzáadásával az elvonatkoztatás mértéke tovább csökkenthető. Korábbi tapasztalat alapján meghatározható a felhasznált eszközök – a mikroprocesszor és az FPGA, az AD/DA-átalakítók – pontos típusa, megbecsülhető a szoftveres megoldásban szükséges külső memória mérete vagy az FPGA kapacitása. Ugyancsak tekinthető rendszerszintű tervezési feladatnak a felhasználói interfész adatkapcsolati protokolljának megtervezése, bár ezt már sorolhatjuk a vonatkozó részfunkció részletes tervezésének kérdései közé is. Bár a rendszerterv rengeteg információt tartalmazhat a megvalósítás részleteire vonatkozóan, az egyes részfunkciók pontos viselkedését nem rögzíti formálisan, ami komoly korlátot jelent a teljes rendszer viselkedésének vizsgálata során. Ahhoz, hogy az egyes részfunkciók együttműködését részletesen vizsgálhassuk, olyan modellekre van szükség, amelyek a viselkedést teljesen egyértelműen, formális módon rögzítik olyan formában, hogy a rendszer

[9] A fejlesztési idő és költség vizsgálatánál persze figyelembe kell venni, hogy ezek az újrafelhasználható IP magok – ahogy arra az *„Intellectual Property"* kifejezés is utal, nem tekinthetők közkincsnek. Ezek az előre tervezett áramkörök – különösen összetett funkció esetén – igencsak drágák lehetnek.

működése gépi úton szimulálható legyen. Ezeket a modelleket viselkedési modelleknek nevezzük, megvalósításukhoz pedig rendszerint magas szintű programozási nyelveket használunk.



3

Viselkedési modellezés

A viselkedési modellezés elsődleges célja a rendszert alkotó almodulok funkciójának formális rögzítése. A modellezésnek ezen a szintjén a formális jelleget, tehát az egyértelműséget gépi nyelvek képesek csak biztosítani, a specifikáció természetes nyelvű leírásaival és az elvontabb, rendszerszintű, kötetlen formátumú modellekkel ellentétben. Digitális funkció rögzítésére a legelterjedtebben használt gépi nyelvek a MATLAB parancsfájlok, a SystemVerilog hardverleíró nyelv újabb változatai,[10] illetve a C, C++, Java és Python programozási nyelvek önmagukban, vagy egyedi osztálykönyvtárakkal (pl. SystemC, SystemC TLM) kiegészítve, amelyek e speciális alkalmazási területen még hatékonyabbá teszik magát a nyelvet.

Ebben a fejezetben azt mutatjuk be, hogy a szabványos C++ nyelvet önmagában miként lehet speciálisan digitális rendszermodellezési célra használni. Az alkalmazott C++ nyelvi szerkezetek természetesen jól ismertek a szoftverfejlesztésben járatos olvasók számára, e fejezet célja annak bemutatása, hogy ezeknek az ismert nyelvi szerkezeteknek a segítségével hogyan készíthetők olyan leírások, amelyeknek bár elsődleges célja a funkció, a viselkedés specifikálása, mégis a lehető legjobban „érzékeltetik", hogy valójában áramköri modellekről van szó, ezáltal könnyítve meg az ezeket a leírásokat referenciának, specifikációnak tekintő RTL tervezők munkáját.

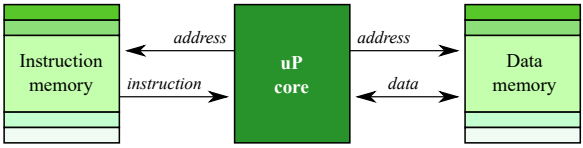
Az egyes részfunkciókat megvalósító almodulok, részáramkörök mindegyikére igaz, hogy jól definiált határfelülettel, interfésszel rendelkeznek, továbbá – a bennük lévő, egyedileg összekapcsolt erőforrások segítségével – valamilyen speciális feladat elvégzésére alkalmasak. A rendszerszinten is igen hasznos „külső interfész + belső működés" szétválasztás az objektumorientált szemlélet alkalmazását a viselkedési szintű modellezésben gyakorlatilag elkerülhetetlenné teszi. A különálló részfunkciókat ellátó almodulok leképezése a megfelelő C++ osztály egy-egy példányára már-már magától értetődőnek tűnik, és – ahogy az itt bemutatott példák alapján is látható lesz – valóban kiválóan használható absztrakció, olyan jól áttekinthető viselkedési szintű modellekhez vezet, amelyek – algoritmikus jellegük ellenére – az elképzelt rendszer szerkezetét is meglehetősen részletesen vissza tudják adni.

[10] Általános célú modellező nyelv lévén a VHDL is alkalmas viselkedési szintű modellezésre, de a mai mérnöki gyakorlatban a nyelvnek ezt a képességét inkább csak a funkcionális verifikáció során, az áramköri környezetet modellező verifikációs komponensek viselkedésének megfogalmazásakor használjuk ki (lásd 5. fejezet).

3.1 A MiniCore mikroprocesszor viselkedési szintű modellje

Az első példa egy nagyon egyszerű utasításkészletet megvalósító, harvard-architektúrájú mikroprocesszornak egy lehetséges viselkedési szintű modelljét mutatja be. Fontos hangsúlyozni, hogy az itt bemutatott megoldás csak egy a rengeteg lehetséges megközelítés közül, ahogyan egy ilyen rendszer viselkedését le lehet írni. E példarendszer bemutatásával – ahogy a következő alfejezetben bemutatott másik példa esetén is – célunk az, hogy rajta keresztül megmutassuk, hogy milyen módszerekkel lehet már egy algoritmusszintű modellben is utalni a szándékolt hardvermegvalósítás részleteire, hogyan lehet a viselkedési szintű modelleket „hardveres szemlélettel" elkészíteni.

Tekintve, hogy a viselkedési szintű modellezés során a specifikáció szerepét a rendszerterv tölti be, induljunk ki itt is ebből. A 3-1. ábra – a mikroprocesszor rendszertervének részeként – egy vázlatos blokkdiagramot mutat be, ami tulajdonképpen csak a rendszer fő komponenseit szemlélteti.



3-1. ábra. *Az egyszerű mikroprocesszor rendszerterve*

A rendszer tehát három fő összetevőből áll, a rendszerterv azonban egyéb tulajdonságait is megfogalmazza a leendő áramkörnek, amelyeket ugyancsak informális módszerrel, egy természetes nyelvű specifikációhoz hasonló módon, szöveges formában rögzítünk:

- **Utasításmemória**: A gépi utasításokat tároló, 1024 bejegyzés tárolására alkalmas memória.
- **Adatmemória**: Általános célú, 1024 bejegyzés tárolására alkalmas memória.
- **Processzormag**: A gépi utasításokat az utasításmemóriából kiolvasó és azokat végrehajtó almodul, amely képes az adatmemóriába adatot írni és onnan adatot kiolvasni. A processzormag 3-címes utasításkészletet valósít meg, a köztes adatok eltárolása céljából tartalmaz egy 8 bejegyzés méretű belső regisztertömböt. Két állapotjelzőt (*flag*-et) tart számon az utasítások végrehajtása során, amelyeket a feltételes ugró utasítások használnak fel:
    - Zero (Z): Jelzi, ha a legutóbb végrehajtott utasítás eredménye nulla.
    - Negatív (N): Jelzi, ha a legutóbb végrehajtott utasítás eredménye negatív.

Ugyancsak a rendszerterv része magának a megvalósított utasításkészletnek a definíciója, amelyet a 3-1. táblázat tartalmaz.

Az utasításkészlet definíciójának a jelölései a következők:

- **RF** *(Register File)*: A 8 bejegyzés tárolására képes, belső regisztertömb.
- **DM** *(Data Memory)*: Adatmemória.

Mnemonik	Gépi utasítás funkciója	Állapotjelzők
loadimm	RF[D] = IMM	Z, N
memread	RF[D] = DM[RF[A] + IMM]	Z, N
memwrite	DM[RF[D] + IMM] = RF[A]	-
add	RF[D] = RF[A] + RF[B]	Z, N
sub	RF[D] = RF[A] - RF[B]	Z, N
mul	RF[D], RF[D+1] = RF[A] * RF[B]	-
div	RF[D] = RF[A] / RF[B] RF[D+1] = RF[A] mod RF[B]	-
and	RF[D] = RF[A] & RF[B]	Z, N
or	RF[D] = RF[A] | RF[B]	Z, N
xor	RF[D] = RF[A] ^ RF[B]	Z, N
shl	RF[D] = RF[A] >> RF[B]	Z, N
shr	RF[D] = RF[A] << RF[B]	Z, N
jump	IP = IMM	-
branchzero	if (Z == 1) then IP = IMM else IP = IP + 1	-
branchnegative	if (N == 1) then IP = IMM else IP = IP + 1	-
stop	stop the execution	-

3-1. táblázat. *A MiniCore mikroprocesszor utasításkészlete*

- **A**: Az A operandus regisztertömb-indexe.
- **B**: A B operandus regisztertömb-indexe.
- **D** *(Destination)*: Az eredményt tároló regiszter indexe.
- **IMM** *(Immediate)*: Az utasításban közvetlenül megadott konstans.
- **IP** *(Instruction Pointer)*: Utasításmutató, a végrehajtás alatt álló gépi utasítás programtárbeli címe.
- []: Indexelés, pl.: RF[A]: a regisztertömb A. eleme.

A fenti információk összessége jelenti a rendszertervet, ezekből kiindulva kell tehát egy olyan modellt készítenünk, amely képes leírni a processzor viselkedését, működését.

Az újrafelhasználhatóságra törekvés a tervezésnek minden szintjén kifizetődő, ezért a viselkedési szintű modell elkészítése során is igyekszünk olyan tervezési egységeket létrehozni, amelyeket a modell több pontján is fel tudunk majd használni. Ebben a példában a memória funkció az, ami többször is szerepel; az utasításmemória gépi utasításokat, az adatmemória és a processzormag belső regisztertömbje pedig adatokat tárol. Bár adattípus és méret tekintetében e három almodul különbözik, de az alapfunkciójuk ugyanaz, ami lehetőséget ad arra, hogy egy általánosított, ún. sablonosztály segítségével írjuk le működésüket (lásd 3-2. ábra).

Vizsgáljuk meg az ábrán látható memóriaosztályt mint modellt, vagyis tekintsük át, hogy egy memóriának mint áramkörnek melyek azok a tulajdonságai, amelyeket ez a leírás visszaad, és melyek azok, amelyekről semmilyen részletet nem fogalmaz meg. A fenti modell egy valós memória-áramkörnek azt a képességét írja le, hogy adott mennyiségű és típusú bejegyzés tárolására képes. Minden egyes, a memóriában tárolt bejegyzést a címe egyértelműen azonosít. Megfogalmazza továbbá azt is, hogy a memória felhasználója két,

C++

```
template < unsigned size, typename item_type >
class memory {

    private: item_type content[size];

    public:

        const item_type& read(unsigned address) const {
            if ( address < size ) {
                return content[address];
            } else throw ("Memory access error: invalid read address");
        }

        void write(const unsigned address, const item_type& data_in) {
            if ( address < size ) {
                content[address] = data_in;
            } else throw ("Memory access error: invalid write address");
        }

};
```

3-2. ábra. *Általánosított memória-sablonosztály*

egymástól független műveletet végezhet a memórián; elhelyezhet egy bejegyzést egy adott címre, vagy kiolvashat egy korábban elhelyezett bejegyzést egy megadott címről. Vegyük észre, hogy ez a modell valóban „csak” egy viselkedési modell; leírja például, hogy a kiolvasás során a felhasználó által megadott címen lévő bejegyzést kapjuk vissza, de semmit nem mond arról, hogy ez fizikailag miként valósul meg. A kiolvasás folyamata egy valós áramkörben – természetesen a konkrét memóriatípustól függően – nagyon bonyolult, egy sor, megfelelően ütemezett és időzített vezérlő jelet igényel. Ezek azonban olyan tényezők, amelyeket viselkedési szinten nem célunk modellezni, ezek a részletek az alacsonyabb elvonatkoztatású modellek – tipikusan az RTL modellek – problémái lesznek.

Egy mikroprocesszorban a gépi utasításokat bitvektorokként – fizikailag valamilyen tároló elemek, például flip-flopok vagy SRAM cellák által megvalósított nulla-egy sorozatként – kell elképzelnünk, amelyeknek a különböző helyiértékein (mezőin) más-más információt tárolunk. Egy szokásos MIPS architektúrában például jellemzően a 32-bites utasítás

C++

```
class instruction {
  public:
    enum instruction_type { uninitialized, loadimm, memread, ... } instruction_type;
    instruction();
    instruction(const instruction_type& opcode,
                const unsigned& destination_index,
                const unsigned& op_a_index,
                const unsigned& op_b_index,
                const int& immediate);
    instruction(const instruction& src);

    // setter/getter functions...

    void operator = (const instruction& src);

  private:
    instruction_type opcode;
    unsigned destination_index;
    unsigned op_a_index;
    unsigned op_b_index;
    int immediate;

};
```

3-3. ábra. *Az absztrakt utasítás-osztály deklarációja*

C++

```
class processor {

  private:

    // internal resources
    instruction ir;
    unsigned ip;
    memory<8, int> rf;
    unsigned destination_index, op_a_index, op_b_index, mar, mdr;
    int immediate;
    bool zero_flag, negative_flag;
    bool run;

    // external resources
    const memory<1024, instruction>& program_memory;
    memory<1024, int>& data_memory;

    // private functions
    void fetch();
    void decode();
    void execute();

  public:

    processor(memory<1024, instruction>& program_memory, memory<1024, int>& data_memory);
    void run_program();

};
```

3-4. ábra. *A processzormagot leíró osztály deklarációja*

felső 6 bitje tartalmazza az utasítástípus (*add*, *sub*, *mov* stb.) egyedi azonosítóját, a közvetlen címzést *(immediate)* használó utasítások pedig az alsó 20-22 bitben tárolják az utasításban közvetlenül megadott adatot. Az itt bemutatott viselkedési szintű modellben nem célunk a gépi utasítások pontos szerkezetének modellezése, nem foglalkozunk tehát azzal, hogy az egyes utasítások melyik bitjei milyen információt kódolnak. Ezek olyan megvalósítási részletek, amelyeknek a kidolgozását az RTL tervezőre bízzuk. A viselkedési szintű modellünk ehelyett egy elvont utasításfogalommal fog dolgozni, amely csak azt a tényt képes visszaadni, hogy mindezeket az információkat a gépi utasítások tartalmazzák, de azt nem, hogy hogyan. Ennek megfelelően a gépi utasításokat is egy megfelelően felépített osztály példányai fogják képviselni. A 3-3. ábrán ennek az absztrakt utasítás-osztálynak a deklarációja látható.

Ehhez az absztrakt utasítás-osztályhoz nem készítünk olyan tagfüggvényeket, amelyek bármi módosítást végeznének a tagváltozókon, segítségével lényegében csak egységbe foglalunk egy sor, a modellünkben összetartozónak számító adatot.

A következő tervezési egység maga a processzormag, amelynek leírásához – a memóriákhoz hasonlóan – egy egyedi osztályt hozunk létre. Mint ahogy az objektumorientált modellekben minden osztály, úgy ez is azt fogja leírni, hogy az adott tervezési egységet, vagyis az osztály egy példányát milyen belső állapot jellemzi, illetve milyen műveletek végezhetők a belső állapotot leíró tagváltozókon, vagy – ahogy az objektumorientált modellezés elméletével foglalkozók szeretnek fogalmazni – milyen műveleteket tud elvégezni az adott objektum a saját belső állapotát leíró tagváltozókon.[11] A 3-4. ábra a processzormagot leíró osztály deklarációját mutatja be.

[11] Az objektumorientált szemlélet egy kulcsfontosságú eleme, hogy a belső állapoton módosítást csak az osztály publikus tagfüggvényein keresztül, ellenőrzött módon végezhetünk, ami az objektum adattagjai közötti konzisztencia fenntartásához elengedhetetlen (lásd például a memóriaosztály esetét, ahol a túlcímzés ellen maga a memóriaosztály védekezik). Ilyen értelemben valóban helyesebb azt mondani, hogy a belső állapotot az objektum maga változtatja meg.

Vegyük sorra, hogy a processzort leíró osztály milyen változók segítségével tartja számon egy processzorobjektum állapotát:

- `instruction ir` *(Instruction Register)*: A végrehajtás alatt álló gépi utasítást leíró utasításobjektum, a 3-3. ábrán látható osztály egy példánya.
- `unsigned ip` *(Instruction Pointer):* A végrehajtás alatt álló gépi utasítás programtárbeli címét tároló változó.
- `memory<8, int> rf` *(Register File):* A rendszerterv által előírt, 8 bejegyzés tárolására alkalmas, belső regisztertömb. A 3-2. ábrán bemutatott memória-sablonosztály egy példánya.
- Az aktuális gépi utasítás dekódolásának eredményeképpen előálló belső változók:
  - `unsigned destination_index`: A végrehajtás alatt álló utasítás eredményét tároló rekesz címe a belső regisztertömbben.
  - `unsigned op_a_index, op_b_index`: A végrehajtás alatt álló gépi utasítás operandusainak címe a belső regisztertömbben.
  - `unsigned mar, mdr`: Memóriareferens utasítások esetén a külső memóriát címző változó (`mar`) és memóriaírás esetén a külső memóriába kiírandó adatot tároló változó (`mdr`).
  - `int immediate`: Az utasításban közvetlenül megadott konstanst tároló változó.
  - `bool zero_flag, negative_flag`: A legutóbb végrehajtott utasítás eredményétől függően beállított Z és N állapotjelzőket tároló kétállapotú változók.
  - `bool run`: Kétállapotú változó, amely igaz értékű, amikor a processzor programot hajt végre, hamis egyébként. E változó a felhasználó által kezdeményezett programvégrehajtás első lépéseként automatikusan igaz értéket vesz fel, majd a *stop* utasítás végrehajtásakor értéke ismét hamis lesz, ami egyúttal a programvégrehajtás végét is jelenti.
- `const memory<1024, instruction>& program_memory`: Az utasításmemóriára mutató referencia.
- `memory<1024, int>& data_memory`: Az adatmemóriára mutató referencia.

A 3-4. ábrán látható kódrészlettel kapcsolatban az első tisztázandó dolog a belső és a külső erőforrások megkülönböztetésének létjogosultsága. Az osztály deklarációját a 3-1. ábra blokkvázlatával összevetve nyilvánvaló, hogy azokat a dolgokat nevezzük belső erőforrásnak, amelyek a blokkdiagramon középen szereplő almodulnak részei (így az ilyen részletek leírásához nem kellően kidolgozott blokkdiagramon tulajdonképpen nem is látszanak), míg külső erőforrásnak a blokkdiagramon szereplő, a processzormaghoz kívülről – egyelőre nem specifikált interfészen – kapcsolódó almodulokat nevezzük. Ez a megkülönböztetés tehát tulajdonképpen egy olyan modellezési megoldás, amely a leendő áramköri szerkezetre utal; három különálló áramkörről beszélünk, az utasításmemóriáról, az adatmemóriáról és a processzormagról. Ez utóbbi az előző kettőt, mint külső erőforrást fogja használni működése során. A külső és belső erőforrások megkülönböztetése ebben az esetben technikailag úgy valósul meg, hogy míg a belső erőforrásokat egyszerű tagváltozók képviselik,

addig a külső erőforrásokat egy-egy referenciával tartjuk számon.[12] Érdekes megfigyelni, hogy míg az utasításmemóriára konstans referenciával hivatkozunk, tehát a processzor nem fogja tudni megváltoztatni a programtár tartalmát, addig az adatmemóriára ez nem igaz, hiszen a processzor írni is tud a memóriába. Ezzel az apró különbséggel formálisan is modelleztük az informális blokkvázlatnak azt a tulajdonságát, hogy az utasításmemóriát a processzorral összekötő adatbusz csak egyirányú, míg az adatmemória esetén ugyanez a busz kétirányú.

A processzor-osztály három privát tagfüggvényt tartalmaz, amelyek a processzorok utasítás-feldolgozásának jól ismert részfolyamatait hivatottak leírni:

- `fetch()`: A soron következő gépi utasítást kiolvassuk a programtárból és betöltjük az utasításregiszterbe.
- `decode()`: A beolvasott gépi utasítást dekódoljuk, vagyis az utasításregiszter különböző mezőit áttöltjük más, speciális célú belső regiszterekbe, például a belső regisztertömb címzésére használt regiszterekbe, vagy a külső memóriát címző regiszterbe.
- `execute()`: Az előkészített, speciális célú regisztereket felhasználva végrehajtjuk az utasításkód által kijelölt feladatot.

A processzorosztály publikus interfésze – a konstruktoron kívül – lényegében egyetlen tagfüggvényt tartalmaz (`run_program()`), amelynek segítségével az osztály felhasználója a processzorobjektumot utasíthatja a hozzá kapcsolt utasításmemóriában található program futtatására. Ahogy azt később látni fogjuk, ez tulajdonképpen nem más, mint egy végtelen ciklus, amelyben az utasításfelhozás–dekódolás–végrehajtás lépés-hármas ismétlődik.

Az osztály interfészének áttekintése után vizsgáljuk meg annak implementációját is, hiszen magának a viselkedésnek a formális rögzítése a tagfüggvények definíciójában valósul meg. Az utasításfelhozás folyamatát a `fetch()` tagfüggvény definíciója fogalmazza meg (lásd 3-5. ábra).

C++

```
void processor::fetch() {
    ir = program_memory.read(ip);
}
```

3-5. ábra. *Az utasításfelhozást leíró tagfüggvény definíciója*

Az utasításfelhozás tehát tulajdonképpen egy, az utasításmemórián végrehajtott olvasási művelet, amelyhez a kiolvasandó bejegyzés címét a processzor utasításmutató regisztere adja. A 3-6. ábrán a dekódolás műveletet leíró tagfüggvény implementációja látható.

[12] A referencia használata azt vonja maga után, hogy egy-egy processzor-példány létrehozásakor a hozzá kapcsolandó memória-példányoknak már léteznüük kell, hiszen a referencia-tagváltozókat már a processzorosztály konstruktorában inicializálnunk kell. Ez a kötöttség szükség esetén kiküszöbölhető, ha a referenciák helyett pointerekkel hivatkozunk a külső erőforrásokra, modellezési szempontból azonban e két megoldás ekvivalensnek tekinthető.

C++

```
void processor::decode() {
    destination_index = ir.get_destination_index();
    op_a_index = ir.get_op_a_index();
    op_b_index = ir.get_op_b_index();
    immediate = ir.get_immediate();
    if ( ir.get_opcode() == instruction::memread ) mar = rf.read(ir.get_op_a_index()) + ir.get_immediate();
    else mar = rf.read(ir.get_destination_index()) + ir.get_immediate();
    mdr = rf.read(ir.get_op_a_index());
}
```

3-6. ábra. *Az utasításdekódolást leíró tagfüggvény definíciója*

A dekódolás lépésében az utasításregiszter tartalmát „értelmezzük”; az egyes mezőinek tartalmát átmásoljuk a processzormagnak specifikusan az adott mezőben található információ tárolására allokált belső változójába. Az is megfigyelhető, hogy a memóriareferens utasításokhoz tartozó regiszterek (`mar, mdr`) esetén az ún. eltolt regiszter–indirekt címzés miatt nemcsak egyszerű másolást értünk dekódolás alatt, hanem regisztertömb-olvasást és aritmetikai műveletet is kell végeznünk.

A legterjedelmesebb definíció ebben a modellben az utasítások végrehajtását leíró `execute()` függvényhez tartozik (lásd 3-7. ábra).

C++

```
void processor::execute() {
    switch ( ir.get_opcode() ) {
        case instruction::loadimm:        /* loadimm behavior        */ break;
        case instruction::memread:        /* memread behavior        */ break;
        case instruction::memwrite:       /* memwrite behavior       */ break;
        case instruction::add:            /* add behavior            */ break;
        case instruction::sub:            /* sub behavior            */ break;
        ...
        case instruction::jump:           /* jump behavior           */ break;
        case instruction::branchzero:     /* branchzero behavior     */ break;
        case instruction::branchnegative: /* branchnegative behavior */ break;
        case instruction::stop:           /* stop behavior           */ break;
    }
}
```

3-7. ábra. *Az utasításvégrehajtást leíró tagfüggvény definíciója (részlet)*

Az utasításvégrehajtási fázisban az összes gépi utasítás viselkedését meg kell fogalmaznunk. Azt, hogy egy konkrét végrehajtási ciklusban melyik funkció végrehajtása zajlik, az utasításkódtól függ. Nem meglepő tehát, hogy a végrehajtás fázist leíró tagfüggvény tulajdonképpen egyetlen C++ utasítás, egy switch-case szerkezet segítségével valósul meg. A 3-8., 3-9. és 3-10. ábrák rendre a memóriaolvasás, az összeadás és az egyik feltételes ugró utasítás végrehajtásának megvalósítását mutatják be.

C++

```
...
case instruction::memread:
    rf.write(destination_index, data_memory.read(mar));
    if ( rf.read(destination_index) == 0 ) zero_flag = true; else zero_flag = false;
    if ( rf.read(destination_index) < 0 ) negative_flag = true; else negative_flag = false;
    ++ip;
    break;
...
```

3-8. ábra. *A memóriaolvasó utasítás végrehajtását leíró kódrészlet*

C++

```
...
case instruction::add:
    rf.write(destination_index, rf.read(op_a_index) + rf.read(op_b_index));
    if ( rf.read(destination_index) == 0 ) zero_flag = true; else zero_flag = false;
    if ( rf.read(destination_index) < 0 ) negative_flag = true; else negative_flag = false;
    ++ip;
    break;
...
```

3-9. ábra. *Az összeadás utasítás végrehajtását leíró kódrészlet*

C++

```
...
case instruction::branchzero:
    if ( zero_flag ) ip = immediate;
    else ++ip;
    break;
...
```

3-10. ábra. *Egy feltételes ugró utasítás végrehajtását leíró kódrészlet*

Immár adott a processzor belső működésének három részfolyamatát leíró modell, most már csak azt a jellegzetességet kell kifejeznünk valamilyen módon, hogy ezek a részfolyamatok egy program végrehajtása során a megfelelő sorrendben egymást követően, ciklikusan mennek végbe. A 3-11. ábra a processzor-osztály publikus interfészének részét képező, tehát a felhasználó által hívható `run_program()` tagfüggvény definícióját mutatja.

C++

```
void processor::run_program() {

    run = true;

    while (run) {
        fetch();
        decode();
        execute();
    }

}
```

3-11. ábra. *A programfuttatást leíró tagfüggvény definíciója*

A `run_program()` tagfüggvény megfelelően sorba rendezi és ciklusba szervezi a három részfolyamatot.

A viselkedési szintű modellezés egyik legfontosabb tulajdonsága, hogy a modellezett rendszer működése gépi úton szimulálható. Esetünkben ez azt jelenti, hogy a fenti osztályok használatával készíthetünk egy programot, amelynek lefuttatása tulajdonképpen a modell viselkedési szimulációját jelenti. Ahhoz, hogy ezt elvégezhessük, még két dolog hiányzik; elő kell állítanunk egy utasításokat tartalmazó programtárat, és létre kell hoznunk egy tesztkörnyezetet, amelyben a modell részegységei összekapcsolhatók és a programfuttatás elindítható.

A programtár a futtatandó utasításokat tartalmazó memória, amit a 3-2. ábrán látható memória-sablonosztály egy példánya valósít meg, amelyben a tárolt bejegyzések a 3-3. ábrán látható utasításosztály példányai. Közvetlen módon tehát egy futtatható programot úgy hozhatunk létre, hogy a kívánt utasításokat az utasításosztály konstruktora segítségével létrehozzuk, majd a memóriaosztály írás metódusának segítségével beírjuk őket egy konkrét memóriapéldányba. Az e feladatot ellátó segédfüggvény látható a 3-12. ábrán.

C++

```
void create_testprogram_1(memory<1024, instruction>& pmem) {

  pmem.write(0, instruction(instruction::loadimm,7,0,0,7));  // loadimm 7,7          # rf[7] <= 7
  pmem.write(1, instruction(instruction::loadimm,6,0,0,0));  // loadimm 6,0          # rf[6] <= 0
  pmem.write(2, instruction(instruction::loadimm,5,0,0,0));  // loadimm 5,0          # rf[5] <= 0
  pmem.write(3, instruction(instruction::loadimm,4,0,0,1));  // loadimm 4,1          # rf[4] <= 1
  pmem.write(4, instruction(instruction::memread,6,7,0,12)); // loop: memread 6,7,12 # rf[6] <=
                                                                                     # dm[rf[7]+12]]
  pmem.write(5, instruction(instruction::add,5,5,6,0));      // add     5,5,6        # rf[5] <= rf[5]+rf[6]
  pmem.write(6, instruction(instruction::sub,7,7,4,0));      // sub     7,7,4        # rf[7] <= rf[7]-rf[4]
  pmem.write(7, instruction(instruction::branchnegative,0,0,0,9)); // bn      9       # if (n) goto exit
  pmem.write(8, instruction(instruction::jump,0,0,0,4));         // jmp     4       # goto loop
  pmem.write(9, instruction(instruction::memwrite,0,5,0,0)); // exit: memwrite 0,5,0 # dm[rf[0]] <= rf[5]
  pmem.write(10, instruction(instruction::stop,0,0,0,0));    // stop                 # stop execution

}
```

3-12. ábra. *A processzormodellen futtatandó program beírása az utasításmemóriába*

Ez a módszer a futtatható programok létrehozására igencsak körülményesnek tűnhet, ami az itt bemutatott modell példajellegének köszönhető. Részletesen kidolgozott viselkedési szintű processzormodellek – ún. utasításkészlet-szimulátorok – esetén a programtár inicializálására tipikusan egyedi assembly nyelvet definiálunk és az assembly nyelvű forráskód alapján az absztrakt utasításosztály példányait tartalmazó programtár-objektumot egy megfelelő algoritmussal hozzuk létre. Az itt látható tesztprogram funkciója viszonylag egyszerű; az adatmemória 12. és 19. rekeszei között elhelyezkedő adatokat összeadja, és az eredményt az adatmemória 0. rekeszébe tölti.

Ezek után nincs más hátra, mint elkészíteni egy tesztkörnyezetet, amelyben létrehozhatók a rendszer komponensei, megvalósíthatók a közöttük lévő kapcsolatok, és elindítható a modell szimulációja. A tesztkörnyezetet legegyszerűbben egy megfelelő `main()` függvény elkészítésével írhatjuk le, ahogy az a 3-13. ábrán is látható.

C++

```
int main() {

    // Instantiate a program memory and a data memory.
    memory<1024, instruction> pmem; create_testprogram_1(pmem);
    memory<1024, int> dmem;
    for ( unsigned i = 0; i < 256; ++i ) dmem.write(i, 9 * i);

    // Instantiate a processor.
    processor p(pmem, dmem);

    // Start program execution.
    p.run_program();

    // Check the result.
    int tmp = 0;
    int acc = 0;
    for ( unsigned i = 19; i >= 12; --i ) {
        acc += dmem.read(i);
    }

    tmp = dmem.read(0);
    if ( tmp == acc ) std::cout << "data_memory[0]: " << tmp << " -> CHECK OK" << std::endl;
    else std::cout << "data_memory[0]: " << tmp << " -> CHECK FAIL" << std::endl;

    return 0;

}
```

3-13. ábra. *A processzormodell tesztkörnyezete*

Ennek a tesztkörnyezetnek van egy, a funkcionális verifikáció szempontjából kulcsfontosságú tulajdonsága. Miután a processzormodellünk végrehajtotta a programtárba töltött tesztprogramot, a futás eredményének ellenőrzését maga a tesztkörnyezet végzi el úgy, hogy a programtárban lévő utasítássorozat által megvalósított algoritmust maga is végrehajtja. Gyakorlatilag az történik, hogy az egyszerű tesztprogramunkat nemcsak az általunk írt processzormodell hajtja végre, hanem a modellt futtató processzor is. Ha helyesen valósítottuk meg a processzor modelljét és a tesztprogramot is, akkor a modell és az azt futtató gép ugyanarra az eredményre jut, a teszt tehát sikeres, ellenkező esetben valami hiba történt. Az ilyen tesztkörnyezeteket, amelyek automatikusan meg tudják állapítani, hogy egy teszt sikeres volt, vagy sem, önellenőrző *(self-checking)* verifikációs vagy tesztkörnyezeteknek nevezzük. Az önellenőrző tesztkörnyezeteknek kiemelt jelentősége van az RTL modellek funkcionális verifikációja során alkalmazott, ún. automatizált regressziós tesztelési eljárásban (lásd 5.3. alfejezet).

A bemutatott kódrészletekből jól látszik, hogy miként írja le a processzorosztály egy mikroprocesszor viselkedését. Vizsgáljuk meg ezt az osztályt áramkörmodellezés szempontjából. Milyen megvalósítási részleteket rögzít ez a modell, és melyek azok, amelyeknek a kifejtését az alacsonyabb elvonatkoztatási szintű tervezési lépésekre bízza? Bár, ahogy korábban is említettük, ennek az elvonatkoztatási szintnek az elsődleges feladata a funkció, a viselkedés formális rögzítése, a megfelelő kódolási technikákat alkalmazva rengeteg olyan megvalósítási részletet is „belerejthetünk" a modellbe, amely tulajdonképpen már az áramköri szerkezetre vonatkozik, a leendő áramkör felépítésére, pontos áramköri megvalósítására utal. Mikroprocesszorok esetén ezt az áramköri szerkezetet mikroarchitektúrának is nevezzük, ami az elvonatkoztatás mértéke tekintetében leginkább a regiszter-transzfer szintnek feleltethető meg.

Az első, látszólag önkényesnek tűnő döntés az algoritmusszintű modellben, hogy az utasítások végrehajtását három különálló függvény segítségével írjuk le. Tömörebb modellt kaptunk volna, ha ezt a három részfolyamatot nem szervezzük külön függvényekbe, ráadásul elvileg még a modell futása is gyorsabb lehetne a függvényhívásokhoz kapcsolódó kontextusváltások elkerülésével. A részfolyamatok különálló függvényekbe szervezésével azonban érzékeltethetjük, hogy e három műveletet három különálló részáramkörrel kívánjuk majd megvalósítani. E három részáramkör saját funkcióját a viselkedési modell egyes függvényei egymástól függetlenül specifikálják, ami azt jelenti, hogy a három áramkör RTL tervezése akár teljesen párhuzamosan is történhet.

Ugyancsak önkényesnek, sőt feleslegesnek tűnhet maga a dekódolás fázis és a hozzá tartozó összes belső erőforrás. A végrehajtási fázisban közvetlenül is használhatnánk a dekódolásnál jobbértékként használt kifejezéseket, ahelyett, hogy egy sor belső változót veszünk fel e kifejezések értékének eltárolásához (pl. `ir.get_immediate()` kifejezés használata az `immediate` változó helyett a `loadimm` utasítás végrehajtása során az `execute()` függvényben). Ezeknek a változóknak a bevezetésével azonban azt igyekeztünk kifejezni, hogy a leendő áramkörben szándékunkban áll fizikailag is létrehozni ezeket a belső változókat egy-egy regiszter formájában. RTL tervezési szempontból ezek a regiszterek ún. pipeline regiszterréteget fognak képezni a processzor műveletvégző egységében, ami lehetőséget biztosít arra, hogy az utasításvégrehajtás egyes fázisait átlapoltan hajtsuk végre, így gyorsítva a processzor működését.

Figyeljük meg azt is, hogy az `op_a` és `op_b` operandusok értékét a dekódolás során még nem, hanem csak a végrehajtási fázisban olvassuk ki a regisztertömbből, míg a `mar` és `mdr` változók értékének kiszámításához szükséges információt már a dekódolásnál kiolvassuk ugyanabból a regisztertömbből. Mi lehet az oka ennek a különbségtételnek? Miért nem olvassuk ki az `op_a` és `op_b` operandusokat is a dekódolás fázisban ahelyett, hogy csupán a címüket (`op_a_index, op_b_index`) tároljuk el, és az értékeiket csak a végrehajtás során szedjük ki a regisztertömbből? Nos, a megoldás ismét az, hogy áramköri megvalósítási részletet igyekeztünk kifejezni; a regisztertömböt fizikailag egy memóriamodullal szeretnénk megvalósítani, amiről tudjuk, hogy egyszerre nem lehet belőle akármennyi adatot kinyerni, mivel egy ilyen modul író/olvasó interfészeinek száma korlátozott. Azzal, hogy a memóriareferens utasításokhoz tartozó változók értékét és az operandusok értékét két külön fázisban olvassuk ki a regisztertömbből, tulajdonképpen már viselkedési szinten megoldottuk a leendő memória-modulunk ütemezési problémáit; a dekódolás és a végrehajtás fázisban is csupán két olvasó interfészre van szükségünk, a végrehajtás során pedig egy további, ezekkel párhuzamosan működő beíró interfészre. Ez fizikailag viszonylag könnyen megvalósítható a hardverigény jelentős növekedése nélkül (lásd 4.5.4. pont: true-dual-port memória), ellentétben mondjuk egy négy interfészen olvasható és egy interfészen írható, ún. multiport memóriával.

Megfelelő utasításkódolással elérhető, hogy a memóriareferens utasítások és az aritmetikai utasítások a dekódolás fázisban a regisztertömb ugyanazon bejegyzéseire hivatkozzanak, ami szintén megoldaná az olvasó interfészek számának problémáját. Ugyanakkor a dekódolás során az operandusok helyett az operandusok címének az ideiglenes eltárolása a leendő áramkör erőforrás-felhasználása szempontjából is előnyös lehet; ha a dekódolás során mindkét operandus értékét kiolvasnánk a regisztertömbből, akkor legalább két akkora regiszterre lenne szükségünk, mint az operandusok mérete. Az operandusok vagyis a regisztertömb bejegyzéseinek méretét ugyan a viselkedési modell még nem definiálja, de feltételezhetjük, hogy ez a regisztertömb címzéséhez sükséges három bitnél nagyobb,[13] tehát egy sor flip-flopot megspórolhatunk azáltal, hogy csak az operandusok címét tároljuk el, a végrehajtó elemek – például az aritmetikai/logikai egység(ek) – pedig közvetlenül a regisztertömböt megvalósító memóriamodul kimeneti portjait olvasva végzik el feladatukat.[14]

Emeljük ki azokat a részleteket is, amelyekkel ez a viselkedési modell nem foglalkozik. Az első a számábrázolás kérdése. Ez a modell igen egyszerű ebből a szempontból, kizárólag előjel nélküli és előjeles egész számokkal operál, előbbit címzésre, utóbbit pedig aritmetikára használja. Ez a modell csak akkor tekinthető pontosnak számábrázolás szempontjából, ha a leendő áramkör pontosan ugyanazt az „int" és „unsigned" fogalmat használja, mint az a processzor, amely a modellünket futtatja. Ha például egy 8-bites processzor viselkedési modelljét készítjük el, akkor figyelembe kell vennünk, hogy a modellezett processzor „unsigned"-értelmezése eltérhet a modellt futtató processzorétól, tehát vagy saját adattípusokat kell bevezetnünk, amelyek figyelembe veszik a korlátozott bitszélességből adódó

[13] A 4.9. alfejezetben bemutatott RTL modellben például a regisztertömb bejegyzései 32-bitesek.

[14] Ez a fajta részletesség – ami már-már azt sugallja, hogy ezt az algoritmikus modellt valójában egy identitászavaros RTL tervező készítette – azért általánosságban nem jellemző a viselkedési modellekre, de mint példa, látványosan mutatja be, hogyan lehet „hardverszemlélettel" viselkedési modelleket létrehozni.

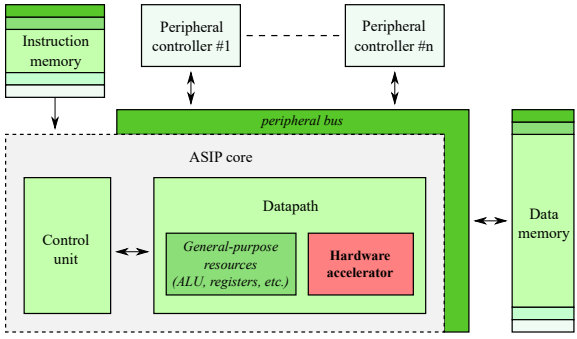
sajátosságokat (ábrázolható tartomány korlátai vagy kettes komplemens kódolás esetén az előjelbit helyiértéke), vagy magát a modellt kell úgy megírnunk, hogy ezeket figyelembe vegye (például egy összeadás művelet esetén a 32/64 biten kiszámított eredmény helyett az <eredmény % 256> értéket használjuk).

Az itt bemutatott modell ugyancsak nagyvonalúan bánik az aritmetikai műveletek megvalósítási nehézségeivel. Ebben a modellben semmi nem utal például arra, hogy az összeadás és az osztás művelet áramkörileg nagyon eltérő dolgok. Egy összeadás egy egyszerű – bár nagy bitszélességek esetén kétségkívül méretes – kombinációs hálózattal megvalósítható úgy, hogy tulajdonképpen még ütemezési problémák sem merülnek fel, mert feltételezhetjük, hogy egyetlen órajelciklus alatt elvégezhető a feladat. Ezzel szemben az osztás egy bonyolult művelet, amit reális hardverigény mellett csak iteratív úton, sok-sok órajelciklus alatt tudunk megvalósítani egy egész sor műveletvégző egység és ezek megfelelően ütemezett vezérlő jeleit előállító állapotgép segítségével (pl. *shift&subtract* vagy *Goldschmidt* algoritmussal). Modellünk tehát az osztás műveletről csak annyit állít, hogy valahogyan megtörténik, de azt nem részletezi, hogy hogyan.

3.2 Alkalmazásspecifikus áramkör viselkedési szintű modellje

Az előző példában látható processzor esetén egy egyszerű blokkdiagram, az utasításkészlet leírása és néhány, szöveges formában rögzített tulajdonság alkotta a rendszertervet, ezt tekintettük a viselkedési szintű modell specifikációjának. Az ebben a pontban bemutatott példa esetén ugyancsak viselkedési szintű, algoritmikus modellt készítünk, de ebben az esetben a rendszerterv eltérő jellegű lesz. E rendszerterv megértéséhez tekintsük át azt az áramköri környezetet, amelybe a leendő áramkört be akarjuk építeni.

A példa egy ún. hardveres gyorsítást megvalósító alkalmazásspecifikus mikroprocesszor (*Application-Specific Instruction-set Processsor*, ASIP) gyorsító áramkörének *(accelerator)* viselkedési szintű modelljét mutatja be. A 3-14. ábrán egy ilyen ASIP blokkvázlata látható.

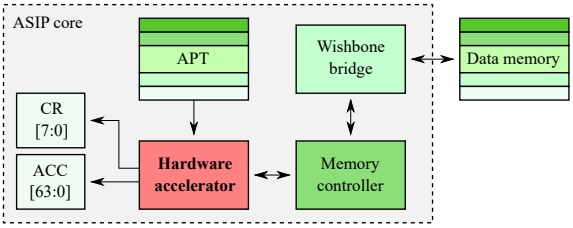


3-14. ábra. *Egy hardveres gyorsítást megvalósító ASIP felépítése*

Ez az ASIP tulajdonképpen egy olyan általános célú mikroprocesszor, amelynek műveletvégző egységébe egy alkalmazásspecifikus, tehát egy konkrét feladat vagy feladatkör (pl.

digitális jelfeldolgozás, képfeldolgozás stb.) hatékony elvégzésére képes áramkört építünk be. A processzor utasításkészlete a szokásos gépi utasításokon túl olyanokat is tartalmaz, amelyek kifejezetten ennek az egyedi áramkörnek a működtetésére szolgálnak. Ez a megoldás rugalmasság szempontjából nem túl előnyös, a gyorsító áramkör funkciója nem változtatható meg, tehát a processzorba épített extra hardver csak egy szűk alkalmazási területen lesz hasznos. Tegyük fel azonban, hogy ezt a rendszert újrakonfigurálható technológiára, például FPGA-ra szánjuk, és *soft-core*, tehát szintetizálható HDL leírás formájában bocsátjuk a felhasználó rendelkezésére. Ebben az esetben, ha több, eltérő funkciójú gyorsító áramkört is megvalósítunk, akkor lehetőség van arra, hogy a felhasználó szintézis előtt válassza ki a kívánt funkciót. Így egy sokkal rugalmasabb, szélesebb körben alkalmazható rendszert kapunk. Ahhoz, hogy az eltérő funkciójú gyorsító áramkörök mind beépíthetőek legyenek ugyanabba a műveletvégző egységbe, szigorúan definiálnunk kell, hogy a gyorsító áramkör a processzoron belül milyen erőforrásokkal van közvetlen kapcsolatban, milyen módon kommunikál velük. Röviden tehát létre kell hoznunk egy közös interfész-specifikációt, amihez funkciótól függetlenül minden gyorsító áramkörnek igazodnia kell.

A 3-15. ábra a példarendszerünk gyorsító áramkörének áramköri környezetét mutatja be.



3-15. ábra. *A példarendszer gyorsító áramkörének áramköri környezete*

Vegyük sorra, hogy a gyorsító áramkör milyen más erőforrásokkal van kapcsolatban az ASIP műveletvégző egységén belül:

- Gyorsító paramétertábla (*Accelerator Parameter Table*, APT): 32 × 32-bites regisztertömb, amelybe a processzor speciális adatmozgató utasítása segítségével írhatunk adatokat. A gyorsító áramkör ezt a regisztertömböt csak olvasni tudja. Elsődleges célja, hogy a gyorsító áramkör által végrehajtott feladat, algoritmus futásidejű paramétereit tárolja. Például ha a gyorsító áramkör gyors-Fourier transzformációt végez, akkor ebben a regisztertömbben tárolhatjuk el a transzformálandó vektor kezdőcímét az adatmemóriában vagy a transzformálandó vektor méretét. A konkrét gyorsító áramkör funkciójától függően természetesen nemcsak paramétereket, feldolgozandó adatokat is tárolhatunk itt, ha a regisztertömb mérete elegendő.
- Akkumulátor regiszter *(ACC)*: 64-bites regiszter a gyorsító áramkör által kiszámított eredmény tárolására. Az akkumulátor regiszter tartalmát a processzor be tudja írni az általános célú regisztertömbjébe, így további műveleteket tud végezni rajta.
- Hibakódregiszter *(Cause Register, CR)*: 8-bites hibakódregiszter, amelynek célja, hogy a gyorsító áramkör működése során felmerült esetleges hibák okát kódolja.

A processzor ennek a regiszternek az értékét is be tudja tölteni az általános célú regisztertömbjébe.
- Memóriavezérlő *(Memory controller)*: A gyorsító áramkör korlátozás nélkül hozzáfér a külső adatmemóriához is a memóriavezérlőn keresztül.

Az, hogy egy konkrét gyorsító áramkör pontosan mire használja az itt felsorolt erőforrásokat, természetesen a megvalósított funkciótól függ. A minden gyorsító áramkörre vonatkozó, közös interfészspecifikáció csupán azt rögzíti, hogy ezeken az erőforrásokon kívül a gyorsító áramkör nem „láthat" semmi mást, a külvilághoz való kapcsolódás csak ezeken keresztül lehetséges.

Az a tény, hogy a különböző funkciójú gyorsító áramköröknek vannak közös tulajdonságaik, nevezetesen a külső interfészük, modellezés szempontjából is fontos részlet. Az objektumorientált szemléletmód ennek a jellegzetességnek a modellezésére is kiválóan alkalmas eszközt biztosít, mégpedig az ősosztály és az öröklés fogalmakat. A gyorsítóáramkörök közös tulajdonságait egy közös ősosztály segítségével írjuk le, az egyedi funkciót megvalósító, konkrét gyorsító áramköröket pedig ennek az ősosztálynak a leszármazott osztályai képviselik. A közös ősosztály deklarációját a 3-16. ábra mutatja be.

C++

```
class accelerator {

    public:

        accelerator(std::vector<double>& the_parameter_table, std::vector<double>& the_data_memory,
                    double& the_acc, unsigned& the_cause_register) :
            parameter_table(the_parameter_table),
            data_memory(the_data_memory),
            acc(the_acc),
            cause_register(the_cause_register) {}

        const std::vector<double>& parameter_table;
        std::vector<double>& data_memory;
        double& acc;
        unsigned& cause_register;

        virtual ~accelerator() {}
        virtual void run() = 0;

};
```

3-16. ábra. *Az accelerator ősosztály deklarációja*

A gyorsító ősosztály a processzoros példánál látottakhoz hasonlóan a leendő áramkör környezetében lévő erőforrásokra referenciákkal hivatkozik és deklarál egy `run()` függvényt, amelynek hívásával a gyorsító áramkör működése elindítható. Ez a függvény az absztrakt ősosztályban tisztán virtuális tagfüggvény, hiszen az ősosztály esetén még nincs konkrét funkció, ezt a függvényt az egyedi funkciót megvalósító leszármazottak fogják túlterhelni.

Példánkban a konkrét funkció, amit a gyorsító áramkör megvalósít, a klaszteranalízis alkalmazási területén széleskörűen alkalmazható, ún. súlyozott Minkowski-távolság kiszámítása lesz. Ez az algoritmus két, tetszőleges méretű vektor távolságát számítja ki egy harmadik, ún. súlyvektort is felhasználva az alábbi definíció alapján:

$$d_{ab} = \sqrt[r]{\sum_{k=1}^{p}(w_k^r \times |x_{ak} - x_{bk}|^r)}$$

A fenti távolság-definíció jelölései a következők:

- d_{ab}: A két vektor távolsága;
- r: távolság típusa (pl. 1: Manhattan távolság, 2: Euklideszi távolság);
- p: A vektorok dimenziószáma;
- w: súlyvektor;
- x_{ak}: az „A” vektor k-adik koordinátája;
- x_{bk}: a „B” vektor k-adik koordinátája.

A 3-17. ábra a Minkowski-távolságot kiszámító áramkör, a „Minkowski-gyorsító” viselkedési modelljét jelentő osztály deklarációját mutatja be.

C++

```
class accelerator_minkowski : public accelerator {

    private:

        void nth_root(double argument, unsigned n, double& nth_root);
        bool prim_division(bool division_type, double dividend, double divisor,
                           double& quotient, int& remainder);

    public:

        accelerator_minkowski(std::vector<double>& the_parameter_table,
                              std::vector<double>& the_data_memory,
                              double& the_acc,
                              unsigned& the_cause_register) :
            accelerator(the_parameter_table, the_data_memory, the_acc, the_cause_register) {}

        ~accelerator_minkowski() {}

        void run();

};
```

3-17. ábra. *A Minkowski-gyorsítót képviselő osztály deklarációja*

A Minkowski-gyorsító viselkedési modelljével kapcsolatban – a processzoros példánál látottakhoz hasonlóan – elmondható, hogy látszólag indokolatlan strukturáltság jellemzi. Bevezettünk két privát tagfüggvényt; egyet az n-edik gyök kiszámítására (`nth_root()`), egyet pedig az osztás művelet elvégzésére (`prim_division()`). Ennek az algoritmusszinten meghozott döntésnek is modellezési szempontból van jelentősége; azzal, hogy e funkciókat különálló, saját interfésszel rendelkező függvényekként valósítjuk meg, valójában létrehoztunk két új, akár a gyorsító áramkör saját funkciójával párhuzamosan fejleszthető almodult.

Felmerülhet a kérdés, hogy miért pont gyökvonás és osztás? Az, hogy n-edik gyököt kell számolnunk, a Minkowski-norma definíciójából egyértelmű. Reálisnak tekinthető hardverigény mellett a gyökvonás műveletét iteratív úton érdemes elvégezni. Ennek az iteratív műveletnek a részeként reciprokszámításra is szükség van, amit a különálló tervezési

egységként modellezett osztó fog elvégezni. E két segédfüggvény definícióját a 3-18. és 3-19. ábrák mutatják be.

C++

```
void accelerator_minkowski::nth_root(double argument, unsigned n, double& nth_root) {

    int dummy = 0;
    prim_division(false, argument, 2, nth_root, dummy);

    double diff = 0;
    double c = 0;
    double p = 0;
    double a = 0;

    prim_division(false, 1, n, c, dummy);

    for ( unsigned i = 0; i < 10; ++i ) {

        p = pow(nth_root, (double)(n-1));
        prim_division(false, argument, p, a, dummy);

        diff = c * ( a - nth_root );
        nth_root = nth_root + diff;
    }

}
```

3-18. ábra. *Az n-edik gyököt kiszámító függvény definíciója*

C++

```
bool accelerator_minkowski::prim_division(bool division_type, double dividend,
                                          double divisor, double& quotient, int& remainder) {

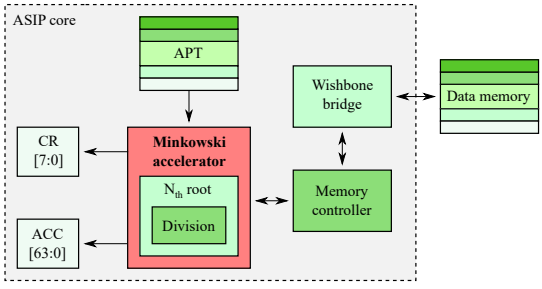
    if ( divisor == 0 ) return false;

    if ( division_type == false ) {
        quotient = dividend / divisor;
    } else {
        quotient = int(dividend / divisor);
        remainder = int(dividend) % int(divisor);
    }
    return true;

}
```

3-19. ábra. *Az osztást megvalósító függvény definíciója*

Azzal, hogy a gyökvonás és osztás műveleteket különálló függvényekbe szerveztük ki, tulajdonképpen a 3-20. ábrán látható áramköri szerkezetet igyekeztünk kifejezni.



3-20. ábra. *A Minkowski-távolságot kiszámító gyorsító áramkör szerkezete*

A gyorsító áramkörrel kapcsolatban rögzíteni kell, hogy miként fogja használni az őt körülvevő, a 3-15. ábrán szemléltetett erőforrásokat. Az elvégzendő feladatot specifikáló matematikai összefüggés alapján elmondható, hogy a számítási művelet bemenete két vektor *(x_a* és *x_b)*, amelyeknek a távolságát akarjuk kiszámítani, egy súlyvektor *(w)*, a kiszámítandó távolságmérték típusa *(r)* és a bemenő vektorok dimenziószáma *(p)*. Bemenő paraméterek tárolására az APT regisztertömböt használhatjuk fel, de mivel a bemenő vektorok mérete előre nem ismert, ezért a vektor-paramétereket nem másolhatjuk oda, ezeknek csak az adatmemóriabeli kezdőcímeit tárolhatjuk itt. A gyorsító áramkörnek a számítás során az adatmemóriából kell kiolvasnia a két bemenő vektort és a súlyvektort is. Az APT-ben tárolt paraméterek tehát a következők:

- APT[0]: Az x_a vektor kezdőcíme az adatmemóriában;
- APT[1]: Az x_b vektor kezdőcíme az adatmemóriában;
- APT[2]: A w vektor kezdőcíme az adatmemóriában;
- APT[3]: A bemenő vektorok mérete;
- APT[4]: A távolságmérték típusát megadó r paraméter.

A gyorsító áramkör eredményül egyetlen számot ad, ennek eltárolására az akkumulátor regisztert fogjuk használni. A megvalósított számítás matematikai definíciójából adódóan az r paraméter értéke nem lehet egynél kisebb. Ha a felhasználó rossz értéket ad meg, akkor ezt valamilyen módon jeleznünk kell, amire a hibakód-regiszter tökéletesen alkalmas.

C++

```
void accelerator_minkowski::run() {

    unsigned base_address_a = parameter_table[0];
    unsigned base_address_b = parameter_table[1];
    unsigned base_address_w = parameter_table[2];
    unsigned variable_vector_size = parameter_table[3];
    unsigned minkowski_r = parameter_table[4];

    double v_a_i = 0;
    double v_b_i = 0;
    double v_w_i = 0;
    double p1 = 0;
    double p2 = 0;
    double distance = 0;

    if (minkowski_r < 1) {
        cause_register = 1;
        throw ("r parameter of the Minkowski-distance cannot be less than 1!");
    }

    for (unsigned i = 0; i < variable_vector_size; ++i) {
        v_a_i = data_memory[base_address_a + i];
        v_b_i = data_memory[base_address_b + i];
        v_w_i = data_memory[base_address_w + i];
        p1 = pow(abs(v_a_i - v_b_i), (double)minkowski_r);
        p2 = pow(v_w_i, (double)minkowski_r);
        distance = distance + p2 * p1;
    }

    nth_root(distance, minkowski_r, distance);
    acc = distance;

}
```

3-21. ábra. *A Minkowski-távolságot kiszámító függvény definíciója*

Ha *r* értéke a számítás megkezdésekor egynél kisebb, akkor a 0x01 hibakódot írjuk a hibakódregiszterbe. A fenti konvenciók rögzítése után a Minkowski-távolságot kiszámító áramkör viselkedési szintű modellje elkészíthető (lásd 3-21. ábra).

A gyorsító áramkör funkcionális ellenőrzéséhez létrehozott tesztkörnyezet a processzoros példánál látottakhoz hasonlóan létrehozza az áramköri modell komponenseit, megvalósítja a közöttük lévő kapcsolatokat, gerjesztéssel látja el a modellt, majd ellenőrzi az az általa visszaadott eredményt. A Minkowski-gyorsítóhoz készített tesztkörnyezet a 3-22. ábrán látható.

C++

```
int main() {

    // Instantiate the model elements.
    std::vector<double> apt(32);
    std::vector<double> data_memory(1024);
    double acc = 0;
    unsigned cause_register = 0;
    accelerator_minkowski accel_mink(apt, data_memory, acc, cause_register);

    // Prepare the input data for the accelerator.
    data_memory[0] = 2.0; data_memory[1] = 3.6; data_memory[2] = 7.9;  // v_a
    data_memory[3] = 0.0; data_memory[4] = 3.4; data_memory[5] = 1.2;  // v_b
    data_memory[6] = 2.0; data_memory[7] = 3.0; data_memory[8] = 5.0;  // v_w

    apt[0] = 0; // v_a base address
    apt[1] = 3; // v_b base address
    apt[2] = 6; // v_w base address
    apt[3] = 3; // variable_vector_size
    apt[4] = 2; // r parameter of the Minkowski-distance

    // Start simulation.
    accel_mink.run();

    // Check the result
    double p1 = 0;
    double p2 = 0;
    double distance_ref = 0;

    for ( unsigned i = 0; i < apt[3]; ++i ) {
        p1 = pow(abs(data_memory[apt[0]+i] - data_memory[apt[1]+i]), (double)apt[4]);
        p2 = pow(data_memory[apt[2]+i], (double)apt[4]);
        distance_ref = distance_ref + p2 * p1;
    }
    distance_ref = sqrt(distance_ref);

    if ( abs(acc == distance_ref) < 1e-6 ) std::cout << "CHECK OK" << std::endl;
    else std::cout << "CHECK FAIL" << std::endl;

    return 0;

}
```

3-22. ábra. *A Minkowski-gyorsító viselkedési modelljének tesztkörnyezete*

Ez a tesztkörnyezet a gyorsító áramkört természetesen nem verifikálja kimerítően, csupán egyetlen tesztesetet futtat le. A verifikáció során arra kell törekedni, hogy ilyen tesztesetekből minél többet létrehozzunk annak érdekében, hogy az áramköri modell esetleges hibáira fény derüljön, hiszen ezt a modellt az alacsonyabb elvonatkoztatási szintű tervezési folyamatok a saját verifikációs eljárásukban referenciának fogják tekinteni.

Abból adódóan, hogy a programozási nyelvek – különösen az objektumorientált szemlélettel együtt alkalmazva őket – tulajdonképpen általános modellező nyelveknek

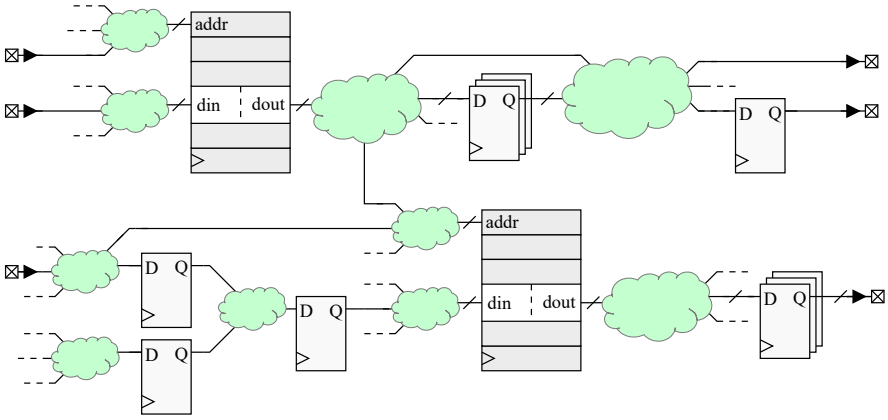
tekinthetők, a viselkedési modellek részletessége tovább növelhető. A fenti példákban is láthatóhoz hasonló modellezési konvenciók (pl. „osztály/függvény = almodul", „referencia = külső interfész") bevezetésével modellünk egyre részletesebben lehet képes visszaadni a tervező szándékát az áramkör viselkedésére és szerkezetére vonatkozóan egyaránt. Abból azonban, hogy a modellező nyelv saját eszközei helyett konvenciókra van szükség azt sejteti, hogy valójában eljutottunk egy olyan részletességig, olyan alacsony a leírni kívánt modell elvonatkoztatásának mértéke, hogy már igazából nem programozási nyelveket kellene használnunk a terv további finomításához. A programozási nyelvek és az objektumorientált szemlélet kiválóan alkalmas a viselkedés rögzítésére és az áramkör nagyvonalú szerkezetének leírására, azonban a finomabb, regiszterszintű felépítés és a regiszterek közötti adatátvitel részletes leírására, vagyis a regiszter-transzfer szintű modellezésre érdemes a kifejezetten erre a célra optimalizált hardverleíró nyelveket használnunk.



4

RTL tervezés, hardverleíró nyelvek

A digitális áramkörökre funkciójuktól és összetettségüktől függetlenül igaz az, hogy fizikailag tároló elemekből (flip-flopok, memóriák), kombinációs hálózatokból és az ezeket összekapcsoló vezetékekből állnak (lásd 4-1. ábra).



4-1. ábra. *A digitális áramkörök általános szerkezete* [15]

Ez a zavarba ejtően egyszerű modell bármely digitális áramkör esetén megállja a helyét, az egyszerű időzítő áramköröktől a legbonyolultabb többprocesszoros rendszerekig. A regiszter-transzfer szintű tervezés során a célunk ennek a finomszerkezetnek a kifejezése, vagyis annak meghatározása, hogy a viselkedési szinten specifikált feladatot ellátó áramkörben milyen tároló elemekre van szükség, valamint hogy azok ki- és bemenetei között milyen logikai műveleteket milyen sorrendben kell elvégezni. Az RTL tervezés – a magasabb elvonatkoztatású modellek létrehozásával ellentétben – már nagyon hasonlít arra a folyamatra, amire hagyományosan „áramkörtervezés”-ként gondolunk; egy viszonylag egyszerű alkatrészkészletből (regiszterek, memóriák, multiplexerek, aritmetikai/logikai funkciót ellátó kombinációs áramkörök) a megfelelő elemeket kiválasztva és azokat alkalmas módon

[15] A 4-1. ábrán látható felhő szimbólumok a kombinációs hálózatokat jelképezik *(combinational cloud)*. Ez a modell elsőre haszontalannak tűnhet, hiszen az áramkörnek a funkciójáról nem mond semmit, ugyanakkor szerkezeti szempontból ez a reprezentáció elég sok információt rejt, különösen akkor, ha bármely két regiszter közötti kombinációs felhő késleltetését is ismerjük. Éppen ezért az ilyen modelleket gyakran használjuk már elkészült áramkörök időzítési viszonyainak vizsgálatához, ahol a konkrét funkció nem, csak a struktúra és a késleltetések számítanak.

összekapcsolva hozunk létre egy összetettebb funkciót megvalósító áramkört, hasonlóan ahhoz, mint amikor ellenállásokból, kondenzátorokból, erősítőkből, komparátorokból és ehhez hasonló alkatrészekből építünk analóg áramköröket.[16]

4.1 Erőforrás-allokáció és ütemezés

Az RTL tervezés során tulajdonképpen erőforrás-allokációs és ütemezési feladatokat kell elvégeznünk. Mielőtt beleásnánk magunkat az RTL tervezés során felmerülő problémák és megoldások részleteinek tárgyalásába, tekintsünk át egy, az erőforrás-allokáció és ütemezés kérdését szemléletesen bemutató, egyszerű példát. Induljunk ki egy C++ nyelvű viselkedési modellből annak ellenére, hogy a funkció nagyon egyszerű (lásd 4-2. ábra).[17]

C++

```
void accumulator(int& sum, bool& zero, int op_a, int op_b, int op_c, int op_d) {
    sum = op_a + op_b + op_c + op_d;
    if ( sum == 0 ) zero = 1;
    else zero = 0;
}
```

4-2. ábra. *Egy akkumulátor áramkör viselkedési szintű modellje*

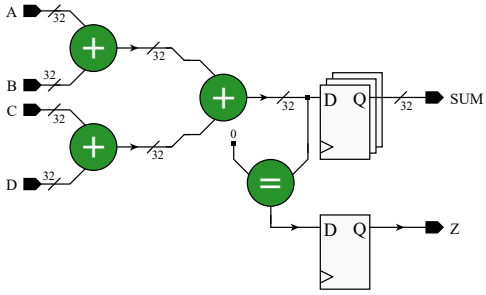
A feladat tehát egy olyan áramkör RTL modelljének elkészítése, amely négy, előjeles egész szám összeadását végzi el, és jelzi, hogy az eredmény nulla-e vagy sem. Fontos megjegyezni, hogy az RTL tervezés a specifikáció ehhez hasonló szubjektív értelmezésével, a feladat „megértésével" kezdődik akkor is, ha a specifikáció formális módon adott, ti. az RTL tervezés ma még az esetek túlnyomó többségében kézi úton történik.[18] Az algoritmikus modell ebben a példában semmiféle figyelmet nem fordít arra, hogy a leendő áramkör szerkezetéről bármit is sugalljon, így RTL tervezőként a mi feladatunk lesz kitalálni azt, hogy milyen erőforrásokat kell használnunk, és ezeket hogyan kell összekapcsolnunk annak érdekében, hogy a négy szám összegét előállítsuk. Féktelen kreativitásunktól vezérelve az erőforrás-allokáció fázisában arra a következtetésre jutunk, hogy ezt a funkciót alapvetően összeadó áramkörökkel lehet kezelni. Tudjuk azt, hogy a cellakönyvtárainkban az összeadó áramkörök kétoperandusú kivitelben állnak rendelkezésre, ezért úgy döntünk, hogy az RTL modellezés során a 32-bites, kétoperandusú összeadó áramkört tekintjük alapelemnek. Valójában már ez a döntésünk is egy alkalmas absztrakció, hiszen nem foglalkozunk azzal, hogy ez az összeadó egy *ripple-carry*, *carry-save* stb. összeadó, a kapuszintű megvalósítás tehát egyelőre nem fontos. Annak megállapítására, hogy az összeg nulla-e, egy komparátort

[16] Vegyük észre, hogy bár a digitális rendszertervezés 1. pontban bemutatott módszerére alapvetően a top-down megközelítés jellemző, a design-flow részfolyamataiban elrejtve a bottom-up jelleg is tetten érhető.

[17] Az RTL tervező számára a specifikáció sokféleképpen rendelkezésre állhat. Bár az 1. fejezetben bemutatott tervezési folyamat azt sugallja, hogy az RTL tervezés bemenete mindig egy viselkedési szintű formális modell, bizonyos esetekben az algoritmikus reprezentáció hiányzik, vagy legalábbis nem formális jellegű, és az RTL modellt egy, a rendszerszintű modellhez közelebb álló, informális specifikáció (pl. egy kommunikációs interfészt leíró szabvány szövege) alapján közvetlenül kell előállítanunk. Az itt bemutatott példához hasonló, nagyon egyszerű esetekben rendszerint nincs viselkedési szintű reprezentáció.

[18] Léteznek eljárások az RTL modellek automatizált előállítására formális viselkedési szintű modellekből (*High Level Synthesis*, HLS), de ezek általában csak valamilyen szűk alkalmazási területen, például DSP alkalmazásokban használhatók hatékonyan, és ma még nem tekinthetők széles körben elterjedt, általánosan használt módszereknek.

fogunk használni, aminek a belső szerkezetével, kapuszintű megvalósításával ugyancsak nem foglalkozunk. Az összeadó és a komparátor áramköröket egyaránt kombinációs hálózatként kezeljük. El akarjuk érni azt is, hogy az RTL modellünk szinkron működésű áramkört írjon le, így regiszterekre is szükségünk lesz az eredmény eltárolásához. A kiválasztott erőforrások legegyszerűbben a 4-3. ábrán látható módon kapcsolhatók össze a kijelölt funkció megvalósításához.

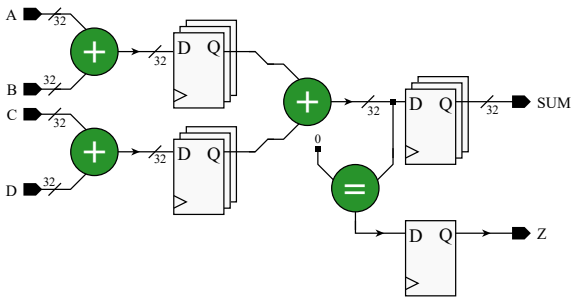


4-3. ábra. *Az akkumulátor áramkör egy lehetséges RTL modellje (1)*

Ez a modell informális, kapcsolási séma jellege ellenére tekinthető RTL modellnek, hiszen tartalmazza azt, hogy milyen tároló és logikai erőforrásokat használunk, megfogalmazza, hogy ezeket hogyan kell összekapcsolni, és még az áramkör működésének ütemezését is pontosan leírja; ebben a megvalósításban a rendszerórajel minden felfutó élénél – és csak akkor – előáll egy új eredmény az akkumulátor áramkör kimenetén, úgy, hogy ez a kimenet annak a négy operandusnak az összege, amelyek ugyanannak a felfutó élnek a pillanatában a bemeneteken voltak. Ez a modell tehát kevésbé elvont, sokkal több információt tartalmaz, mint a viselkedési modell, amiből kiindultunk.[19]

Egy alacsonyabb elvonatkoztatású modell egy magasabb elvonatkoztatású modell alapján természetesen többféleképpen is előállítható, hiszen nemcsak átalakítjuk a leírást a kiinduláshoz képest, hanem új információval bővítjük azt, az adott absztrakciós szintre jellemző tervezői döntések meghozásával. A különböző tervezői döntések tehát más-más megoldást eredményeznek. Regiszter-transzfer szinten ezek a döntések elsősorban a felhasznált erőforrásokra, azok összekapcsolására és a működés ütemezésére vonatkoznak, lássuk tehát, hogy milyen más RTL modellt lehet még készíteni az akkumulátor áramkörünkhöz. A 4-3. ábrán látható megoldásra tekintve például – dörzsölt RTL tervezőként – arra juthatunk, hogy 32-bites, tisztán kombinációs hálózatként megvalósított összeadó áramköröket sorbakötni nem biztos, hogy jó ötlet, mert a kiadódó logikai szintek száma, ezáltal a fizikailag megvalósított áramkör adatútjainak késleltetése nagyon nagy lesz, ami korlátozni fogja az áramkör órajelének frekvenciáját. Ezt a problémát pipeline regiszterek beiktatásával fogjuk kezelni. Az új RTL modell a 4-4. ábrán látható.

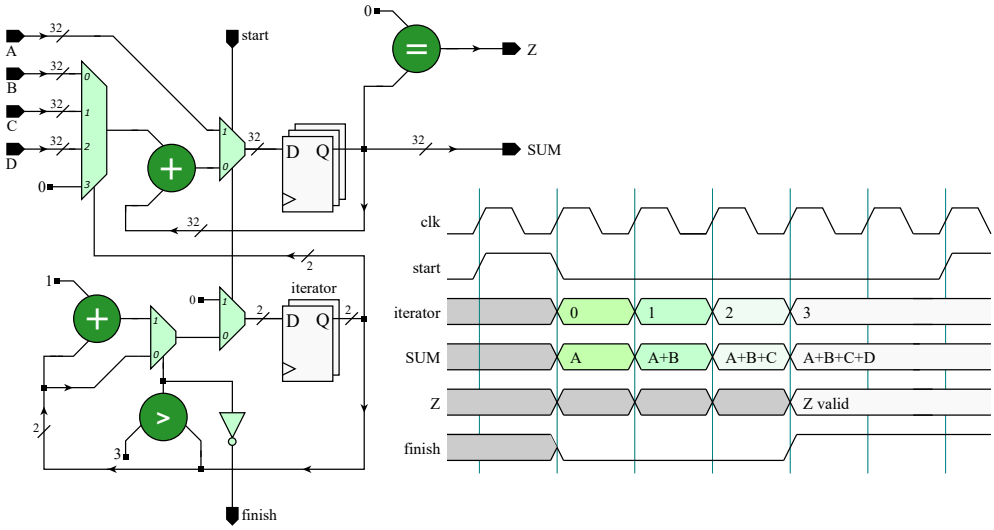
[19] Természetesen egy ilyen kapcsolási séma csak akkor tekinthető megfelelő RTL modellnek, ha az egyes alkatrész-szimbólumok jelentése konvencionális, jól ismert a modellt használók számára.



4-4. ábra. *Az akkumulátor áramkör egy lehetséges RTL modellje (2)*

A korábbi modellben kritikusnak ítélt kombinációs hálózatokat tehát kettészeltük egy pipeline regiszterréteggel, ami erőforrás-allokációs és ütemezési szempontból is egy, az előzőtől eltérő megvalósítást jelent. 2 db 32-bites regiszterrel többet használunk, mint korábban, ráadásul, bár továbbra is igaz, hogy az órajel minden felfutó élénél előáll egy új eredmény az áramkör kimenetén, de az n-edik felfutó élnél előálló eredmény ezúttal nem az n-edik, hanem az (n-1)-edik felfutó élnél jelenlévő operandusok összege. Ez persze nem jelenti azt, hogy az RTL modell ne adná vissza helyesen azt a funkciót, amit a viselkedési modell megfogalmazott. A viselkedési modell semmit nem állított az áramkör időzítési tulajdonságairól, tehát ez a megoldás is ugyanúgy helyesnek tekintendő, mint az első. A 4-4. ábrára tekintve ezúttal az a tény lehet szálka a szemünkben, hogy ennek az egyszerű funkciónak a végrehajtására felhasználtunk 97 db flip-flopot, 3 db 32-bites összeadót és még némi logikát – 31 db kétbemenetű OR kaput és egy invertert (aki nem hiszi, járjon utána) – a komparátorhoz. Ez a nagy erőforrásigény lehetővé teszi, hogy az áramkörünk átbocsátóképessége, vagyis az egy órajelciklus alatt feldolgozott adatmennyiség a lehető legnagyobb legyen, mégis joggal érezhetjük kényelmetlenül magunkat, hiszen a nagy erőforrásigényhez vezető nagy átbocsátóképességet nem írta elő senki, a specifikációnak ez nem képezte részét. Márpedig ebben az esetben egy olyan modellt állítottunk elő, ami a feltétlenül szükségesnél nagyobb áramkört ír le. Lelkiismeretes – és nem mellesleg ráérő – RTL tervezőként újabb modellt készítünk, amely a négy szám összeadásának műveletét iteratív úton, az összeadó áramkörök vezérelt újrafelhasználásával oldja meg (lásd 4-5. ábra).

Ez az RTL modell erőforrás-felhasználás szempontjából kedvezőbb, mint az előző két megoldás, mivel csak egy darab 32-bites összeadót tartalmaz. Maximális órajel-frekvencia szempontjából pedig az előző két megoldás között helyezkedik el, mivel az első esetben két összeadó, a másodikban egy összeadó, ebben pedig egy összeadó és egy multiplexer képezik a legnagyobb késleltetésű adatutat. Ehhez arra volt szükség, hogy az áramkör szerkezetén kicsit bonyolítsunk, és bevezessünk néhány olyan erőforrást – egy ún. számlálós vezérlőt –, amely elvégzi az egyetlen 32-bites összeadó áramkörünk operandusainak megfelelő időzítésű multiplexálását. Ütemezési szempontból tehát ugyancsak bonyolultabb lett a helyzet, ezúttal a *start* jel felhasználó általi kiadása után csak az órajel negyedik felfutó élénél áll elő az eredmény. Mivel a felhasználónak a műveletvégzés alatt kötelessége változatlanul tartani az operandusok értékét, ezért a teljesség kedvéért előállítottunk egy állapotjelző kimenetet



4-5. ábra. *Az akkumulátor áramkör egy lehetséges RTL modellje (3)*

is *(finish)*, ami jelzi a felhasználónak, hogy a legutóbb kezdeményezett művelet véget ért, az eredményt kiolvashatja és kezdeményezheti a következő műveletet.

Fontos hangsúlyozni, hogy az itt bemutatott példák közül egyiket sem lehet a „minden szempontból tökéletes" megoldásnak tekinteni, és ez az RTL tervezésre általánosan is igaz. Egy-egy problémára rengeteg féle jó megoldást lehet adni, amelyek erőforrásigény, számítási teljesítmény, fogyasztás és – nem utolsó sorban – a fejlesztésre és verifikációra fordítandó idő szempontjából eltérnek. Az RTL tervező feladata az, hogy az összetett követelményrendszer alapján, az egymásnak ellentmondó peremfeltételeket fontossági sorrendbe állítva – határidő előtt – megtalálja azt a bizonyos „nem minden szempontból tökéletes, de megfelelő" megoldást. Az RTL tervezés alapproblémájának szemléltetése után e fejezet további részei az RTL modellezés részleteivel foglalkoznak.

4.2 RTL modellek formális leírása

A hagyományos narratíva szerint „RTL tervezésre hardverleíró nyelveket használunk". Ez a megfogalmazás – bár lényegében igaz – kissé pontatlan. Az RTL tervezés valójában fejben történik. Erre, az egyébként bármilyen tervezési folyamatra jellemző sajátosságra az ezzel foglalkozó szakirodalom kognitív modellezésként hivatkozik. E rendkívül tudományosan hangzó kifejezéssel egy kocsmai beszélgetés során hamar a figyelem középpontjába – vagy asztaltársaságtól függően annak peremére – kerülhetünk ugyan, de lényegében csak arról van szó, hogy a tervező elképzeli az előírt funkciót megvalósító áramkör egy részletét, majd ezután valamilyen formában rögzíti azt, mert a teljes rendszert megvalósító megoldást egyszerűen nem lenne képes fejben tartani. A tervnek e köztes rögzítéséhez nem feltétlenül szükségesek még a HDL-ek, ahogy azt az előző pontban bemutatott egyszerű

példa is alátámasztja; három különböző RTL tervet mutattunk be anélkül, hogy akár csak megemlítettük volna a HDL-eket. Ugyanakkor az is igaz, hogy az ott látható kapcsolási rajzokkal önmagukban a CAD szoftverek mit sem tudnak kezdeni, hiszen az egyes alkatrész-szimbólumokhoz nincsen egzaktul hozzárendelve az általuk jelképezett alkatrész funkciója, viselkedése. Pontosan ez az, ami miatt a HDL-ekre szükségünk van; ezek azok a formális nyelvek, amelyek ellentmondásmentesen, a tervezést segítő számítógépes eszközök – szimulátorok, szintézis és analízis szoftverek – számára is feldolgozható formában képesek rögzíteni az RTL tervező szándékát *(RTL intent)*. A gépi módszerek hajnalának hagyatékaként bizonyos szoftverrendszerek ugyan még ma is lehetővé teszik azt, hogy RTL modelljeinket részletes kapcsolási rajzzal definiáljuk, ez a megoldás azonban – egyrészt a jellemző összetettség megnövekedése miatt, másrészt amiatt, hogy ezek a modellek az adott szoftverre jellemző, nem szabványos, ezáltal nem hordozható formában definiálják a tervet[20] – nem tekinthető olyan általánosnak, mint a HDL-ek, azon belül is a VHDL és a Verilog/SystemVerilog használata.

4.2.1 Egy kis történelem…

A VHDL angol betűszó, feloldása: *VHSIC Hardware Description Language*, amelyben a VHSIC a *Very High Speed Integrated Circuits* rövidítése. Ez utóbbi az USA Védelmi Minisztériumának (*Department of Defense*, DoD) programja, amelynek keretein belül 1983-ban az IBM, a Texas Instruments és az Intermetrics cégek nyerték el azt a pályázatot, amelynek célja egy szabványosítható, integrált áramkörök működésének és szerkezetének formális rögzítésére alkalmas modellező nyelv és a hozzá tartozó alapvető szoftver eszközök kidolgozása volt. A VHDL nyelvnek több verzióját is kifejlesztették (VHDL 2.0, VHDL 6.0), majd 1984-ben a munkát az IEEE *(International Institute of Electrical and Electronics Engineers)* folytatta a nyelv szabványosításával. Ennek a munkának eredményeként született meg 1987-ben az IEEE STD 1076/A jelű szabvány, amelynek címe *VHDL Language Reference Manual*. A szabványt többször módosították (IEEE STD 1076-1993, IEEE STD 1076-2002, IEEE STD 1076-2008), legutóbbi frissítésére 2019-ben került sor. Az ebben a fejezetben bemutatott összes példa megfelel a VHDL-2019 szabványnak.

A Verilog *(VERIfication and LOgic)* nyelvet a Gateway Design Automation cég fejlesztette ki 1984-ben. 1989-ben a Gateway Design Automationt a Cadence Design Systems felvásárolta, majd 1990-ben a Verilog nyelvet nyílt szabvánnyá tette az Open Verilog International gondozásában. A Verilog 1995-ben lett IEEE szabvány (IEEE STD 1364). 2005-ben jelentős fejlesztéseket végeztek a szabványon (IEEE STD 1364-2005), főként a funkcionális verifikáció hatékonyabbá tétele érdekében. Ugyancsak 2005-ben jelent meg a SystemVerilogot leíró szabvány (IEEE STD 1800-2005), amely teljes egészében tartalmazta magát a Verilogot. Ezt a két szabványt 2009-ben összevonták (IEEE STD 1800-2009). A SystemVerilog nyelvet leíró legfrissebb szabványmódosítást 2017-ben végezték (IEEE STD 1800-2017). Ebben a fejezetben a SystemVerilog nyelv 2017-es verzióját

[20] A kapuszintű modellekre léteznek szabványos formátumok (pl. EDIF).

használjuk. A szakmai közösség továbbra is egyszerűen Verilogként utal a nyelvre akkor is, ha SystemVerilogról van szó, ezt a konvenciót követjük mi is a fejezet további részeiben.

4.2.2 VHDL vs Verilog

A VHDL és a Verilog szintaxis tekintetében eléggé különbözőek, ami annak köszönhető, hogy nyelvtani szempontból a VHDL az Ada, a Verilog pedig a C nyelven alapul. Ebből adódóan az általános vélekedés az, hogy a Verilogot könnyebb megtanulni, és ez valószínűleg igaz is, tekintve, hogy akik a HDL-alapú modellezés elsajátítását tűzik ki célul, azok általában már használták a C programozási nyelvet, az Ada viszont csak szűk körben ismert. Ez a magyarázata annak is, hogy – különösen hazánkban – az egyetemi kurzusok a HDL-tervezés oktatásával kapcsolatban inkább a Verilogot preferálják. A Verilog túlreprezentáltsága az oktatásban érdekes módon nem eredményezte a VHDL teljes eltűnését, ami valószínűleg annak köszönhető, hogy idővel azok is elsajátítják a VHDL rejtelmeit, akik a HDL-tervezést a Veriloggal kezdték. Az ipari gyakorlatban ugyanis mindkét nyelvet használják, sokszor egymással párhuzamosan, akár egyetlen projekten belül is, szóval egy RTL tervezőnek előbb-utóbb mindkettőt meg kell tanulnia.

RTL modellezés szempontjából a VHDL és a Verilog egyformán hatékonyak, amit mi sem bizonyít jobban, mint hogy a szintetizálható nyelvi elemek halmaza az elmúlt három évtizedben szinte semmit nem változott, annak ellenére, hogy mindkét szabvány átesett kisebb-nagyobb módosításokon. Általánosságban elmondható, hogy a VHDL a Verilognál sokkal „szigorúbb", erősen típusos nyelv, amely a modellezett áramkörrel kapcsolatban a lehető legtöbb részlet egzakt kifejezését várja el a tervezőtől, semmit nem „próbál meg kitalálni". A Verilog ugyanakkor sok implicit szabályt tartalmaz (pl. a `logic` típusú vektorok alapértelmezés szerint előjel nélküli egésznek tekintendők, egybites jeleket és regisztereket nem kell deklarálni stb.), ami a kód terjedelme, így közvetve a fejlesztés gyorsasága szempontjából kétségkívül előnyösebb, mint a VHDL „szószátyár, szőrszálhasogató stílusa", ugyanakkor be kell látnunk, hogy a különböző szimulációs és szintézis szoftverek által sokszor nem egységesen kezelt, implicit szabályok és a csak a Verilogra jellemző, ún. kritikus versenyhelyzetek láttán az RTL tervezők jelentős részét joggal kerülgeti a rosszullét. A következő alfejezetekben nem kisebb feladatra vállalkozunk, mint hogy a két nyelvet egymással párhuzamosan mutassuk be, így az Olvasó könnyen összehasonlíthatja őket.

4.2.3 Szimulációs és szintézis szemantika

Ebben a fejezetben egy RTL tervező szemével tekintjük át a VHDL és a Verilog nyelvek képességeit, tulajdonságait. Ez azt jelenti, hogy a nyelvtani sajátosságok minden részletre kiterjedő tárgyalása helyett a nyelveknek csupán az ún. „szintetizálható részhalmazával" foglalkozunk, vagyis azokkal a nyelvi szerkezetekkel, amelyeket nemcsak a szimulációs, hanem a szintézis szoftvereink is képesek feldolgozni, belőlük alacsonyabb elvonatkoztatású,

kapuszintű reprezentációkat tudnak előállítani. Ebből a gondolatból rögtön látszik az is, hogy pontosan mi is a célunk a HDL-ekkel leírt RTL modellekkel:

- **Logikai szimuláció**: Ha elkészítettünk egy RTL modellt, szeretnénk megbizonyosodni róla, hogy az a modell helyes, vagyis a specifikációban előírt funkciót hibátlanul valósítja meg. Ez tulajdonképpen nem más, mint az 5. fejezetben részletesen tárgyalt funkcionális verifikáció, amelynek kulcsfontosságú eszközei a HDL modellek dinamikus viselkedését „színlelni" képes szoftverek, a HDL szimulátorok.[21]
- **Automatizált logikai/RTL szintézis**: Ahogy az az 1. fejezetben bemutatott design-flow alapján is látható, a HDL nyelvű RTL modellekből automatizált úton állítjuk elő az RTL modellel logikailag azonos viselkedést megvalósító kapuszintű, flip-flopokból, memóriákból és logikai kapukból álló modelleket. Ezt a folyamatot hívjuk logikai vagy RTL szintézisnek,[22] amelyhez a HDL bemenet feldolgozására képes szoftver eszközöket, ún. szintézer szoftvereket használunk.

A HDL modellekre tehát mindig kétféle nézőpontból kell tekintenünk; egyrészt, hogy funkcionálisan mit írnak le az egyes nyelvi szerkezetek, másrészt, hogy „mi fog belőlük szintetizálódni". Ezt a két eltérő szempontot a HDL-ek „szimulációs és szintézis szemantikájának", vagyis jelentéstanának nevezzük. A nehézség abban rejlik, hogy a digitális áramkörtervezés alapgondolata, amely szerint a tervezés regiszter-transzfer szinten kezdődik *(design entry)*, és a rendelkezésre álló alapelemekből felépülő, „valódi" áramkör automatizált úton áll elő, csak akkor működik, ha az automatizáltan előállított kapuszintű modell valóban ugyanazt a funkciót valósítja meg, mint az RTL modell. A HDL-ek nyelvtani repertoárja sokkal szerteágazóbb annál, semhogy tetszőleges leírásból előállítható legyen az egyazon funkciót megvalósító, valódi digitális áramkör. Valójában a HDL nyelvi elemeknek csak kis része alkalmas arra, hogy automatizált áramkörszintézis alapját képezze, és ezeket a nyelvi szerkezeteket is igen körültekintően kell alkalmaznunk, ha jó minőségű, könnyen érthető, lehetőleg technológiafüggetlen[23] modellt akarunk készíteni. Fontos hangsúlyozni, hogy ennek nem az az elsődleges oka, hogy a szintézis szoftverek nem elég kifinomultak, hanem az, hogy a digitális áramköröket felépítő alapelemek nem túl sokfélék. Ahogy azt a 4-1. ábra is érzékelteti, a digitális áramkörök fizikailag tároló elemekből és kombinációs hálózatokból állnak, és még RTL modelljeikben is csak néhány alkatrésztípust tudunk megkülönböztetni; regiszterek, memóriák, aritmetikai/logikai elemek, multiplexerek és állapotgépek. Ezeknek a konstrukcióknak rendelkezésre állnak a megbízhatóan szintetizálható HDL-beli megfelelőik, és RTL tervezőként másra nincs is szükségünk.

A helyzetet nehezíti, hogy a különböző szintézis szoftverek hasonlóan, de nem feltétlenül teljesen azonosan kezelik a különböző nyelvi elemeket. Léteznek ugyan általános igazságok azzal kapcsolatban, hogy egy-egy áramköri elemet hogyan érdemes leírni (pontosan ezekre fogunk koncentrálni ebben a fejezetben), de a HDL modellek elkészítése előtt

[21] „Simulare": latin szó, jelentése színlel, lemásol, imitál.

[22] „Szüntheszisz" (görög), „synthesis" (latin): összegez, egységbe foglal.

[23] Technológiafüggetlenség alatt itt azt értjük, hogy a HDL modell egyformán alkalmas különböző gyártóktól származó programozható logikai eszközökön, vagy akár standard cellás ASIC technológián való megvalósításra.

mindig érdemes a használni kívánt szintézis szoftver ajánlásait, az ún. RTL kódolási szabályokat *(RTL coding guidelines)* áttanulmányozni.[24]

4.3 Alapvető adattípusok

Mindkét HDL többféle adattípus használatát teszi lehetővé, amelyek egy része szintetizálható, más részét csak szimulációs környezetben, viselkedési szintű modellekben vagy tesztkörnyezetek leírása során használhatjuk, mert automatizált szintézisük nem megoldott, vagy egyszerűen értelmezhetetlen (pl. VHDL: `real, time`). RTL leírásokban kiemelkedő jelentőségük van azoknak a típusoknak, amelyek a modellezett áramkör belső csomópontjainak, jeleinek állapotát reprezentálják. Ezek a típusok VHDL esetén az `std_logic` és annak több bites változata, az `std_logic_vector`, Verilog esetén pedig a `logic`. Tekintve, hogy digitális áramkörökről beszélünk, azt gondolhatnánk, hogy az e típusok valamelyikével deklarált jelek csak kétféle értéket, logikai alacsony és magas értéket vehetnek fel. A helyzet ennél kicsit bonyolultabb. Még ha feltételezzük is, hogy csak CMOS kapcsolástechnikán megvalósított áramköröket szeretnénk pontosan modellezni – ami tulajdonképpen igaz is, a HDL-eket ma szinte kizárólag olyan áramkörök modellezésére használjuk, amelyeket ezzel a technológiával valósítunk meg –, akkor is szükségünk van egy harmadik állapotra a háromállapotú *(tri-state)* buszok megfelelő leírásához (nagyimpedanciás állapot, 'Z'), a két lehetséges érték tehát nem elegendő. Ugyancsak le kell tudnunk írni azt a helyzetet, ha egy csomópont állapota nem értelmezhető, határozatlan. Ennek két oka lehet; a szimuláció kezdetén nem inicializáltuk megfelelően a jelet, vagy két különböző forrásból, két eltérő értéket kényszerítünk rá, ami – később részletezett okokból kifolyólag – az RTL modellekben igen könnyen előidézhető. Áramköri szempontból, tehát a szintézis szemantika szerint vizsgálva a dolgot ez azt jelentené, hogy összekötöttünk két flip-flop vagy kapukimenetet, amelyek közül az egyik alacsony, a másik magas szintet próbál kényszeríteni az adott csomópontra. Ez utóbbi helyzet nemcsak a szimulációs környezetben, hanem valós áramköri körülmények között is egyfajta határozatlan értéket eredményez; ha összekötünk két CMOS kapukimenetet, amelyek közül az egyik tápfeszültségre, a másik földpotenciálra húzza a csomópontot, akkor nagyjából a komparálási szint körüli potenciál adódik, ami logikailag értelmezhetetlen.[25] A Verilog `logic` típusa pontosan a fent említett négy állapotot képes leírni (0, 1, Z, X), míg a VHDL `std_logic` típusa ennél is tovább megy, összesen kilenc állapotot különböztet meg. Ennek oka az, hogy eredetileg a VHDL-t már létező, és nem feltétlenül CMOS kapcsolástechnikával megvalósított integrált áramkörök viselkedésének formális dokumentálására hozták létre, így képesnek kellett lennie megkülönböztetni a különböző típusú kapukimenetek (például TTL kapuk, passzív terhelésű MOS

[24] Ez persze csak akkor segít, ha tudjuk, hogy az adott modellt milyen szintézis szoftverrel fogjuk szintetizálni. Ez az esetek jelentős részében nem ismert, mert a bevezetőben is említett, IP-alapú tervezési alapelvnek megfelelően technológiafüggetlen, újrafelhasználható modellek előállítására törekszünk. Ilyenkor csak az e fejezetben is tárgyalt általános módszerekre és a korábbi tapasztalatainkra támaszkodhatunk.

[25] Nem beszélve arról, hogy egyébként a CMOS eszközeink egy ilyen helyzetben jó eséllyel tönkre is mennek, ha ez a helyzet sokáig fennáll, mert amit az imént körülírtunk, az valójában egy táp-föld rövidzár.

kapuk stb.) által előidézett állapotokat egy csomóponton (erős és gyenge fel/lehúzás). Külön kezeli továbbá az előbb említett inicializálatlan ('U', *undefined*) és határozatlan állapotokat, továbbá a logikai szempontból érdektelen, *„don't care"* állapotot. Az RTL tervezés során azonban e kilenc állapotból az esetek túlnyomó részében csak a Verilog `logic` típusa által is használt négyet, időnként ezeken kívül a *„don't care"* állapotot használjuk.

Míg a Verilog `logic` típusa – amit egyébként a SystemVerilog vezetett be – magának a nyelvnek is része, addig a VHDL `std_logic` típusa egy, a nyelvet kiegészítő, de annak nem szerves részét képező, ún. csomagban deklarált felsorolt típus. Emiatt a VHDL-ben leírt RTL modellek gyakorlatilag kivétel nélkül mind a 4-6. ábrán is látható könyvtár- és csomagdeklarációval kezdődnek.

VHDL

```
library ieee;
use ieee.std_logic_1164.all;
```

4-6. ábra. *Könyvtár- és csomagdeklarációk az `std_logic` típus használatához*

4.4 A HDL modellek szerkezete, utasítástípusok

Ahogy az objektumorientált programozási nyelvek esetén az osztály, úgy a HDL-ek esetén az „egyed" vagy „modul" a tervezés, egyúttal az áramköri hierarchia alapegysége. Ez az az absztrakció, amelyhez – mint egy önállóan értelmezhető áramköri modellhez – nyelvtani szinten is egyértelműen jelölt, a külső erőforrásokhoz való kapcsolódást lehetővé tévő interfészt és a belső működés részleteit leíró implementációt társítunk. VHDL-ben a tervezésnek ez az alapegysége az ún. „egyed-architektúra pár" (*entity-architecture pair*), Verilogban pedig a modul (*module*) (lásd 4-7. ábra).

VHDL

```
entity accumulator is

  -- interfészjelek deklarációja

end entity accumulator;

architecture rtl of accumulator is

  -- belső jelek deklarációja

begin

  -- belső működést leíró utasítások
  -- (implementáció)

end architecture rtl;
```

Verilog

```
module accumulator (

  // interfészjelek deklarációja

);

  // belső jelek deklarációja
  // belső működést leíró utasítások
  // (implementáció)

endmodule
```

4-7. ábra. *Egyed-architektúra pár deklarációja VHDL-ben, modul deklarációja Verilogban*

A Veriloggal ellentétben a VHDL lehetővé teszi, hogy egy interfészhez több implementációt, vagyis egy egyeddeklarációhoz több architektúrát rendeljünk. Ezért van az, hogy a VHDL leírásban az architektúrának is külön nevet kell adnunk (a példában az architektúra neve `rtl`). Ha egy interfészhez több implementációt is létrehozunk, akkor a modul

beültetésénél, példányosításánál kell nyilatkoznunk arról, hogy az adott környezetben melyik implementációt akarjuk használni.

Az áramkör interfésze tulajdonképpen azoknak a jeleknek a listája (portlista), amelyek „kívülről is láthatók”[26] (lásd 4-8. és 4-9. ábra), az áramkör csak e jeleken keresztül képes kölcsönhatásba lépni a külvilággal.

VHDL

```vhdl
entity accumulator is
  port (
    clk:    in  std_logic;                       -- órajel-bemenet
    rst:    in  std_logic;                       -- reset-bemenet
    A:      in  std_logic_vector (31 downto 0);  -- "A" operandus-bemenet
    B:      in  std_logic_vector (31 downto 0);  -- "B" operandus-bemenet
    C:      in  std_logic_vector (31 downto 0);  -- "C" operandus-bemenet
    D:      in  std_logic_vector (31 downto 0);  -- "D" operandus-bemenet
    SUM:    out std_logic_vector (31 downto 0);  -- "SUM" kimenet
    Z:      out std_logic                        -- "Z" kimenet
  );
end entity accumulator;
```

4-8. ábra. *Portlista VHDL-ben.*

Verilog

```verilog
module accumulator (
  input   logic         clk,  // órajel-bemenet
  input   logic         rst,  // reset-bemenet
  input   logic [31:0]  A,    // "A" operandus-bemenet
  input   logic [31:0]  B,    // "B" operandus-bemenet
  input   logic [31:0]  C,    // "C" operandus-bemenet
  input   logic [31:0]  D,    // "D" operandus-bemenet
  output  logic [31:0]  SUM,  // "SUM" kimenet
  output  logic         Z     // "Z" kimenet
);
```

4-9. ábra. *Portlista Verilogban*

A portlistán az interfészjelek nevét, irányát, típusát, valamint több bites jelek, vagyis buszok esetén azok méretét kell definiálnunk a legfelső és legalsó helyiértékű bitek indexeinek megadásával.[27] A buszok indexelésénél a legelterjedtebb megoldás, hogy az MSB-hez rendeljük a legnagyobb indexet, ahogy a fenti ábrákon is látható, bár esetenként találkozhatunk a fordított megoldással is (pl. Xilinx MicroBlaze rendszer).

A HDL modellekben az interfész-deklarációt az implementációs rész követi. A modul implementációja tulajdonképpen HDL utasítások halmaza, amely a modul belső szerkezetét és részegységeinek pontos viselkedését hivatott kifejezni. Ezek az utasítások három kategóriába sorolhatók: folytonos értékadás, példányosítás és folyamat.

4.4.1 Folytonos értékadás

Az értékadások „folytonos” jellege a HDL-ek szimulációs szemantikája szempontjából azt jelenti, hogy ezek az értékadások mindig kiértékelődnek, amikor a szimuláció során a jobbértékben valamilyen változás következik be, tehát tulajdonképpen a kifejezés két

[26] „Inter-”: latin elöljárószó, jelentése között, közé; „facies”: ugyancsak latin eredetű, jelentése arc.

[27] Az ebben a fejezetben bemutatott VHDL példák esetén az egyeddeklarációk előtti csomagdeklarációkat és a modulok interfészjeleinek deklarációit csak akkor tüntetjük fel, ha hiányuk nehezítené a példa megértését.

oldala között fennálló állandó hozzárendelést fejeznek ki. Szintézis szempontból ez egy állandó logikai kapcsolatot jelent a jobbérték-kifejezésben szereplő jelek és a balértéket képviselő jel között. Ezt az utasítástípust rendszerint közvetlen aritmetikai és logikai kapcsolatok kifejezésére, valamint multiplexerek leírására használjuk. VHDL-ben az egyszerű konkurens értékadás, valamint a `when` és a `with-select` utasítások, Verilogban az `assign` utasítás képviseli. A 4-10. ábra egy félösszeadót modellez az abban található két logikai kapu egy-egy folytonos értékadással való leírásával.

VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity half_adder is
  port (
    a, b:  in  std_logic;
    s, c:  out std_logic
  );
end entity half_adder;

architecture logic of half_adder is
begin

  s <= a xor b;
  c <= a and b;

end architecture logic;
```

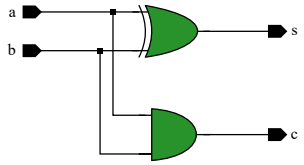
Verilog

```
module half_adder (
  input  logic a, b,
  output logic s, c
);

  assign s = a ^ b;
  assign c = a & b;

endmodule
```

Áramköri struktúra



4-10. ábra. *Félösszeadó leírása folytonos értékadásokkal*

A VHDL és a Verilog közötti különbség a logikai kapcsolatok kifejezésében a fenti példából jól látható; míg a VHDL angol nyelvű kulcsszavakat használ, addig a Verilog szimbólumokat. Ez persze csak formai eltérés, mindkét nyelvben rendelkezésre állnak a szükséges műveleteknek megfelelő nyelvi elemek (lásd 4-1. táblázat).

Logikai kapcsolat	VHDL	Verilog
ÉS	`c <= a and b;`	`assign c = a & b;`
VAGY	`c <= a or b;`	`assign c = a | b;`
NEM	`b <= not a;`	`assign b = ~a;`
NEM ÉS	`c <= not (a and b);`	`assign c = ~(a & b);`
NEM VAGY	`c <= not (a or b);`	`assign c = ~(a | b);`
antivalencia	`c <= a xor b;`	`assign c = a ^ b;`
ekvivalencia	`c <= a xnor b;`	`assign c = a ~^ b;`

4-1. táblázat. *Logikai kapcsolatok kifejezése*

A logikai műveletek egy folytonos értékadás jobbérték-kifejezésében természetesen tetszőlegesen kombinálhatók is, így összetettebb logikai kapcsolatokat is ki tudunk fejezni egyetlen utasítás segítségével.

A 4-11. ábra a VHDL `when` és `with-select` utasításaira mutat példát, amelyek Verilogbeli megfelelője a `?:` operátor.

VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity some_logic is
  port (
    a, b, s1, s2: in  std_logic;
    y:            out std_logic
  );
end entity some_logic;

architecture logic of some_logic is

  -- A 'c' csomópont a modell belső jele,
  -- amit deklarálnunk kell:
  signal c: std_logic;

begin

  c <= (a and b) when s1 = '1' else (a xor b);
  with s2 select y <= c when '0', b when others;

end architecture logic;
```

Verilog

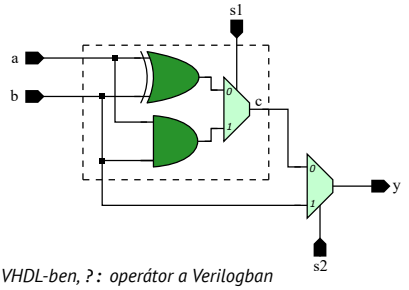
```
module some_logic (
  input  logic a, b, s1, s2,
  output logic y
);

  // A 'c' csomópont a modell belső jele,
  // amit deklarálnunk kell:
  logic c;

  assign c = (s1 == 1'b1) ? (a & b) : (a ^ b);
  assign y = (s2 == 1'b0) ? c : b;

endmodule
```

Áramköri struktúra



4-11. ábra. *when és* `with-select` *utasítások a VHDL-ben,* `?:` *operátor a Verilogban*

A feltételes folytonos értékadások segítségével a háromállapotú meghajtók viselkedése is egyszerűen kifejezhető (lásd 4-12. ábra).

VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity tribuffer is
  port (
    ena:  in  std_logic;
    a:    in  std_logic;
    y:    out std_logic
  );
end entity tribuffer;

architecture logic of tribuffer is
begin

  y <= a when ena = '1' else 'Z';

end architecture logic;
```

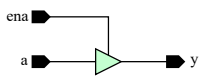
Verilog

```
module tribuffer (
  input  logic ena,
  input  logic a,
  output logic y
);

  assign y = ( ena == 1'b1 ) ? a : 1'bz;

endmodule
```

Áramköri struktúra



4-12. ábra. *Háromállapotú meghajtó leírása feltételes folytonos értékadással*

Ahogy egy viselkedési modellbeli, például C++ függvényre sem jellemző, hogy bemenetei és kimenetei között egyetlen kifejezés segítségével kapcsolatot teremt, úgy a HDL modellekben is szükség van belső „változókra”, ahogy az a 4-11. ábrán is megfigyelhető. A programozási nyelvekben hozzászoktunk, hogy egy függvény lokális változói valamilyen számítás részeredményeit tárolják. A HDL modellek azonban áramköröket írnak le, ennek megfelelően a belső változókra inkább részáramkörök által meghajtott belső csomópontokként,

jelekként kell gondolnunk. A VHDL e belső jeleket `signal`-nak hívja. Áramkörileg egy `signal` jelenthet egy egyszerű vezetéket – ahogy az a 4-11. ábra példájában is látható – vagy egy flip-flopot attól függően, hogy pontosan hogyan használjuk, milyen utasítás segítségével adunk neki értéket. Ha folytonos értékadás vagy egy almodul egy példánya hajtja meg, akkor vezeték, ha egy ún. „szinkron folyamat" (lásd később), akkor flip-flop. A Verilog korai verziói ezzel ellentétben már nyelvi szinten szétválasztották ezt a két fogalmat; a 4-11. ábrán láthatóhoz hasonlóan, folytonosan meghajtott belső változókra a `wire`, a szinkron folyamat által meghajtottakra a `reg` kulcsszót használta. A nyelv legújabb változata (SystemVerilog) ezt a különbségtételt a `logic` típus bevezetésével megszüntette, de a visszafelé való kompatibilitás miatt a `wire` és `reg` kulcsszavakat továbbra is használhatjuk.

A 4-11. ábra kódrészleteiben a konstansok használatára is láthatunk példát. A logikai jelekhez rendelhető, VHDL-ben `std_logic` vagy `std_logic_vector`, Verilogban pedig a `logic` típusú jelekhez rendelhető konstansok használatát a 4-2. táblázat foglalja össze.

Balérték típusa	VHDL	Verilog
Egybites jel	`a <= '0';` `a <= '1';`	`assign a = 1'b0;` `assign a = 1'b1;`

Több bites busz	`a <= "1011";`
	`a <= B"1011";`
	`a <= X"B";`
		`assign a = 4'b1011;`
		`assign a = 4'd11;`
		`assign a = 4'hB;`

4-2. táblázat. *Logikai típusú konstansok használata*

4.4.2 Példányosítás

Mindkét nyelv lehetőséget biztosít arra, hogy egy modulon belül másik modulok példányait helyezzük el és azok interfészéhez, portjaihoz saját belső jeleinket és portjainkat kapcsoljuk. A 4-13. ábrán látható példában két félösszeadó példányosításával hoztunk létre egy teljes összeadót.

VHDL-ben a példányosítás utasításnak több formája is van, a 4-13. ábrán bemutatott megoldás az ún. közvetlen beültetés. Itt meg kell adnunk a példány egyedi azonosítóját (`HA1`, `HA2`), a könyvtár nevét, amelybe az adott modult lefordítottuk (`work`, ez a VHDL szimulátorok alapértelmezett könyvtárneve), a példányosítani kívánt egyed azonosítóját (`half_adder`) és az architektúra nevét (`logic`). Ha az adott egyedhez csak egyetlen architektúrát definiáltunk, akkor ez utóbbi elhagyható. A modul azonosítása után következik a bekötési lista (`port map`), amelyben azt kell meghatároznunk, hogy a modul ki és bemeneteihez a befoglaló modul melyik jelét vagy portját akarjuk hozzákapcsolni. A Verilog modellben lényegében ugyanezt tesszük, csak némileg eltérő sorrendben; azonosítjuk a példányosítandó modult (`half_adder`), a példányt ellátjuk egy egyedi azonosítóval (`HA1`, `HA2`), majd megadjuk a bekötési listát is. Név szerinti porthozzárendelésnek hívjuk azt, amikor a modul portjaihoz hozzárendelendő jeleket a 4-13. ábrán látható módon úgy adjuk meg, hogy néven nevezzük először a portot, majd megadjuk a hozzárendelendő jelnevet. Ehelyett alkalmazhatunk pozíció szerinti porthozzárendelést is, ekkor a modul portjait nem kell megneveznünk, csak egy jellistát adunk meg, ahogy a 4-14. ábrán látható.

VHDL

```
entity full_adder is
  port (
    a, b, c_in: in  std_logic;
    s, c_out:   out std_logic
  );
end entity full_adder;

architecture logic of full_adder is

  signal s1, c1, c2:  std_logic;

begin

  HA1:  entity work.half_adder(logic)
          port map (a => a,
                    b => b,
                    s => s1,
                    c => c1);

  HA2:  entity work.half_adder(logic)
          port map (a => c_in,
                    b => s1,
                    s => s,
                    c => c2);

  c_out <= c2 or c1;

end architecture logic;
```

Verilog

```
module full_adder (
  input  logic a, b, c_in,
  output logic c_out, s
);

  logic s1, c1, c2;

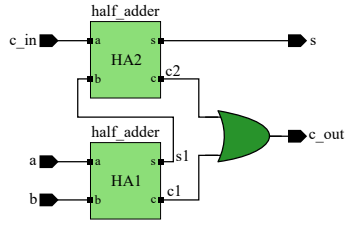
  half_adder HA2(.a(c_in),
                 .b(s1),
                 .s(s),
                 .c(c2));

  half_adder HA1(.a(a),
                 .b(b),
                 .s(s1),
                 .c(c1));

  assign c_out = c2 | c1;

endmodule
```

Áramköri struktúra



4-13. ábra. *Példányosítás név szerinti porthozzárendeléssel*

VHDL

```
entity full_adder is
  port (
    a, b, c_in: in  std_logic;
    s, c_out:   out std_logic
  );
end entity full_adder;

architecture logic of full_adder is

  signal s1:  std_logic;
  signal c1:  std_logic;
  signal c2:  std_logic;

begin

  HA1:  entity work.half_adder(logic)
          port map (a,b,s1,c1);

  HA2:  entity work.half_adder(logic)
          port map (c_in,s1,s,c2);

  c_out <= c2 or c1;

end architecture logic;
```

Verilog

```
module full_adder (
  input  logic a, b, c_in,
  output logic c_out, s
);

  logic s1;
  logic c1;
  logic c2;

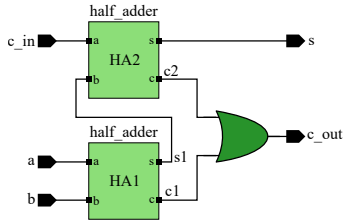
  half_adder HA2(c_in,s1,s,c2);

  half_adder HA1(a,b,s1,c1);

  assign c_out = c2 ^ c1;

endmodule
```

Áramköri struktúra



4-14. ábra. *Példányosítás pozíció szerinti porthozzárendeléssel*

Pozíció szerinti porthozzárendelés esetén a példányosított modulra kapcsolandó külső jeleket a modul portlistájának megfelelő sorrendben kell felsorolni, hasonlóan ahhoz, mint amikor egy C függvény meghívásánál felsoroljuk annak paramétereit a függvénydeklarációnak megfelelő sorrendben. A pozíció szerinti porthozzárendelés tömörebb ugyan, mint a név szerinti, ennek ellenére célszerű mégis a név szerinti megoldást használni, mert sok ki- és bemenettel rendelkező modulok esetén a portlista utólagos változtatása ebben az esetben egyszerűbb. Egy másik érv a név szerinti megoldás mellett, hogy a szintézis szoftverek, amelyek a szintetizált netlistáról HDL modellt is képesek készíteni annak érdekében, hogy azt az RTL modell helyett példányosítva megvizsgáljuk az áramkör működését logikai szimulációval, hajlamosak olyan HDL modellt készíteni, amelynek portlistája megegyezik ugyan az RTL modell portlistájával, de a portok más sorrendben vannak felsorolva benne. Ha pozíció szerinti porthozzárendeléssel példányosítottuk a modellünket a tesztkörnyezetben, akkor a netlista szimulációjához azt gyakorlatilag újra meg kell csinálnunk, az ezzel járó fájdalom pedig a portok számának növekedésével exponenciálisan nő.

4.4.3 Folyamat

A folyamatok a HDL-ek leghatékonyabb nyelvi szerkezetei. Lehetővé teszik, hogy egy áramköri részlet működését valamilyen jelen/jeleken bekövetkező változás hatására végrehajtott, meghatározott sorrendben „lefutó" utasítássorozat segítségével, szekvenciálisan írjuk le. VHDL-ben a folyamat fogalmát az ún. `process`, Verilogban pedig az ún. `always`-blokk valósítja meg. A folyamatok kombinációs és szekvenciális logikák leírására egyaránt alkalmasak; az előbbire kombinációs folyamatként, az utóbbira órajelvezérelt, órajelre érzékenyített vagy egyszerűen csak szinkron folyamatként fogunk hivatkozni. A 4-15. ábrán látható példák azt mutatják be, hogy a 4-11. ábrán folytonos értékadásokkal megvalósított kombinációs logikát miként lehet kombinációs folyamatok segítségével leírni.

A folyamatutasítások a VHDL-ben és a Verilogban egyaránt egy érzékenységi listából és egy folyamattörzsből állnak. A szimuláció során, ha az érzékenységi lista bármely jelén változás következik be, akkor a folyamat törzsében lévő utasítások sorban kiértékelődnek. Kombinációs folyamatok esetén fontos, hogy a modellezett kombinációs hálózat összes bemenetét felsoroljuk az érzékenységi listán,[28] hiszen így modellezzük pontosan egy kombinációs hálózat viselkedését: Egy kombinációs hálózat kimenete bármely bemenet megváltozásának hatására megváltozhat. A folyamattörzs utasításainak kiértékelése egyetlen szimulációs ciklus alatt megy végbe, miközben a szimulációs idő nem nő, tehát a kiértékelés folyamatát a kiértékelt utasítások számától függetlenül egyetlen eseménynek kell tekintenünk. A folyamat utasítások hatékonysága legfőképp annak köszönhető, hogy

[28] Ez a kijelentés a nyelvek legújabb verzióira valójában már nem igaz, új nyelvi elemek bevezetésével a VHDL 2008 és a SystemVerilog egyaránt megoldotta, hogy a bemeneteket ne kelljen felsorolni. Ez a követelmény ti. redundáns információ kézi bevitelét jelentette (a folyamat törzséből is egyértelműen kiderül, hogy mi lesz bemenet és mi kimenet), ami hibalehetőséget rejtett. A szintézis szoftverekben ezek az új nyelvi elemek (VHDL: `process (all)`, Verilog: `always_comb`) nem teljesen támogatottak, ezért az itt bemutatott példákban nem használjuk őket.

VHDL

```
architecture logic of some_logic is

  signal c: std_logic;

begin

  process ( a, b, s1 )
  begin
    if ( s1 = '0' ) then
      c <= a xor b;
    else
      c <= a and b;
    end if;
  end process;

  process ( b, c, s2 )
  begin
    if ( s2 = '0' ) then
      y <= c;
    else
      y <= b;
    end if;
  end process;

end architecture logic;
```

Verilog

```
module some_logic (...);

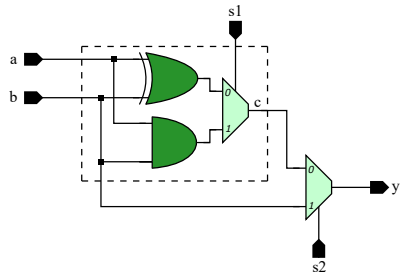
  logic c;

  always @ ( a, b, s1 )
  begin
    if ( s1 == 0 ) c <= a ^ b;
    else c <= a & b;
  end

  always @ ( b, c, s2 )
  begin
    if ( s2 == 0 ) y <= c;
    else y <= b;
  end

endmodule
```

Áramköri struktúra



4-15. ábra. *Kombinációs folyamatok*

VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity vector_merge is
  port (
    a, b: in  std_logic_vector (3 downto 0);
    y:    out std_logic_vector (7 downto 0)
  );
end entity vector_merge;

architecture logic of vector_merge is
begin

    process ( a, b )
    begin
      for i in 0 to 3 loop
        y(2*i) <= not a(i);
        y(2*i+1) <= not b(i);
      end loop;
    end process;

end architecture logic;
```

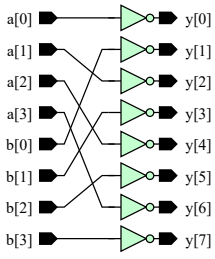
Verilog

```
module vector_merge (
  input  logic [3:0] a, b,
  output logic [7:0] y
);

  always @ ( a, b )
  begin
    for ( int i = 0; i <= 3; i = i+1 )
    begin
      y[2*i] <= ~a[i];
      y[2*i+1] <= ~b[i];
    end
  end

endmodule
```

Áramköri struktúra



4-16. ábra. *Ciklusok használata*

a törzsükben a programozási nyelvekből is ismert vezérlő utasítások, az `if-else`, a `case` utasítások, sőt ciklusok is használhatók (lásd 4-16. ábra).

Szekvenciális hálózatok működését a folyamatok egy speciális típusával, az ún. szinkron folyamattal írhatjuk le. A 4-17. ábrán látható példák szinkron folyamat segítségével írják le a D flip-flop működését.

VHDL

```
architecture rtl of d_flip_flop is
begin

  process ( clk )
  begin
    if ( rising_edge(clk) ) then
      q <= d;
    end if;
  end process;

end architecture rtl;
```

Verilog

```
module d_flip_flop (...);

  always @ ( posedge clk )
  begin
    q <= d;
  end

endmodule
```

4-17. ábra. *D flip-flop leírása VHDL-ben és Verilogban szinkron folyamattal*

A szinkron folyamatokra jellemző, hogy érzékenységi listájuk csak az órajelet és a modellezett tároló elem aszinkron bemeneteit – például aszinkron preset/reset – tartalmazzák, függetlenül attól, hogy a leírt részáramkörnek hány bemenete van. Ez a D flip-flop fenti leírásaiban is megfigyelhető; a `d` bemenet nem része az érzékenységi listának, hiszen a D flip-flop csak akkor változtatja meg kimenetének értékét, ha az órajel nullából egybe vált. A `d` bemenet változásai nem hatnak azonnal az áramkör kimenetére, tehát nem kell hogy a folyamattörzs utasításainak kiértékelését váltsák ki, így a szimulációs modell viselkedése tökéletesen megfelel a szintézis során előálló áramkör viselkedésének.[29]

Általános esetben egy szinkron folyamat nemcsak a flip-flop, hanem az annak bemenetét meghajtó, tetszőlegesen bonyolult kombinációs hálózat viselkedését is leírja. Ezt szemlélteti a 4-18. ábrán látható példa, amely szinkron működésű teljes összeadót ír le.

Ezekben a példákban egy-egy szinkron folyamatban csak egyetlen tároló elemnek határoztuk meg az értékét, minden folyamat csak egy flip-flopot „hajtott meg". Általános esetben ez nincs így, egy folyamat tetszőleges számú flip-flopot meghajthat (lásd 4-19. ábra).

Egy folyamatban tehát adhatunk értéket több jelnek, de ez visszafelé nem igaz; nem adhatunk értéket ugyanannak a jelnek több különböző folyamatban. Egy jelnek valójában az egész HDL modellben csak egyetlen utasításban (folytonos értékadásban, példányosításban vagy folyamatban) adhatunk értéket, más szóval egy jelnek csak egyetlen meghajtója lehet. Ha egy jelnek két különböző utasításban két eltérő értéket adunk, akkor az eredmény a 4.3. pontban említett határozatlan állapot lesz.[30] Ez áramköri szempontból – továbbra is CMOS kapcsolástechnikát feltételezve – azt jelenti, hogy egy csomópont potenciálját csak egy áramköri elem, egy kapu- vagy flip-flop-kimenet határozhatja meg, tehát áramköri elemek kimeneteit nem köthetjük össze. Kivételt jelent ez alól a háromállapotú buszok esete.

[29] Ezek alapján érthető meg igazán, hogy miért „folytonos" a folytonos értékadás. A szinkron folyamatokban is értékadásokat írunk le, de ezek nem folytonosak, csak az órajel felfutó élénél történnek meg.

[30] VHDL-ben ez valójában csak az úgynevezett „feloldott" (resolved) jeltípusokra igaz, mint például az `std_logic`. Nem feloldott jelek esetén a többszörös meghajtás fordítási idejű hibát jelent.

VHDL

```
entity sync_full_adder is
  port (...);
end entity sync_full_adder;

architecture rtl of sync_full_adder is
begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      s <= '0';
    elsif ( rising_edge(clk) ) then
      s <= (a xor b) xor c_in;
    end if;
  end process;

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      c_out <= '0';
    elsif ( rising_edge(clk) ) then
      c_out <= ((a xor b) and c_in) or (a and b);
    end if;
  end process;

end architecture rtl;
```

Verilog

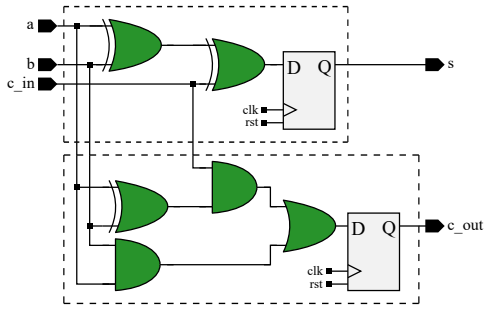
```
module sync_full_adder (...);

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) s <= 1'b0;
    else s <= (a ^ b) ^ c_in;
  end

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) c_out <= 1'b0;
    else c_out <= ((a ^ b) & c_in) | (a & b);
  end

endmodule
```

Áramköri struktúra



4-18. ábra. *Szinkron működésű teljes összeadó aszinkron resettel (1)*

VHDL

```
architecture rtl of sync_full_adder is
begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      s <= '0';
      c_out <= '0';
    elsif ( rising_edge(clk) ) then
      s <= (a xor b) xor c_in;
      c_out <= ((a xor b) and c_in) or (a and b);
    end if;
  end process;

end architecture rtl;
```

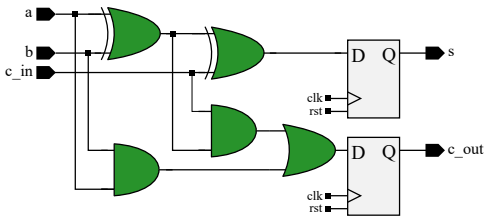
Verilog

```
module sync_full_adder (...);

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) begin
      s <= 1'b0;
      c_out <= 1'b0;
    end else begin
      s <= (a ^ b) ^ c_in;
      c_out <= ((a ^ b) & c_in) | (a & b);
    end
  end

endmodule
```

Áramköri struktúra



4-19. ábra. *Szinkron működésű teljes összeadó aszinkron resettel (2)*

4.4.4 Egyidejűség és sorrendiség a HDL-ekben

A fenti példák a HDL-eknek egy fontos, a programozási nyelveket készségszinten használóknak elsőre talán nehezen érthető sajátosságát is igyekeznek szemléltetni, nevezetesen hogy a modul belsejét modellező utasítások között nem értelmezhető semmiféle sorrendiség. A folytonos értékadások, példányosítások és folyamatok egymással párhuzamosan működnek, teljesen mindegy, hogy a HDL modellben melyiket írjuk le előbb. A modul HDL modelljére úgy érdemes gondolni, mint egy „szövegesen megfogalmazott kapcsolási rajzra", amelyben minden részáramkört egy-egy HDL kódrészlet, utasítás reprezentál. Ugyanúgy, ahogy egy kapcsolási rajz összetevői között, úgy az azoknak megfeleltethető HDL utasítások között sincs értelme sorrendiségről beszélni. E megközelítés szemléltetése céljából a 4-20. ábra a 4-5. ábrán látható, iteratív akkumulátor áramkör VHDL és Verilog leírását mutatja be. A külön HDL utasítással leírt részáramköröket szaggatott vonallal jelöltük.

Az utasítások közötti sorrendiség hiánya okán a HDL modellek implementációs részét konkurens utasításblokknak is nevezzük. Ugyanakkor – ahogy azt a 4.4.3. pontban is állítottuk – a folyamatok belsejében lévő utasítások a szimuláció során sorban értékelődnek ki, ezek tehát sorrendi, más néven szekvenciális utasításblokkok. Ez a sorrendiség azonban még mindig nem teljesen ugyanazt jelenti, mint amit a programozási nyelvekben megszokhattunk. A 4-19. ábrán látható példában egy folyamaton belül két jelnek is értéket adtunk, viszont e két jel egymástól teljesen független volt. Vizsgáljunk meg egy olyan esetet, amikor ez nem igaz (lásd 4-21. ábra). A folyamatok korábban ismertetett működése alapján találjuk ki, hogy mit is fejeznek ki a fenti HDL modellek: A reset jel (`rst`) felfutó élénél a helyzet egyszerű, a részösszeget (`subsum`) és a végösszeget (`SUM`) reprezentáló jel/port egyaránt nullát kap értékül. Ha a reset bemenet nulla értékű, és az órajelen megjelenik egy felfutó él, akkor a részösszeg megkapja az A és B bemenetek összegét. A végösszeg esetén bajban vagyunk, mert bár tudjuk, hogy a folyamatbeli utasításoknak sorban kell kiértékelődniük, ami azt jelentené, hogy a végösszeg az A, B és C bemenetek összegét fogja értékül kapni, ugyanakkor a folyamatokról azt is tudjuk, hogy a bennük lévő utasítások egyetlen szimulációs ciklus alatt értékelődnek ki, mialatt a szimulációs idő nem változik, tehát a sorrendben való végrehajtást egyetlen eseménynek kell tekintenünk. Így a végösszegnek az éppen ugyanebben a szimulációs időpillanatban megváltozó részösszeg és a stabilnak feltételezhető C bemenet összegét kellene értékül kapnia. Ezt a látszólagos ellentmondást a VHDL és a Verilog szimulációs szemantikája eltérően magyarázza:

- A VHDL definiálja az ún. „delta"-késleltetést, ami azt jelenti, hogy ha egy esemény hatására – például az órajel felfutó éle – értéket adunk egy jelnek (`signal`), akkor ez az értékadás nem azonnal, hanem egy infinitezimálisan (végtelenül) kis késleltetés után, gyakorlatilag a szimulációs ciklus legvégén kapja meg új értékét. A fenti helyzetben ez azt jelenti, hogy amikor a végrehajtás eléri a részösszegre vonatkozó értékadást, akkor a jobbérték-kifejezés ugyan kiértékelődik, de a részösszeg jel maga még nem kapja meg az új értékét, tehát amikor a végrehajtás a következő utasításhoz, a végösszegre vonatkozó értékadáshoz ér, akkor annak jobbérték-kiértékelésénél a szimulátor „még az előző részösszeget látja".

VHDL

```
architecture rtl of accumulator is
  signal iterator: std_logic_vector (1 downto 0);
begin

  Z <= '1' when SUM = X"00000000" else '0';

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      SUM <= (others => '0');
    elsif ( rising_edge(clk) ) then
      if ( start = '1' ) then
        SUM <= A;
      else
        case iterator is
          when "00"  =>  SUM <= SUM + B;
          when "01"  =>  SUM <= SUM + C;
          when "10"  =>  SUM <= SUM + D;
          when others => SUM <= SUM + 0;
        end case;
      end if;
    end if;
  end process;

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      iterator <= "00";
    elsif ( rising_edge(clk) ) then
      if ( start = '1' ) then
        iterator <= "00";
      elsif ( iterator < 3 ) then
        iterator <= iterator + 1;
      end if;
    end if;
  end process;

  finish <= '0' when iterator < 3 else '1';

end architecture rtl;
```

Verilog

```
module accumulator (...);

  logic [1:0] iterator;

  assign Z = (SUM == 32'd0) ? 1'b1 : 1'b0;

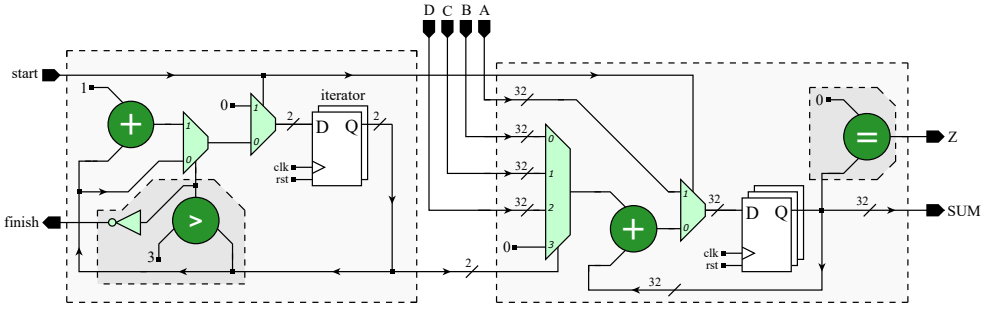
  always @ (posedge clk, posedge rst)
  begin
    if ( rst == 1'b1 ) SUM <= 32'd0;
    else
      if ( start == 1'b1 ) SUM <= A;
      else
        case (iterator)
          2'b00:    SUM <= SUM + B;
          2'b01:    SUM <= SUM + C;
          2'b10:    SUM <= SUM + D;
          default:  SUM <= SUM + 32'd0;
        endcase
  end

  always @ (posedge clk, posedge rst)
  begin
    if ( rst == 1'b1 ) iterator <= 2'b00;
    else
      if ( start == 1'b1 ) iterator <= 2'd0;
      else
        if ( iterator < 2'd3 )
          iterator <= iterator + 1'b1;
  end

  assign finish = (iterator < 2'd3) ? 1'b0 : 1'b1;

endmodule
```

Áramköri struktúra



4-20. ábra. *Iteratív akkumulátor VHDL-ben és Verilogban*

- A Verilog nem használja a delta-késleltetés fogalmát, helyette a <= szimbólum jelentése alapján ad – szerencsére a VHDL-lel egyező eredménnyel járó – magyarázatot a helyzetre. A <= szimbólum egy ún. nem-blokkoló *(non-blocking)* értékadást jelképez, ami azt jelenti, hogy az értékadás kiértékelése nem blokkolja a kiértékelési sort, tehát amikor a szimuláció eléri az n+1-edik értékadást, akkor „még nem láthatja” az n-edik értékadás eredményét, hiszen az még folyamatban van.

VHDL

```vhdl
architecture rtl of accumulator is

  signal subsum:  std_logic_vector (7 downto 0);

begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      subsum <= (others => '0');
      SUM <= (others => '0');
    elsif ( rising_edge(clk) ) then
      subsum <= A + B;
      SUM <= subsum + C;
    end if;
  end process;

end architecture;
```

Verilog

```verilog
module accumulator (...);

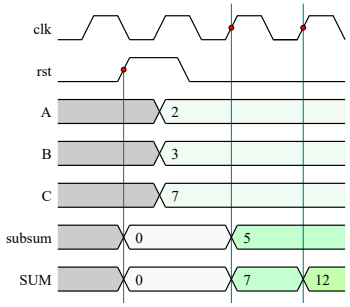
  logic [7:0] subsum;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) begin
      subsum <= 8'd0;
      SUM <= 8'd0;
    end else begin
      subsum <= A + B;
      SUM <= subsum + C;
    end
  end

endmodule
```

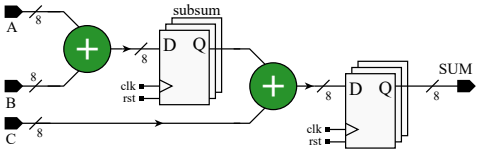
4-21. ábra. *Két jel-értékadás egy folyamaton belül*

Az eredmény tehát az eltérő magyarázatok ellenére ugyanaz; amikor jön egy felfutó él az órajelen, akkor a részösszeg megkapja az A és B bemenetek összegét, a végösszeg azonban az előző szimulációs ciklus végén érvényes részösszeg és a stabil C bemenet összegét kapja értékül, így a három bemenet összege végül csak az órajel második felfutó élénél fog előállni (lásd 4-22. ábra).



4-22. ábra. *A 4-21. ábrán látható HDL-leírás időzítési modellje*

Láttuk tehát, hogy a VHDL és a Verilog szimulációs szemantikája hogyan magyarázza meg ezt az elsőre furcsának tűnő helyzetet, most nézzük, mit jelent mindez áramköri szempontból. A 4-21. ábrán azt látjuk, hogy egy szinkron folyamatban két, 8-bites jelnek adunk értéket. Könnyen belátható, hogy ez két, 8-bites regisztert, és az azok bemeneteit meghajtó kombinációs logikát fog eredményezni a szintézis során (lásd 4-23. ábra).



4-23. ábra. *A 4-21. ábrán látható HDL modellekből szintetizált áramkör*

Ennek az áramkörnek a viselkedése remekül összecseng azzal a viselkedéssel, amit a HDL-ek szimulációs szemantikája is körülír; ha a részösszeget tároló regiszterbe az órajel felfutó élénél töltünk új értéket, akkor ugyanannál a felfutó élnél a végösszeget tároló regiszter új értékét előállító kombinációs hálózat bemenetén még a részösszeg régi értéke lesz. A VHDL delta-késleltetése és a Verilog nem-blokkoló értékadása tehát áramköri szempontból tulajdonképpen az egyes komponensek különböző késleltetéseit – a flip-flopok kimeneti *(clock-to-output)* késleltetését és a kombinációs hálózatok terjedési késleltetését – modellezi. Ezeknek a késleltetéseknek persze ezen az elvonatkoztatási szinten a pontos értékeit még nem ismerhetjük, de azt biztosan tudhatjuk, hogy nullánál nagyobbak lesznek. Ilyen értelemben a delta-késleltetés és a nem-blokkoló értékadás valójában nem időt, hanem csupán ok-okozati viszonyt képes modellezni. Vegyük észre, hogy a fenti gondolatmenet hallgatólagosan az órajelhálózattal szemben is megfogalmaz egy fontos követelményt; nevezetesen azt, hogy az órajelforrás és a flip-flopok órajel-bemenete közötti késleltetés bármelyik flip-flopra ugyanannyi, minden flip-flop pontosan ugyanakkor „érzékeli" az órajel felfutó élét, tehát az órajelelcsúszás *(clock skew)* bármely két, szomszédos flip-flopra értelmezve nulla. Ez a fizikailag megvalósított áramkörökben természetesen nem igaz, de ennek az optimális helyzetnek az elérése magának a fizikai tervezésnek az egyik legfontosabb célkitűzése. RTL tervezőként azonban mindig ideális órajelhálózatot feltételezünk, tehát elegendő azt tudnunk, hogy a kombinációs hálózatoknak és flip-flopoknak véges késleltetése van, az órajelhálózat pedig ideális, tehát nulla késleltetésű. A 4-22. ábrán látható hullámformán az órajel harmadik felfutó élénél azt láthatjuk, hogy a `SUM` regiszter az éppen változó `subsum` regiszterből valamilyen kombinációs hálózat – jelen esetben egy összedó – által előállított, látszólag instabil bemenetet mintavételez, ami a flip-flopok működésének ismeretében hibának tűnhet. Mivel azonban a `subsum` regiszter megváltozását és a `SUM` regiszterben a mintavételezést előidéző órajel-felfutóél ugyanaz, ezért valójában a `SUM` regiszter bemenetén a mintavételezés pillanatában a `subsum` regiszter változása még nem éreztetheti a hatását a delta-késleltetés által modellezett véges áramköri késleltetés miatt, a `SUM` regiszter tehát a `subsum` regiszter előző, még stabil értékét „látja".

A delta-késleltetés és a nem-blokkoló értékadás végső soron tehát azt eredményezte, hogy beiktattunk a jelútba egy regiszterréteget, ami nyilván a modellezett áramkörnek az erőforrásigényére és ütemezési tulajdonságaira is hatással van. De mi van akkor, ha valójában nem áll szándékunkban ezt a regiszterréteget beiktatni? A 4-21. ábrán látható kódrészletek minimális módosításával egy egészen más modellt is elő tudunk állítani (lásd 4-24. ábra).

VHDL

```
architecture rtl of accumulator is
begin

  process ( clk, rst )
    variable subsum:  std_logic_vector (7 downto 0);
  begin
    if ( rst = '1' ) then SUM <= (others => '0');
    elsif ( rising_edge(clk) ) then
      subsum := A + B;
      SUM <= subsum + C;
    end if;
  end process;

end architecture;
```

Verilog

```
module accumulator (...);

  logic [7:0] subsum;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) begin SUM <= 8'd0;
    end else begin
      subsum = A + B;
      SUM <= subsum + C;
    end
  end

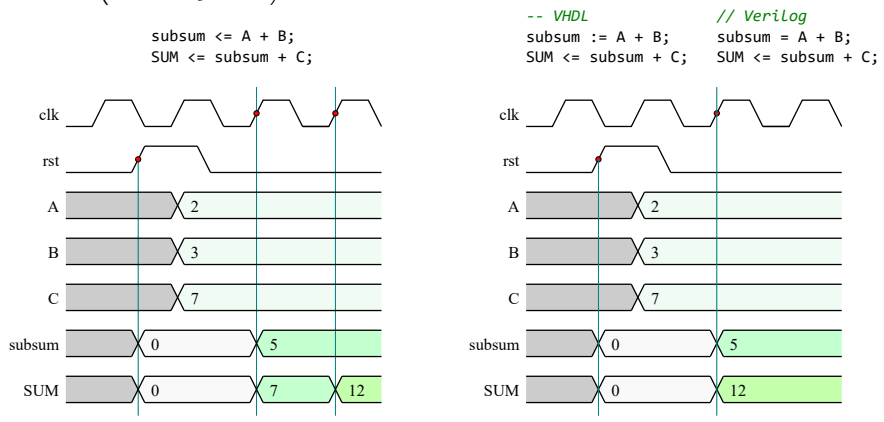
endmodule
```

4-24. ábra. *Értékadás változónak VHDL-ben és blokkoló értékadás Verilogban*

Hasonlítsuk össze ezt a két modellt részletesen:

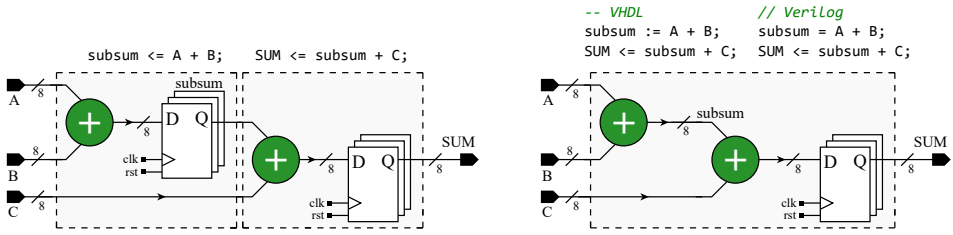
- A VHDL modell esetén most az architektúra deklarációs részében létrehozott jel (`signal`) helyett a folyamat (`process`) deklarációs részében létrehozott, ún. változó (`variable`) hordozza az A és B operandusok összegét. A változóknak a `<=` szimbólum helyett a `:=` szimbólummal adhatunk új értéket. A részösszeget ezúttal reset hatására nem inicializáljuk. A VHDL szimulációs szemantikája alapján a változó-értékadásra a jelértékadással ellentétben a delta-késleltetés nem vonatkozik. A szimuláció során tehát az órajel felfutó élénél a részösszeg azonnal megkapja az A és B operandusok összegét, így az már rendelkezésre áll a végösszeg jobbérték-kifejezésének kiértékelésekor.
- A Verilog modellben a részösszegnek ezúttal a `<=` szimbólum helyett a `=` szimbólummal adunk új értéket, amely az ún. blokkoló *(blocking)* értékadást jelöli, továbbá a részösszeget ezúttal reset hatására nem inicializáljuk. A szimuláció során a részösszegre vonatkozó értékadás blokkolja a további utasítások végrehajtását, tehát a részösszeg frissítéséig a következő utasítások várakoznak. Amikor végül sor kerül a végösszeg kiértékelésére, akkor a részösszeg már tartalmazza az A és B operandusok összegét.

A delta-késleltetés hiánya a VHDL-ben és a blokkoló értékadás a Verilogban tehát azt eredményezi, hogy az órajel felfutó élénél a végösszeg azonnal a három operandus összegét kapja értékül (lásd 4-25. ábra).



4-25. ábra. *Értékadások időzítési modelljének összehasonlítása*

Lássuk, milyen különbséget jelent mindez áramköri szempontból. Ahogy azt a 4-18. ábrán látható példáknál is említettük, egy szinkron folyamatba foglalt értékadások nemcsak a flip-flopokat, hanem az azok bemenetét meghajtó kombinációs hálózatokat is leírják a jobbérték-kifejezéseken keresztül. VHDL-ben a változó-értékadás vagy Verilogban a blokkoló értékadás használata azt eredményezi, hogy ezt a logikát most már nemcsak közvetlenül a flip-flopra vonatkozó értékadás jobbértéke, hanem azzal együtt az értékadást megelőző változó-értékadások vagy blokkoló értékadások is befolyásolják (lásd 4-26. ábra).



4-26. ábra. *A különböző típusú értékadások hatása a szintézis eredményére*

Áramköri szempontból tehát a helyzet a következőképpen is felfogható:[31]

- **VHDL-ben**: Míg a szinkron `process` által meghajtott jelekből (`signal`, `port`) a szintézis során flip-flopok születtek, addig a változókból (`variable`) egyszerű vezetékek.
- **Verilogban**: Míg az órajelvezérelt `always`-blokk által nem-blokkoló értékadással (`<=`) meghajtott változókból a szintézis során flip-flopok születtek, addig a blokkoló értékadással (`=`) meghajtottakból egyszerű vezetékek.

A jel- és változó-értékadás, valamint a blokkoló/nem blokkoló értékadás közti különbségre a ciklusokat tartalmazó szinkron folyamatok esetén különösen oda kell figyelni. A 4-27. ábra egy léptetőregiszter `for`-ciklussal megfogalmazott modelljét mutatja be.

Figyeljük meg, hogy ha ezeket a for-ciklusokat a programozási nyelvekben megszokott szemlélettel próbáljuk értelmezni, akkor azt kapjuk, hogy a ciklustörzs utasításainak lefutása után a `par_out` kimenet nulladik helyiértékű bitje a `ser_in` bemenet értékét tartalmazza, az összes többi bit viszont a `par_out` kimenet nulladik helyiértékű bitjének korábbi értékét. Egy léptetőregiszter nyilván nem így működik, márpedig a szintetizált áramkör valóban egy léptetőregiszter. A látszólagos ellentmondás feloldása abban rejlik, hogy egy folyamat törzsébe foglalt összes utasítás egyazon szimulációs ciklus alatt hajtódik végre úgy, hogy először kiértékelődik minden utasítás jobbértéke, majd ha ez a kiértékelés az összes utasításra lefutott, akkor frissülnek az utasítások bal oldalán szereplő jelek. A ciklus kiértékelése során tehát ha a `par_out` értékét olvassuk azáltal, hogy az értékadásokban a jobb oldalon szerepeltetjük, valójában mindig csak az előző szimulációs ciklusbeli, tehát ebben a példában gyakorlatilag az órajel előző felfutó élénél beállt értékét látjuk. A végeredmény szempontjából tehát az iterációs ciklusok nem egymás után, hanem látszólag egyszerre hajtódnak végre. Ez a megoldás egyenértékű azzal, mintha a `for`-ciklust 8 különálló értékadással helyettesítenénk (lásd 4-28. ábra).

A HDL modellek azt fejezik ki, hogy az órajel felfutó éle hatására a `par_out` busz minden bitjének értékül adjuk az adott bitnél eggyel alacsonyabb helyiértékű bitet, ami tehát valóban egy léptetőregiszterre jellemző viselkedés.

A 4-24. ábrán látható példában megfigyelhettük, hogy a VHDL-beli változók (`variable`) és a Verilog blokkoló értékadásai segítségével elérhető, hogy egy folyamaton belüli értékadás

[31] A szintézisre vonatkozó megállapítások nem teljesen általánosak. Bizonyos esetekben a VHDL-ben változóként deklarált vagy Verilogban blokkoló értékadással használt jelek az ebben a példában tapasztalttal ellentétben flip-flopként szintetizálódnak. A 8.1. pontban erre az esetre is láthatunk példát.

VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity shift_reg is
  port (
    clk, rst: in  std_logic;
    ser_in:   in  std_logic;
    par_out:  out std_logic_vector (7 downto 0)
  );
end entity shift_reg;

architecture rtl of shift_reg is
begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      par_out <= (others => '0');
    elsif ( rising_edge(clk) ) then
      for i in 1 to 7 loop
        par_out(i) <= par_out(i-1);
      end loop;
      par_out(0) <= ser_in;
    end if;
  end process;

end architecture rtl;
```

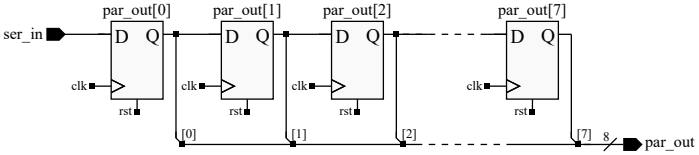
Verilog

```
module shift_reg (
  input  logic clk, rst,
  input  logic ser_in,
  output logic [7:0] par_out
);

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 )
      par_out <= 8'd0;
    else begin
      for ( int i = 1; i < 8; i = i+1 )
        par_out[i] <= par_out[i-1];
      par_out[0] <= ser_in;
    end
  end

endmodule
```

Áramköri struktúra



4-27. ábra. *Léptetőregiszter leírása `for`-ciklussal*

VHDL

```
architecture rtl of shift_reg is
begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      par_out <= (others => '0');
    elsif ( rising_edge(clk) ) then
      par_out(1) <= par_out(0);
      par_out(2) <= par_out(1);
      par_out(3) <= par_out(2);
      par_out(4) <= par_out(3);
      par_out(5) <= par_out(4);
      par_out(6) <= par_out(5);
      par_out(7) <= par_out(6);
      par_out(0) <= ser_in;
    end if;
  end process;

end architecture rtl;
```

Verilog

```
module shift_reg (...);

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) par_out <= 8'd0;
    else begin
      par_out[1] <= par_out[0];
      par_out[2] <= par_out[1];
      par_out[3] <= par_out[2];
      par_out[4] <= par_out[3];
      par_out[5] <= par_out[4];
      par_out[6] <= par_out[5];
      par_out[7] <= par_out[6];
      par_out[0] <= ser_in;
    end
  end

endmodule
```

4-28. ábra. *Ciklus helyettesítése értékadásokkal*

„azonnal lássa” az előtte lévő értékadások eredményét. Vizsgáljuk meg ilyen szempontból a 4-29. ábrán látható példát, ami egy sorozatos összeadással megvalósított szorzás műveletet ír le; az operandust tizennégyszer adjuk hozzá önmagához, így áll elő a tizenötszörös szorzat.

VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mult_add is
  port (
    clk, rst: in  std_logic;
    a:        in  unsigned (3 downto 0);
    y:        out unsigned (7 downto 0)
  );
end entity mult_add;

architecture rtl of mult_add is
begin

  process ( clk, rst )
    variable tmp: unsigned (7 downto 0);
  begin
    if ( rst = '1' ) then
      y <= (others => '0');
    elsif ( rising_edge(clk) ) then
      tmp := (others => '0');
      for i in 0 to 14 loop
        tmp := tmp + a;
      end loop;
      y <= tmp;
    end if;
  end process;

end architecture rtl;
```

Verilog

```verilog
module mult_add (
  input  logic clk, rst,
  input  logic [3:0] a,
  output logic [7:0] y
);

  logic [7:0] tmp;

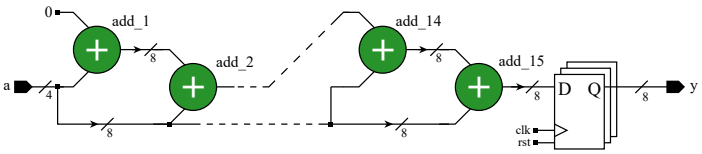
  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) y <= 8'd0;
    else begin
      tmp = 8'd0;
      for ( int i = 0; i <= 14; i = i+1 )
        tmp = tmp + a;
      y <= tmp;
    end
  end

endmodule
```

4-29. ábra. *Szorzás megvalósítása ciklikus összeadással*

A rész-szorzatokat ezúttal VHDL-ben egy változóban tároljuk, Verilogban pedig blokkoló értékadást használunk a részszorzatokra vonatkozó utasítások esetén. Korábban láttuk, hogy e két megoldás tulajdonképpen pontosan ugyanazt írja le; a szimuláció során a `tmp` változóra vonatkozó értékadások végrehajtásánál a változó azonnal megkapja új értékét, így a következő értékadások – tehát ebben az esetben a következő iterációs ciklusban lévő értékadások – kiértékelése során már ezzel számolunk. Az eredmény az, hogy a `tmp` változó valóban egy akkumulátorként működik, ahogy annak a szorzás funkció megvalósításához lennie kell. De mit jelent ez áramköri szempontból? A 4-24. ábra példájában láthattuk, hogy ha egy folyamaton belüli értékadás jobb oldalán VHDL-ben egy változóra vagy Verilogban blokkoló értékadással használt jelre hivatkozunk, akkor az előálló kombinációs hálózat miként bővül ki az ezeknek a köztes értékeknek az előállításához szükséges hardverelemekkel. A 4-29. ábrán látható példában ugyanez történik, csak ebben az esetben köztes eredménynek kell tekintenünk a ciklustörzs tizenötszöri végrehajtásának kumulált eredményét, tehát az azt megvalósító logika is tizenötször szintetizálódik. Esetünkben ez tizenöt darab, megfelelően sorba kapcsolt összeadó áramkört jelent (lásd 4-30. ábra).

Ebből a példából jól látható, hogy bár iteratív természetű nyelvi szerkezetet, ciklust használtunk a funkció kifejezésére, áramköri szempontból egyáltalán nem tekinthető a megoldás iteratívnak, hiszen az egész szorzás műveletet tizenöt darab, egyszerre működő összeadó áramkör valósítja meg. Az összeadó áramkörök sorba kapcsolásának köszönhetően



4-30. ábra. *Az iteratív nyelvi szerkezetből szintetizált, teljesen párhuzamos architektúra*

kialakuló kombinációs hálózat késleltetése meglehetősen nagy lehet, így ez a megoldás inkább csak elvi jelentőségű, a gyakorlatban a szorzás műveletet másképp kell megoldani. Figyeljük meg azt is, hogy egy ilyen modellben a hardver-erőforrások mennyisége a ciklusváltozó maximális értékétől függ, ami azt jelenti, hogy ez az érték fordítási időben meghatározható konstans kell, hogy legyen ahhoz, hogy a leírás egyáltalán szintetizálható legyen. Fontos tehát megjegyezni, hogy az iteratív, egy adott számítási műveletet több órajelciklus alatt végrehajtó, tehát az erőforrások vezérelt újrafelhasználását megvalósító architektúráknak és a HDL-ek ciklus utasításainak gyakorlatilag semmi közük nincs egymáshoz.[32]

Mindez persze nem jelenti azt, hogy a fent bemutatott, ciklusra épülő nyelvi szerkezetet ne lehetne szintézisre szánt modellekben használni. A 4-31. ábrán látható példa egy paritásgenerátor áramkört mutat be, amely esetén a HDL modell és ezáltal a szintetizált áramkör struktúrája is megegyezik az előző példa szorzó áramkörénél látottakkal.

VHDL

```
architecture rtl of parity_gen is
begin

  process ( clk, rst )
    variable tmp: std_logic;
  begin
    if ( rst = '1' ) then
      parity <= '0';
    elsif ( rising_edge(clk) ) then
      tmp := '0';
      for i in 0 to 7 loop
        tmp := tmp xor din(i);
      end loop;
      parity <= tmp;
    end if;
  end process;

end architecture rtl;
```

Verilog

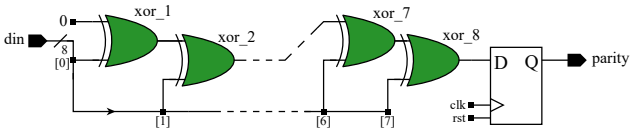
```
module parity_gen (...);

  logic tmp;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) parity <= 1'b0;
    else begin
      tmp = 1'b0;
      for ( int i = 0; i < 8; i = i+1 )
        tmp = tmp ^ din[i];
      parity <= tmp;
    end
  end

endmodule
```

Áramköri struktúra



4-31. ábra. *Paritásgenerátor-áramkör leírása `for`-ciklussal*

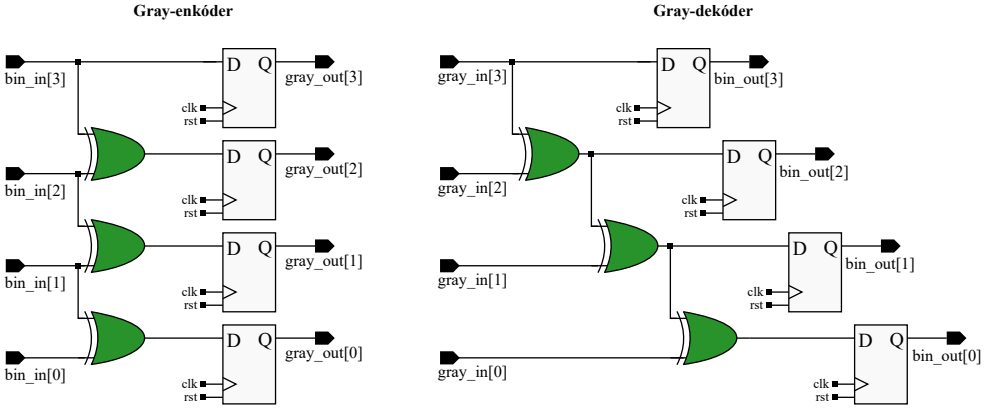
A fentiekhez hasonló, VHDL-beli változó-értékadásokat és a Verilogbeli blokkoló értékadásokat tartalmazó folyamattörzs-leírások tehát alkalmasak arra, hogy viszonylag összetett kombinációs hálózatokat írjanak le, amelyeket a folyamatnak az érzékenységi listája tesz

[32] Az erőforrások vezérelt újrafelhaszanálását másképp kell megvalósítanunk. Erre láthattunk egy egyszerű példát a 4-20. ábrán, az összetettebb, általánosabb megoldást a 4.8. pontban tárgyaljuk.

szinkronná. Az ilyen struktúrák leírása jobban átlátható, kifejezőbb lehet, sőt a kódunk újrafelhasználhatóságát is javíthatja, ha a kombinációs hálózat funkcióját egy különálló HDL tervezési egységbe szervezzük ki. Erre alkalmas nyelvi szerkezetek a függvények *(function)*. A függvények használatát egy ún. Gray-enkóder/dekóder áramkörön mutatjuk be. A Gray-enkóder egy egyszerű kombinációs hálózat, amely egy több bites bináris kombinációhoz egy ugyanolyan bitszélességű, másik kombinációt rendel azzal az egyedi jellegzetességgel, hogy ha a bemenetét binárisan ábrázolt egész számként értelmezzük, akkor a kimenetén előálló, két szomszédos számhoz rendelt kombinációk között a Hamming-távolság mindig egy lesz. Ha tehát a bemenet egy felfelé vagy lefelé számláló áramkör kimenete, akkor biztosak lehetünk benne, hogy a Gray-enkóder kimenetén előálló kombináció minden iteráció során csupán egyetlen bitben fog eltérni az előző értéktől. A 4-3. táblázat egy négybites vektor esetén szemlélteti a Gray-enkódolást, a 4-32. ábra pedig a szinkron kivitelű Gray-enkóder és a Gray-dekóder áramkörök egy lehetséges megvalósítását mutatja be.

Bináris érték	Gray-enkódolt érték
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

4-3. táblázat. *Gray-enkódolás négybites esetben*



4-32. ábra. *A szinkron Gray-enkódolást és dekódolást megvalósító áramkörök szerkezete*

VHDL

```vhdl
entity gray_encoder is
  port (
    clk, rst:   in  std_logic;
    bin_in:     in  std_logic_vector (3 downto 0);
    gray_out:   out std_logic_vector (3 downto 0)
  );
end entity gray_encoder;

architecture rtl of gray_encoder is

  function gray_enc (bin_in:  std_logic_vector) return
    std_logic_vector is
    variable tmp: std_logic_vector (3 downto 0);
  begin
    tmp := X"0";
    for i in 0 to 2 loop
      tmp(i) := bin_in(i) xor bin_in(i+1);
    end loop;
    tmp(3) := bin_in(3);
    return tmp;
  end gray_enc;

begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      gray_out <= X"0";
    elsif ( rising_edge(clk) ) then
      gray_out <= gray_enc(bin_in);
    end if;
  end process;

end architecture rtl;
```

Verilog

```verilog
module gray_encoder (
  input   logic clk, rst,
  input   logic [3:0] bin_in,
  output  logic [3:0] gray_out
);

  function [3:0] gray_enc;

    input [3:0] bin_in;
    logic [3:0] tmp;

    begin
      tmp = 4'b0;
      for ( int i = 0; i <= 2; i = i+1 )
        tmp[i] = bin_in[i] ^ bin_in[i+1];
      tmp[3] = bin_in[3];
      gray_enc = tmp;
    end

  endfunction

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) gray_out <= 4'd0;
    else gray_out <= gray_enc(bin_in);
  end

endmodule
```

4-33. ábra. *Szinkron kimenetű Gray-enkóder megvalósítása függvénnyel*

VHDL

```vhdl
entity gray_decoder is
  port (
    clk, rst: in  std_logic;
    gray_in:  in  std_logic_vector (3 downto 0);
    bin_out:  out std_logic_vector (3 downto 0)
  );
end entity gray_decoder;

architecture rtl of gray_decoder is

  function gray_dec (gray_in:  std_logic_vector) return
    std_logic_vector is
    variable tmp: std_logic_vector (3 downto 0);
  begin
    tmp := X"0";
    tmp(3) := gray_in(3);
    for i in 2 downto 0 loop
      tmp(i) := tmp(i+1) xor gray_in(i);
    end loop;
    return tmp;
  end gray_dec;

begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      bin_out <= X"0";
    elsif ( rising_edge(clk) ) then
      bin_out <= gray_dec(gray_in);
    end if;
  end process;

end architecture rtl;
```

Verilog

```verilog
module gray_decoder (
  input   logic clk, rst,
  input   logic [3:0] gray_in,
  output  logic [3:0] bin_out
);

  function [3:0] gray_dec;

    input [3:0] gray_in;
    logic [3:0] tmp;

    begin
      tmp = 4'b0;
      tmp[3] = gray_in[3];
      for ( int i = 2; i >= 0; i = i-1 )
        tmp[i] = tmp[i+1] ^ gray_in[i];
      gray_dec = tmp;
    end

  endfunction

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) bin_out <= 4'd0;
    else bin_out <= gray_dec(gray_in);
  end

endmodule
```

4-34. ábra. *Szinkron kimenetű Gray-dekóder megvalósítása függvénnyel*

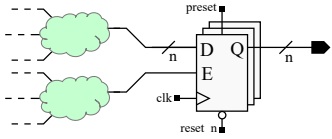
A Gray-kódolás gyakorlati jelentőségére a 6.3.1. pontban visszatérünk. A 4-33. és 4-34. ábra azt mutatja be, hogy a HDL-ekben hogyan lehet függvények segítségével a 4-32. ábrán látható szinkron kimenetű Gray-enkódert és dekódert készíteni.

4.5 RTL építőelemek

Az alábbi fejezetben az RTL tervezés során gyakran használt nyelvi szerkezetekre és az azokból szintézis útján előálló áramköri részletekre mutatunk példákat.

4.5.1 Regiszterek

A *„register-transfer”* kifejezésből is sejthető, hogy az RTL modellekben kritikus jelentősége van annak, hogy miként modellezzük magukat a regisztereket, flip-flopokat.[33] Az előző alfejezetben bemutatott példák jól mutatják, hogy a HDL modellekben többnyire nem önmagukban írjuk le a regisztereket, hanem szinkron folyamatokban adunk értéket jeleknek, amelynek eredményeként a szintézis során előáll egy, a 4-35. ábrán láthatóhoz hasonló szerkezetű áramköri részlet, amely a regisztert és az annak bemeneteit (az adatbemenetet és – ha szükséges – az órajel-engedélyező bemenetet) meghajtó kombinációs hálózatot egyaránt tartalmazza.



4-35. ábra. *Szinkron folyamatbeli jelértékadás eredménye*

Természetesen ha a folyamatbeli értékadás jobbérték-kifejezéseiben nem fogalmazunk meg semmilyen aritmetikai és/vagy logikai kapcsolatot, akkor egyszerűen csak regisztereket kapunk eredményül, logika nélkül. A jobb áttekinthetőség kedvéért az alábbi példákban is így teszünk, annak ellenére, hogy valós körülmények között az ilyen „lecsupaszított” leírások nem túl gyakoriak.

A regiszterek sokfélék lehetnek, használhatunk aszinkron vagy szinkron resetet vagy presetet attól függően, hogy szeretnénk-e valamilyen globális jel hatására alapállapotba állítani a tárolót, dönthetünk e jelek polaritásáról, használhatunk órajel-engedélyező bemenetet, ami lehetővé teszi, hogy az adott regiszter csak bizonyos feltételek fennállása esetén frissítse a tartalmát, sőt, még azt is meghatározhatjuk, hogy a regiszter az órajel felfutó vagy lefutó élére legyen érzékeny. A regisztereknek ezek a tulajdonságai a HDL modellben a szinkron folyamatok megvalósításának pontos részleteiben jelennek meg. Az eddig

[33] A „flip-flop” és „regiszter” kifejezések jelentése a különböző forrásokban eltérhet. Egyesek regiszter alatt flip-flopok vektorát értik (pl. egy mikroprocesszor utasításmutató regisztere), mások pedig az órajel-engedélyező bemenettel ellátott flip-flopokat hívják regiszternek. E könyvben a két kifejezést szinonímaként használjuk.

bemutatott példák esetén az órajel felfutó élére érzékeny, magas-aktív, aszinkron reset bemenettel ellátott regisztereket használtunk, ami az egyik legszélesebb körben alkalmazott megoldás (4-36. ábra).

VHDL

```
architecture rtl of d_flip_flop is
begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then q <= '0';
    elsif ( rising_edge(clk) ) then
       q <= d;
    end if;
  end process;

end architecture rtl;
```

Verilog

```
module d_flip_flop (...);

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) q <= 1'b0;
    else q <= d;
  end

endmodule
```

4-36. ábra. *Felfutó élre érzékeny regiszter aszinkron, magas-aktív resettel*

A regiszterek fent említett tulajdonságai a legelemibb szinten befolyásolják leendő áramkörünk erőforrásigényét és időzítési tulajdonságait is, és bár az RTL modelleket igyekszünk teljesen technológiafüggetlenül megfogalmazni, ez a cél valójában már itt sem teljesülhet maradéktalanul minden esetben. Ennek az az oka, hogy a céltechnológia által rendelkezésünkre bocsátott regiszterek nem feltétlenül képesek közvetlenül, önmagukban implementálni az általunk megfogalmazott viselkedést. Különösen igaz ez a programozható logikai eszközök esetén, ahol ezek a regiszterek már ott vannak az eszközben és képességeik bizonyos mértékben konfigurálhatók ugyan, de messze nem nyújtanak olyan rugalmasságot, mint egy standard cellakönyvtár, ahol sokkal többféle regiszter-típusból válogathatunk. A 4-37. ábra azt mutatja be, hogy az Intel (Altera) MAX10 FPGA-család esetén hogyan változik meg a szintézis eredménye, ha aszinkron resetről szinkronra térünk át.

VHDL

```
architecture async_reset of register is
begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then q <= '0';
    elsif ( rising_edge(clk) ) then
      q <= d;
    end if;
  end process;

end architecture async_reset;
```

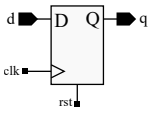
Verilog

```
module register (...);

  always @ ( posedge clk,
             posedge rst )
  begin
    if ( rst == 1'b1 ) q <= 1'b0;
    else q <= d;
  end

endmodule
```

Áramköri struktúra




```
architecture sync_reset of register is
begin

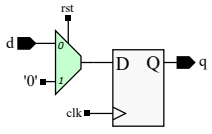
  process ( clk )
  begin
    if ( rising_edge(clk) ) then
      if ( rst = '1' ) then q <= '0';
      else q <= d;
      end if;
    end if;
  end process;

end architecture async_reset;
```

```
module register (...);

  always @ ( posedge clk )
  begin
    if ( rst == 1'b1 ) q <= 1'b0;
    else q <= d;
  end

endmodule
```



4-37. ábra. *Aszinkron és szinkron resettel ellátott regiszter szintézise Intel MAX10 FPGA esetén*

Mivel a MAX10 FPGA-ban fizikailag jelen lévő regiszterek aszinkron reset bemenettel vannak ellátva, ezért a szintézis eszköz kénytelen volt a szinkron resetet egy, a jelútba beiktatott multiplexerrel megvalósítani. Ez azt jelenti, hogy ha egy egész tervben – ami regiszterek százezreit jelentheti – következetesen mindenütt szinkron resetet használunk, akkor a szintetizált áramkör minden egyes regiszterének bemenetére egy multiplexer kerül, ami jelentős erőforrás-többlet az aszinkron resetet alkalmazó megoldáshoz képest. Ez a példa is jól mutatja, hogy a HDL modellek elkészítése során lehet ugyan technológiafüggetlenségre törekedni, de a megvalósítás alapjául szolgáló eszköz, cellakönyvtár, sőt a logikai és fizikai szintézis szoftverek mély ismerete is elengedhetetlen ahhoz, hogy igazán jó minőségű RTL modelleket tudjunk készíteni. Éppen ezért technológiafüggetlenség alatt tipikusan azt értjük, hogy modellünk bármilyen technológia esetén szintetizálható, nem pedig azt, hogy az adott technológián optimális. Az adott alkalmazástól függ az, hogy a kód hordozhatósága vagy a technológia optimális kihasználása a fontosabb peremfeltétel.

A regisztereket bemutató ábrákon – ahogy az összes többi ábra esetén is – az órajel- és reset-bemeneteket szándékosan másképp ábrázoltuk, mint az adat- (D) és engedélyező (E) bemeneteket. Az órajel és az aszinkron vezérlő bemenetek ún. globális jelek, amelyeket az esetek túlnyomó többségében – különösen programozható logikai eszközök esetén – nem kombinációs hálózatok hajtanak meg, hanem közvetlenül chip-szintű, kis késleltetésű és elcsúszású *(low-skew)*, ún. órajel- és reset-terjesztő vezetékhálózatokhoz (órajelfa, *clock tree*) kapcsoljuk őket. Ahogy azt a 4.4.4. pontban részleteztük, az RTL tervezés során azt feltétez lezzük, hogy az órajel-hozzávezetés késleltetése bármely flip-flop esetén azonos, tehát az ún. órajelelcsúszás *(clock skew)* mértéke nulla. A fizikailag megvalósított áramkörünk is annál jobb[34] lesz, minél inkább sikerül megközelíteni ezt a helyzetet. Ugyanakkor éppen az órajel és a reset azok a jelek, amelyek minden flip-flophoz el kell, hogy jussanak, tehát az ezeket a jeleket meghajtó áramkörök terhelése a legnagyobb. RTL tervezőként gondoskodnunk kell arról, hogy a szintézis algoritmus feltétlenül be tudja azonosítani ezeket a globális jeleket a leírásunkban, és semmi olyasmit ne írjunk bele a modellbe, ami megakadályozza a szintézis szoftvert abban, hogy a megfelelő áramköri elemeket, globális buffereket beépítse a netlistába a kedvező időzítési tulajdonságok elérése érdekében. Általános ökölszabályként megfogalmazható, hogy az órajelet és a reset jelet a flip-flopok megfelelő meghajtásán kívül semmire ne használjuk, és ezeket a jeleket semmilyen más jellel ne kombináljuk össze. Ha követjük ezeket az egyszerű szabályokat, akkor azt fogjuk észrevenni, hogy tulajdonképpen minden szinkron folyamatunk egyfajta mintázatot követ; mindegyik úgy néz ki, mint amit a 4-37. ábrán is láthatunk:

- Az érzékenyégi listán kizárólag az órajel és – aszinkron reset esetén – a reset jel szerepel.
- A folyamat egy `if-else-end` szerkezetet tartalmaz, amelynek `if` ága az adott folyamatban meghajtott flip-flopok kezdőértékeit beállító értékadásokat, `else` (VHDL-ben `elsif`) ága pedig az órajel felfutó élénél végrehajtandó feladatot leíró

[34] „Jó" alatt itt az időzítési tulajdonságokat értjük.

utasításokat tartalmazza.[35] Ez az utasításblokk bármilyen bonyolult lehet a leírandó funkciótól függően.

Az egyszerű regisztereket gyakran használjuk ún. éldetektorként. Az éldetektorok egy jelen bekövetkező felfutó és/vagy lefutó élekből egy órajelperiódus ideig fennálló impulzusokat állítanak elő.[36] A 4-38. ábra egy felfutóél-detektort mutat be.

VHDL

```
architecture rtl of edge_detector is

  signal t_d: std_logic;

begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      t_d <= '0';
    elsif ( rising_edge(clk) ) then
      t_d <= t;
    end if;
  end process;

  t_rising <= t and not t_d;

end architecture rtl;
```

Verilog

```
module edge_detector (...);

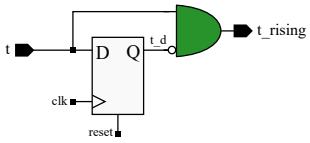
  logic t_d;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) t_d <= 1'b0;
    else t_d <= t;
  end

  assign t_rising = t & ~t_d;

endmodule
```

Áramköri struktúra



4-38. ábra. *Felfutóél-detektor*

Az éldetektor kimeneti logikai kapuját megváltoztatva befolyásolható, hogy az éldetektor a felfutó, lefutó vagy mindkét típusú élváltásra érzékeny legyen. A kimenet negálásával az előállított impulzus polaritását is meghatározhatjuk. A 4-39. ábra egy futásidőben konfigurálható, általános célú éldetektort mutat be.

Az éldetektorok jelentősége abban áll, hogy egy jelen végbemenő élváltást szinkron működésű áramköreink számára egyszeri eseménnyé „konvertál", úgy, hogy az élváltás kiváltotta tranziens után az éldetektor maga ugyanabba az állapotba kerül vissza, amiben volt. Így kimenetét felhasználhatjuk arra, hogy további tranziens-jellegű eseményeket váltson ki az áramkörünkön belül. A 4-40. ábrán bemutatott példában egy 8-bites bemenet értékét csak akkor mintavételezzük, ha a megfigyelt triggerjelen lefutó élt detektáltunk. A megoldáshoz felhasználjuk az általános célú éldetektort.

Vezérlési szempontból nagy rugalmasságot jelent, ha kézben tudjuk tartani, hogy egy adott regiszter az órajelnek melyik felfutó élére reagáljon és melyikre ne. Ezt az ún. engedélyezhető regiszterek teszik lehetővé. A 4-41. ábra egy, az órajel lefutó élére érzékeny,

[35] Az RTL modellekre azt is szokás mondani, hogy ezek azok a modellek, amelyek meghatározzák, hogy mi történik az áramkörben az órajel két felfutó éle között.

[36] Egy teljes órajelperiódus ideig fennálló kimeneti impulzus csak akkor keletkezik, ha a bemenetet meghajtó forrás ugyanarról az órajelről jár, amiről az éldetektor.

VHDL

```vhdl
architecture rtl of edge_detector is

  signal t_d, t_e: std_logic;

begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then t_d <= '0';
    elsif ( rising_edge(clk) ) then
      t_d <= t;
    end if;
  end process;

  t_e <= (t and not t_d) when e = "00" else
         (t_d and not t) when e = "01" else
         (t xor t_d);

  t_event <= t_e when neg = '0' else not t_e;

end architecture rtl;
```

Verilog

```verilog
module edge_detector (...);

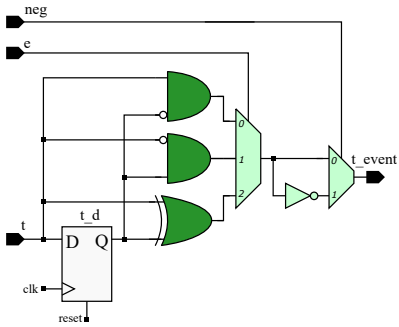
  logic t_d, t_e;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) t_d <= 1'b0;
    else t_d <= t;
  end

  always @ ( t, t_d, e )
  begin
    case ( e )
      2'b00:    t_e <= t & ~t_d;
      2'b01:    t_e <= t_d & ~t;
      default:  t_e <= t ^ t_d;
    endcase;
  end
  assign t_event = (neg == 1'b0) ? t_e : ~t_e;

endmodule
```

Áramköri struktúra



4-39. ábra. *Futásidőben konfigurálható éldetektor*

alacsony-aktív, aszinkron resettel és magas-aktív órajel-engedélyező bemenettel ellátott regisztert mutat be.

Az engedélyezhető regiszter megvalósítása viszonylag egyszerű akkor is, ha a rendelkezésre álló flip-flopok nem rendelkeznek órajel-engedélyező bemenettel, de a mai modern programozható logikai eszközök és standard cellakönyvtárak flip-flopjai ezt a funkciót cella-szinten megvalósítják.

4.5.2 Multiplexerek

A korábbi példákban már láthattuk, hogy az `if-else` és a `case` utasítások használhatók multiplexerek leírására. A 4-42. ábra egy négybemenetű, kombinációs hálózatként, önmagában megvalósított multiplexert mutat be.

Ha kombinációs hálózatként valósítunk meg multiplexert, akkor figyelnünk kell, hogy a kiválasztó bemenet összes lehetséges értékénél határozzuk meg a kimenet értékét. Ha nem így teszünk, akkor egy szintvezérelt tárolót, *latch*-et kapunk (lásd 4-43. ábra), hiszen

VHDL

```
architecture rtl of sample is

 signal t_falling: std_logic;

begin

  FALLDET: entity work.edge_detector(rtl)
             port map (
               clk     => clk,
               rst     => rst,
               e       => "0i1",
               neg     => '0',
               t       => t,
               t_event => t_falling
             );

  process ( clk, rst )
  begin
    if ( rst = '1' ) then d_out <= X"00";
    elsif ( rising_edge(clk) ) then
      if ( t_falling = '1' ) then d_out <= d_in;
      end if;
    end if;
  end process;

end architecture rtl;
```

Verilog

```
module sample (...);

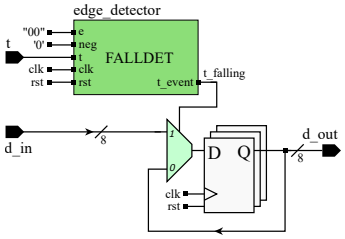
  logic t_falling;

  edge_detector FALLDET(
    .clk(clk),
    .rst(rst),
    .e(2'b01),
    .neg(1'b0),
    .t(t),
    .t_event(t_falling)
  );

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) d_out <= 8'd0;
    else
      if ( t_falling == 1'b1 ) d_out <= d_in;
  end

endmodule
```

Áramköri struktúra



4-40. ábra. *Mintavételezés egy triggerjelen bekövetkező élváltás hatására*

VHDL

```
architecture rtl of d_flip_flop is
begin

  process ( clk, rst )
  begin
    if ( rst = '0' ) then q <= '0';
    elsif ( falling_edge(clk) ) then
      if ( ena = '1' ) then q <= d;
      end if;
    end if;
  end process;

end architecture rtl;
```

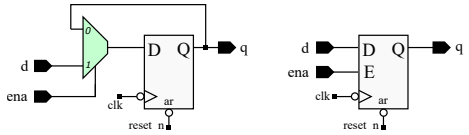
Verilog

```
module d_flip_flop (...);

  always @ ( negedge clk, negedge rst )
  begin
    if ( rst == 1'b0 ) q <= 1'b0;
    else if ( ena == 1'b1 ) q <= d;
  end

endmodule
```

Áramköri struktúra



4-41. ábra. *Engedélyezhető regiszter*

VHDL

```
architecture rtl of mux4 is
begin

  process ( a, b, c, d, sel )
  begin
    case sel is
      when "00"   => y <= a;
      when "01"   => y <= b;
      when "10"   => y <= c;
      when others => y <= d;
    end case;
  end process;

end architecture rtl;
```

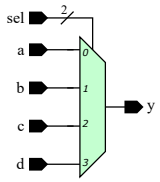
Verilog

```
module mux4 (...);

  always @ ( a, b, c, d, sel )
  begin
    case ( sel )
      2'b00:   y <= a;
      2'b01:   y <= b;
      2'b10:   y <= c;
      default: y <= d;
    endcase
  end

endmodule
```

Áramköri struktúra



4-42. ábra. *Négybemenetű, aszinkron multiplexer*

tulajdonképpen ekkor a HDL modell azt fejezi ki, hogy ha a kiválasztó bemenet olyan értéket vesz fel, amire nem határoztuk meg a kimenet értékét, akkor a kimenet tartsa meg az előző értéket. Ez tehát egy olyan tároló funkció, ami a kiválasztó bemenet értékétől, bitjeinek logikai szintjétől függően tárol vagy nem, ami valóban egy latch-et jelent.

VHDL

```
architecture rtl of mux_latch is
begin

  process ( a, b, c, sel )
  begin
    case sel is
      when "00"  => y <= a;
      when "01"  => y <= b;
      when "10"  => y <= c;
    end case;
  end process;

end architecture rtl;
```

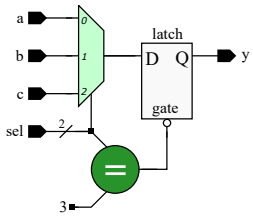
Verilog

```
module mux_latch (...);

  always @ ( a, b, c, sel )
  begin
    case ( sel )
      2'b00:  y <= a;
      2'b01:  y <= b;
      2'b10:  y <= c;
    endcase
  end

endmodule
```

Áramköri struktúra



4-43. ábra. *Latch szintézise ki nem fejtett case szerkezetek miatt*

A latch ugyan megvalósítható áramköri elem, de a leendő áramkörünk időzítési viszonyainak ellenőrzését végző algoritmus (statikus időzítésvizsgálat, *Static Timing Analysis*, STA) számára nehezen, vagy egyáltalán nem kezelhető. Általános tervezési elv, hogy csak élérzékeny tárolókat, vagyis flip-flopokat, azon belül is szinte kizárólag D flip-flopokat használunk. Ha a hibásan megfogalmazott HDL modellünk alapján a szintézis szoftvernek latch-et kellene beiktatnia, akkor rendszerint figyelmeztető üzenetet kapunk, sőt, egyes szintézis szoftverek esetén ez fordítási idejű hibát jelent. A `case` szerkezetek esetén VHDL-ben a `when others`, Verilogban pedig a `default` utasítással gondoskodhatunk róla, hogy a kiválasztó jel összes lehetséges értéke megjelenjen a kiértékelésben. Ezeknek az utasításoknak a használata gyakorlatilag elkerülhetetlen, hiszen a VHDL `std_logic` típusa kilenc, a Verilog `logic` típusa pedig 4 különböző értéket vehet fel, ami azt jelenti, hogy ha a fenti példában a `when others` és a `default` utasítások nélkül szeretnénk felsorolni a kiválasztó jel összes lehetséges értékét, akkor a VHDL modellben elvileg 512, a Verilog modellben 16 sorunk lenne.

Multiplexer funkciót az `if-else` utasításokkal is megvalósíthatunk, de ennek jelentése szintézis szempontjából nem teljesen ugyanaz, mint a `case` szerkezeteké (lásd 4-44. ábra).

VHDL

```
architecture rtl of mux4_prior is
begin

  process ( a, b, c, d, sel )
  begin
    if ( sel = "00" ) then y <= a;
    elsif ( sel = "01" ) then y <= b;
    elsif ( sel = "10" ) then y <= c;
    else y <= d;
    end if;
  end process;

end architecture rtl;
```

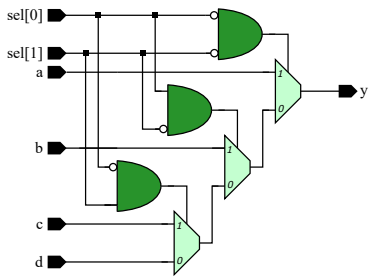
Verilog

```
module mux4_prior (...);

  always @ ( a, b, c, d, sel )
  begin
    if ( sel == 2'b00 ) y <= a;
    else
      if ( sel == 2'b01 ) y <= b;
      else
        if ( sel == 2'b10 ) y <= c;
        else y <= d;
  end

endmodule
```

Áramköri struktúra



4-44. ábra. *Prioritásos multiplexálás if-else szerkezetekkel*

E példa azt szemlélteti, hogy az `if-else` utasításból a szintézis során létrehozott áramkör – a nyelvek szimulációs szemantikájával összhangban – prioritást rendel a kiválasztó bemenet különböző értékeihez. Az `if-else` szerkezetnek ezt a tulajdonságát érdemes észben tartani, amikor bonyolult logikai kapcsolatokat fogalmazunk meg segítségével; az eredményül adódó kombinációs hálózat logikai szintjeinek száma gyorsan nő, ahogy az utasítás bonyolódik, így a leendő áramkör lokális adatútjainak késleltetése nagyon nagyra adódhat. Ha az `if-else` szerkezetbe foglalt feltételek a 4-44. ábrán láthatóhoz hasonló módon köl4 csönösen kizárják egymást, akkor inkább a párhuzamos `case` szerkezetet érdemes használni.

Ha az `if-else` szerkezet feltételei között szándékosan prioritást szeretnénk kifejezni, akkor a folyamatok belsejében lévő utasítások közötti sorrendiséget is kihasználhatjuk.

A 4-45. ábra azt szemlélteti, hogy a folyamatokon belül a később leírt utasítások a szimulációs szemantika alapján később értékelődnek ki, tehát a korábbi kiértékelés eredményét felülírják. Ez áramköri szempontból ugyanúgy prioritást jelent, mint az `if-else` szerkezetekbe expliciten belekódolt prioritás. A latch szintetizálásáról a kifejtetlen `case` szerkezeteknél elmondottak az `if-else` szerkezetek esetén is fennállnak. Ha kombinációs hálózatot modellezünk `if-else` szerkezetekkel, akkor a latch szintetizálásának elkerülése érdekében gondoskodnunk kell arról, hogy a megfogalmazott feltételrendszer teljes legyen. A 4-44. ábrán szereplő példában a feltétel nélküli `else` ág, a 4-45. ábra példájában pedig a folyamat legelső, ugyancsak feltétel nélküli értékadása tölti be az alapértelmezett kimenetet meghatározó kifejezés szerepét.

VHDL

```vhdl
architecture rtl of prior_mux is
begin

  process ( a, b, c, sel )
  begin
    y <= d;
    if ( sel = "10" ) then y <= c; end if;
    if ( sel = "01" ) then y <= b; end if;
    if ( sel = "00" ) then y <= a; end if;
  end process;

end architecture rtl;
```

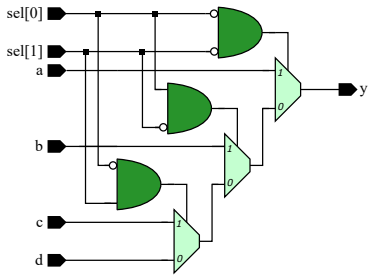
Verilog

```verilog
module prior_mux (...);

  always @ ( a, b, c, sel )
  begin
    y <= d;
    if ( sel == 2'b10 ) y <= c;
    if ( sel == 2'b01 ) y <= b;
    if ( sel == 2'b00 ) y <= a;
  end

endmodule
```

Áramköri struktúra



4-45. ábra. *A folyamatok sorrendi jellegének kihasználása prioritás megfogalmazásához*

4.5.3 Aritmetika

A digitális áramköröket igen gyakran használjuk bonyolult számítási műveletek elvégzésére, nem meglepő tehát, hogy mind VHDL-ben, mind Verilogban megvannak a megfelelő eszközök az aritmetikai műveletek leírására. A 4-46. ábra egy szinkron összeadó/kivonó áramkört mutat be.

Az előző példákhoz képest itt már több különbséget is felfedezhetünk a VHDL és a Verilog modell között. Ezek a különbségek a VHDL típusosságára vezethetők vissza:

- A VHDL `std_logic` és `std_logic_vector` típusai ún. felsorolt típusok, amelyek csupán egy-egy bitet vagy egy bitsorozatot jelentenek. Az `std_logic_1164` csomagban, amely e típusok deklarációját tartalmazza, nem találunk olyan operátorokat, függvényeket, amelyek az ilyen típusú operandusokon végzett aritmetikai műveleteket írnák le. Ahhoz, hogy VHDL-ben aritmetikai műveleteket végezzünk bitvektorokon, a `numeric_std` csomagra is szükségünk van[37]. Ez a csomag deklarálja az `unsigned` és `signed` típusokat, mint az `std_logic_vector` típus altípusait. Ezekre a típusokra már definiálva vannak az aritmetikai műveleteket leíró operátorok, valamint a szükséges konverziós és típusváltó függvények is. A Verilogban a `logic` típusú jelekre maga a nyelv definiálja az aritmetikai operátorokat, tehát megtehetjük, hogy az ilyen típusú jeleket egyszerűen összeadjuk, kivonjuk vagy összeszorozzuk.

[37] A `numeric_std` csomagon kívül léteznek más, aritmetikai operátorokat definiáló csomagok, de a `numeric_std` az, amelyik a szabvány része lett, a legtöbb szimulációs és szintézis szoftver ennek a használatát javasolja.

VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity alu is
  port (
    clk, rst: in  std_logic;
    a, b:     in  unsigned (7 downto 0);
    func:     in  std_logic;
    y:        out unsigned (8 downto 0)
  );
end entity alu;

architecture rtl of alu is
begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      y <= (others => '0');
    elsif ( rising_edge(clk) ) then
      case func is
        when '0' => y <= ('0' & a) + b;
        when '1' => y <= ('0' & a) - b;
        when others => null;
      end case;
    end if;
  end process;

end architecture rtl;
```

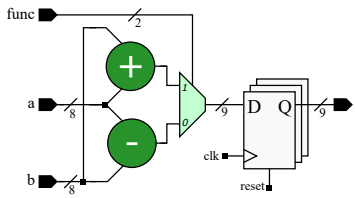
Verilog

```
module alu (
  input   logic       clk, rst,
  input   logic [7:0] a, b,
  input   logic       func,
  output  logic [8:0] y
);

  always @ ( posedge clk,
             posedge rst )
  begin
    if ( rst == 1'b1 ) begin
      y <= 9'd0;
    end else begin
      case ( func )
        1'b0:     y <= a + b;
        1'b1:     y <= a - b;
        default:  ;
      endcase
    end
  end

endmodule
```

Áramköri struktúra



4-46. ábra. *Szinkron aritmetikai egység*[38]

- Egy másik érdekes különbség, hogy VHDL-ben a `numeric_std` csomagban definiált összeadás és kivonás operátorok által visszaadott eredmény bitszélessége mindig a nagyobb bitszélességű operanduséval egyezik meg. A fenti példában tehát ahhoz, hogy 9-bites eredményt kapjunk, legalább az egyik operandust ki kell egészítenünk egy nullával. Verilogban az összeadás operátor által visszaadott eredmény mindig eggyel nagyobb bitszélességű, mint a nagyobb bitszélességű operandus.

A 4-47. ábrán látható példa olyan kombinációs hálózatot ír le, amely előjeles, kettes komplemens kódolású egész számként értelmezve a bemeneteit megállapítja, hogy melyik a nagyobb, és azt juttatja ki a kimenetre. Ha a négybites bemenet bizonyul nagyobbnak, akkor azt a nyolcbites kimenetre előjel-kiterjesztéssel kapcsolja rá.

Összeadás és kivonás esetén az operandusok előjele áramköri szempontból nem fontos, de ha két szám összehasonlításáról van szó, akkor már feltétlenül egyeztetnünk kell az operandusok előjelét a helyes eredményhez. Ha előjeles, kettes komplemens kódolású egész számokként szeretnénk értelmezni a bitvektorokat, akkor VHDL-ben a `signed` típust kell használnunk, ami az `unsigned`-hoz hasonlóan az `std_logic_vector` típusnak egy altípusa,

[38] A VHDL-ben a „&” szimbólum – az Ada nyelvből származó örökségként – bit-összefűzést jelent.

VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity greater is
  port (
    a:  in  signed (7 downto 0);
    b:  in  signed (3 downto 0);
    y:  out signed (7 downto 0)
  );
end entity greater;

architecture logic of greater is
begin

  process ( a, b )
  begin
    if ( a > b ) then y <= a;
    else
      y(7 downto 4) <= (others => b(3));
      y(3 downto 0) <= b;
    end if;
  end process;

end architecture logic;
```

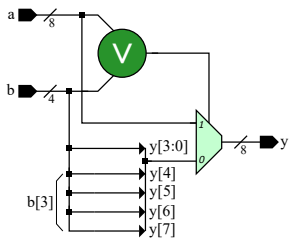
Verilog

```verilog
module greater (
  input   logic signed [7:0] a,
  input   logic signed [3:0] b,
  output  logic signed [7:0] y
);

  always @ ( a, b )
  begin
    if ( a > b ) y <= a;
    else y <= b;
  end

endmodule
```

Áramköri struktúra



4-47. ábra. *Előjeles, kettes komplemens kódolású egész számok összehasonlítása*

amire tehát a `numeric_std` csomagban definiálva vannak az összehasonlító operátorok. A Verilog által alkalmazott egyik implicit, a HDL modellből nem egyértelműen megállapítható szabály, hogy a `logic` típusú vektorokat az aritmetikai operátorok előjel nélküli egészként kezelik. Ha előjeles, kettes komplemens kódolású egész számként szeretnénk értelmezni őket, akkor a `signed` típusmódosítót kell használnunk a jel vagy port deklarálásánál. Ugyancsak fontos különbség a két nyelv között, hogy míg VHDL-ben bármilyen értékadás csak akkor helyes, ha a balérték bitszélessége megegyezik a jobbérték-kifejezés bitszélességével, addig Verilogban az ilyen szituációk feloldására szintén be vannak építve bizonyos implicit szabályok:

- Ha a balérték kisebb bitszélességű, mint a jobbérték-kifejezés, akkor ez utóbbinak a felső helyiértékeken szereplő bitjeit figyelmen kívül hagyjuk.
- Ha a balérték nagyobb bitszélességű, mint a jobbérték-kifejezés, akkor:
  - Ha a jobbérték-kifejezés előjel nélküli, akkor ezt az értéket a balérték alsó bitjeibe töltjük be, a felső biteket pedig nullázzuk.
  - Ha a jobbérték-kifejezés előjeles, akkor az előjel-kiterjesztés automatikusan megtörténik.

VHDL-ben ilyen implicit szabályok egyáltalán nincsenek, a tervező kénytelen szándékát egzaktul megfogalmazni a HDL modellben, ahogy az a 4-47. ábrán is látható; az `y` kimenet [7:4] mezőjének minden bitje a `b` bemenet harmadik helyiértékű bitjének értékét kapja, ezt fejezi ki az „`others`” kulcsszó.

A digitális jelfeldolgozással kapcsolatos alkalmazásokban a szorzás műveletnek kiemelt jelentősége van. A HDL-ekben a szorzást az összeadáshoz és a kivonáshoz hasonlóan a megfelelő operátor segítségével írhatjuk le, figyelembe véve természetesen, hogy a szorzat bitszélessége az operandusok bitszélességeinek összege lesz. A 4-48. ábra a digitális szűrők megvalósításának alapelemét, egy MAC *(Multiply & Accumulate)* egységet mutat be.

VHDL

```
architecture rtl of mac is

  signal x: signed (23 downto 0);

begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      x <= (others => '0');
      y <= (others => '0');
    elsif ( rising_edge(clk) ) then
      if ( clr = '1' ) then
        x <= (others => '0');
        y <= (others => '0');
      else
        if ( ena = '1' ) then
          x <= a * b;
          y <= y + x;
        end if;
      end if;
    end if;
  end process;

end architecture rtl;
```

Verilog

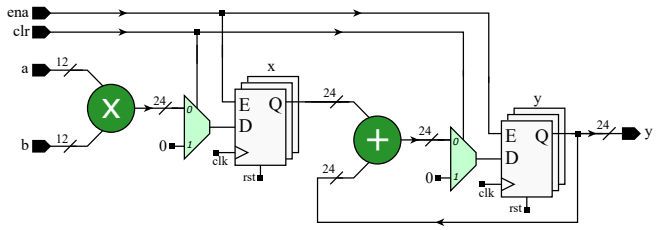
```
module mac (...);

  logic signed [23:0] x;

  always @ ( posedge clk,
             posedge rst )
  begin
    if ( rst == 1'b1 ) begin
      x <= 24'd0; y <= 24'd0;
    end else begin
      if ( clr == 1'b1 ) begin
        x <= 24'd0;
        y <= 24'd0;
      end else begin
        if ( ena == 1'b1 ) begin
          x <= a * b;
          y <= y + x;
        end
      end
    end
  end

endmodule
```

Áramköri struktúra



4-48. ábra. *MAC egység*

Az aritmetikai műveletek szintézisével kapcsolatban ezen a ponton említést kell tennünk az osztásról is. Az osztás művelet számítástechnikailag sokkal összetettebb dolog, mint a három másik alapművelet. Kombinációs hálózatként való megvalósítása a fenti példákban bemutatotthoz hasonló módon nem hatékony, rengeteg erőforrást igényel.[39] Ez természetesen nem jelenti azt, hogy ne tudnánk osztás műveletet végezni digitális áramkörökkel, de ezek rendszerint iteratív algoritmusokat – pl. *shift&subtract*, *Goldschmidt* stb. – megvalósító, vezérlő és műveletvégző elemek megfelelően összekapcsolt hálózatát igénylő megoldások (lásd 4.8. pont).

[39] A szintézis szoftverek általában fel vannak készítve arra a speciális esetre, ha 2 valamely hatványával kell osztani. Ekkor ti. valójában léptetés *(shift)* műveletről van szó, amit a szintézis szoftverek természetesen könnyen meg tudnak valósítani.

4.5.4 Regisztertömbök, memóriák

A digitális áramkörökben az információ ideiglenes tárolására nemcsak különálló regisztereket használhatunk. A biteket helyiértékkel ellátva vektorokat hozhatunk létre, e vektorokhoz indexet rendelve pedig tömböket képezhetünk, így tehát lényegében kétdimenziós rendbe szervezett biteket kapunk. RTL tervezőként azonban mindig fejben kell tartanunk, hogy ha regisztertömböt hozunk létre, akkor minden időpillanatban csak a tárolt információ kis részéhez, egyszerre csak legfeljebb a regisztertömb néhány rekeszéhez, bejegyzéséhez férhetünk hozzá. A 4-49. ábrán látható példa egy ilyen regisztertömböt mutat be.

VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity regfile is
  port (
    clk, rst:   in  std_logic;
    we:         in  std_logic;
    address:    in  unsigned (2 downto 0);
    data_in:    in  std_logic_vector (3 downto  0);
    data_out:   out std_logic_vector (3 downto 0)
  );
end entity regfile;

architecture rtl of regfile is

  type content_t is array (0 to 7) of
    std_logic_vector (3 downto 0);
  signal content: content_t;

begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      content <= ( others => (others => '0') );
    elsif ( rising_edge(clk) ) then
      if ( we = '1' ) then
        content(to_integer(address)) <= data_in;
      end if;
    end if;
  end process;

  data_out <= content(to_integer(address));

end architecture rtl;
```

Verilog

```
module regfile (
  input      logic       clk, rst,
  input      logic       we,
  input      logic [2:0] address,
  input      logic [3:0] data_in,
  output     logic [3:0] data_out
);

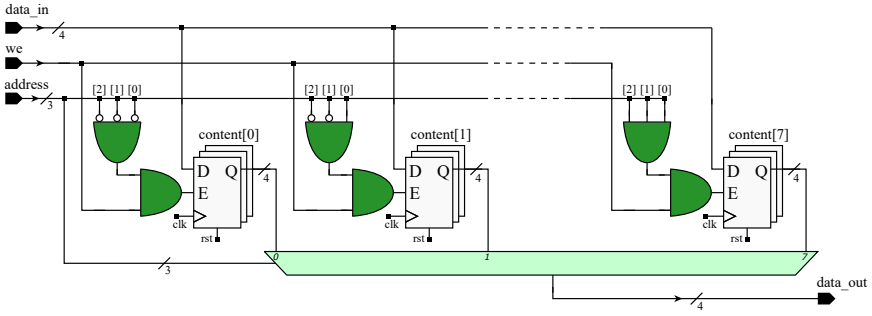
  logic [3:0] content [0:7];

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 )
      for ( int i = 0; i < 8; i = i + 1 )
        content[i] <= 4'd0;
    else if ( we == 1'b1 )
      content[address] <= data_in;
  end

  assign data_out = content[address];

endmodule
```

Áramköri struktúra



4-49. ábra. *8 × 4 méretű regisztertömb*

A fenti példa funkcionálisan ugyan memóriaszerű, de ha a tárolt bejegyzések száma, vagy a címbusz mérete nagy, akkor ez a megoldás megengedhetetlenül sok regisztert és kombinációs logikát használ fel. Nagy mennyiségű adat tárolására a gyakorlatban nem regisztertömböket használunk, mert az ennek alapjául szolgáló regiszter (D flip-flop) viszonylag nagy méretű és fogyasztású eszköz, legalábbis ahhoz képest, hogy mindössze egyetlen bit információt tárol. Nagy mennyiségű eltárolandó adat esetén inkább memóriákat használunk, amelyeket cellaszinten nem D flip-flopok halmazaként, hanem kifejezetten nagy mennyiségű információ tárolására tervezett, technológiájukat tekintve rendszerint SRAM-alapú makrocellákként valósítunk meg. Programozható logikai eszközök – tulajdonképpen itt beszélhetünk kizárólag az FPGA-król – esetén ezek az ún. blokk-RAM modulok, amelyek a flip-flopokhoz hasonlóan rendelkezésre állnak az eszközben. A blokk-RAM modulokat kétféle módon használhatjuk RTL modelljeinkben; példányosíthatjuk közvetlenül az adott programozható eszköz cellakönyvtárának a megfelelő elemét a gyártó által adott HDL sablon (gyakorlatilag egy példányosítást leíró HDL kódrészlet) segítségével, vagy rábízhatjuk a szintézis szoftverre, hogy a memóriaszerű viselkedést leíró HDL leírásunk alapján „rájöjjön" *(inference)*, hogy mire gondoltunk, és megfelelően felparaméterezve maga helyezze el a kapuszintű netlistában a cellakönyvtár memóriamodulját. Ez persze csak úgy lehetséges, ha a leírásunkban megfogalmazott viselkedés megvalósítható a rendelkezésre álló memóriamodul segítségével, itt tehát nagyon körültekintően, a korábban már említett, gyártófüggő kódolási szabályokat figyelembe véve kell a HDL modellt elkészíteni. Standard cellás technológia esetén ezek a modulok ugyancsak a cellakönyvtár, tehát az alapelemkészlet részének tekinthetők, bár szintézisük kicsit körülményesebb, mint más alapcelláké. A cellakönyvtár tervezője – aki egyúttal rendszerint a gyártó is – egy speciális szoftvert, ún. makrocella-generátort bocsát rendelkezésünkre, amivel beállíthatjuk a kívánt memóriamodul paramétereit, előállíthatjuk a példányosításhoz szükséges HDL kódrészletet és egyéb reprezentációkat, például szimulációs modelleket, vagy a layout-rajzolatot.

A 4-49. ábrán látható példa kis módosításával elérhető, hogy azt a szintézis szoftverek memóriaként kezeljék (lásd 4-50. ábra).

VHDL

```
architecture rtl of single_port_ram is

  type content_t is array (0 to 7) of
    std_logic_vector (3 downto 0);
  signal content: content_t;

begin

  process ( clk )
  begin
    if ( rising_edge(clk) ) then
      if ( we = '1' ) then
        content(to_integer(address)) <= data_in;
      end if;
      data_out <= content(to_integer(address));
    end if;
  end process;

end architecture rtl;
```

Verilog

```
module single_port_ram (...);

  logic [3:0] content [0:7];

  always @ (posedge clk)
  begin
    if ( we == 1'b1 ) content[address] <= data_in;
    data_out <= content[address];
  end

endmodule
```

4-50. ábra. *8 × 4 méretű memória*

A regisztertömb modelljét csak annyiban módosítottuk, hogy szinkronná tettük a kimenetet, valamint megszüntettük a reset jelet, mivel az SRAM memóriáknak az inicializálása az egyszerű flip-flopoknál megszokott módon nem lehetséges.

Az alkalmazások egy jelentős részében hasznosnak bizonyul, ha a memóriát egyidőben tudjuk írni és olvasni. Az ilyen memóriákat *dual-port* memóriának nevezzük (lásd 4-51. ábra).

VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dual_port_ram is
  port (
    clk:        in  std_logic;
    we:         in  std_logic;
    waddress:   in  unsigned (2 downto 0);
    raddress:   in  unsigned (2 downto 0);
    data_in:    in  std_logic_vector (3 downto  0);
    data_out:   out std_logic_vector (3 downto 0)
  );
end entity dual_port_ram;

architecture rtl of dual_port_ram is

  type content_t is array (0 to 7) of
    std_logic_vector (3 downto 0);
  signal content: content_t;

begin

  process ( clk )
  begin
    if ( rising_edge(clk) ) then
      if ( we = '1' ) then
        content(to_integer(waddress)) <= data_in;
      end if;
      data_out <= content(to_integer(raddress));
    end if;
  end process;

end architecture rtl;
```

Verilog

```verilog
module dual_port_ram (
  input      logic       clk, we,
  input      logic [2:0] waddress, raddress,
  input      logic [3:0] data_in,
  output     logic [3:0] data_out
);

  logic [3:0] content [0:7];

  always @ ( posedge clk )
  begin
    if ( we == 1'b1 ) content[waddress] <= data_in;
    data_out = content[raddress];
  end

endmodule
```

4-51. ábra. *8 × 4 méretű, dual-port memória*

Az egyszerű dual-port memóriamodul felhasználásával létrehozhatunk ún. *true-dual-port* memóriát, ami azt jelenti, hogy a memóriatartalmat egy porton írhatjuk és két külön porton olvashatjuk, így egyszerre két rekesz tartalmához férhetünk hozzá (vö. 3.1. pont: belső regisztertömb olvasásának ütemezése a viselkedési modellben). A 4-52. ábra egy ilyen mes móriamodult mutat be.

A true-dual-port viselkedést megvalósító modell az egyszerű dual-port memória HDL modelljéből példányosítás nélkül is egyszerűen származtatható (lásd 4-53. ábra). Szintézis szempontjából e két megoldás azonosnak tekinthető.

A fenti módon a kiolvasó interfészek számát tulajdonképpen tetszőlegesen növelhetjük, azonban számolnunk kell azzal, hogy ekkor valójában többszörözzük a felhasznált memóriacellák számát. A probléma tulajdonképpen ugyanaz, mint amit már láthattunk a bevezetőben is az akkumulátorban felhasznált aritmetikai egységek számával kapcsolatban; választhatunk a párhuzamosított, tehát gyors, de nagy erőforrásigényű és a szekvenciális, kis erőforrásigényű, de lassú megoldás között. Az akkumulátoros példában az összeadó

VHDL

```
architecture rtl of true_dual_port_ram is
begin

  RAM_A:  entity work.dual_port_ram(rtl)
            port map (
              clk       => clk,
              we        => we,
              waddress  => waddress,
              raddress  => raddress_a,
              data_in   => data_in,
              data_out  => data_out_a
            );

  RAM_B:  entity work.dual_port_ram(rtl)
            port map (
              clk       => clk,
              we        => we,
              waddress  => waddress,
              raddress  => raddress_b,
              data_in   => data_in,
              data_out  => data_out_b
            );

end architecture rtl;
```

Verilog

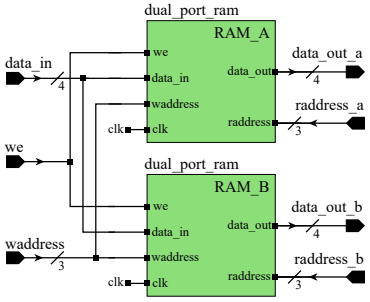
```
module true_dual_port_ram (...);

  dual_port_ram RAM_A(
    .clk(clk),
    .we(we),
    .waddress(waddress),
    .raddress(raddress_a),
    .data_in(data_in),
    .data_out(data_out_a)
  );

  dual_port_ram RAM_B(
    .clk(clk),
    .we(we),
    .waddress(waddress),
    .raddress(raddress_b),
    .data_in(data_in),
    .data_out(data_out_b)
  );

endmodule
```

Áramköri struktúra



4-52. ábra. *True-dual-port memória létrehozása az egyszerű dual-port memória példányosításával*

VHDL

```
architecture rtl of dual_port_ram is

  type content_t is array (0 to 7) of
    std_logic_vector (3 downto 0);
  signal content: content_t;

begin

  process ( clk )
  begin
    if ( rising_edge(clk) ) then
      if ( we = '1' ) then
        content(to_integer(waddress)) <= data_in;
      end if;
      data_out_a <= content(to_integer(raddress_a));
      data_out_b <= content(to_integer(raddress_b));
    end if;
  end process;

end architecture rtl;
```

Verilog

```
module dual_port_ram (...);

  logic [3:0] content [0:7];

  always @ (posedge clk)
  begin
    if ( we ) content[waddress] <= data_in;
    data_out_a <= content[raddress_a];
    data_out_b <= content[raddress_b];
  end

endmodule
```

4-53. ábra. *True-dual-port memória leírása az egyszerű dual-port memória példányosítása nélkül*

VHDL

```vhdl
architecture rtl of fifo is

  type content_t is array (0 to 255) of
    std_logic_vector (7 downto 0);
  signal content: content_t;

begin

  process ( clk, rst )
    variable read_pointer:  unsigned (7 downto 0);
    variable write_pointer: unsigned (7 downto 0);
  begin

    if ( rst = '1' ) then
      read_pointer := X"00";
      write_pointer := X"00";
      empty <= '1';
      full <= '0';
      dout <= X"00";
    elsif ( rising_edge(clk) ) then

      if ( push = '1' and full /= '1' ) then
        content(to_integer(write_pointer)) <= din;
        write_pointer := write_pointer + 1;
        if ( write_pointer = read_pointer ) then
          full <= '1';
        else full <= '0';
        end if;
        empty <= '0';
      end if;

      if ( pop = '1' and empty /= '1' ) then
        dout <= content(to_integer(read_pointer));
        read_pointer := read_pointer + 1;
        if ( write_pointer = read_pointer ) then
          empty <= '1';
        else empty <= '0';
        end if;
        full <= '0';
      end if;

    end if;

  end process;

end architecture rtl;
```

Verilog

```verilog
module fifo (...);

  logic [7:0] content [0:255];
  logic [7:0] read_pointer;
  logic [7:0] write_pointer;

  always @ (posedge clk, posedge rst)
  begin

    if ( rst ) begin
      read_pointer <= 8'd0;
      write_pointer <= 8'd0;
      full <= 1'b0;
      empty <= 1'b1;
      dout <= 8'd0;
    end else begin

      if ( push && !full ) begin
        content[write_pointer] <= din;
        write_pointer = write_pointer + 8'd1;
        if ( write_pointer == read_pointer )
          full = 1'b1;
        else full = 1'b0;
        empty = 1'b0;
      end

      if ( pop && !empty ) begin
        dout <= content[read_pointer];
        read_pointer = read_pointer + 8'd1;
        if ( write_pointer == read_pointer )
          empty = 1'b1;
        else empty = 1'b0;
        full = 1'b0;
      end

    end

  end

endmodule
```

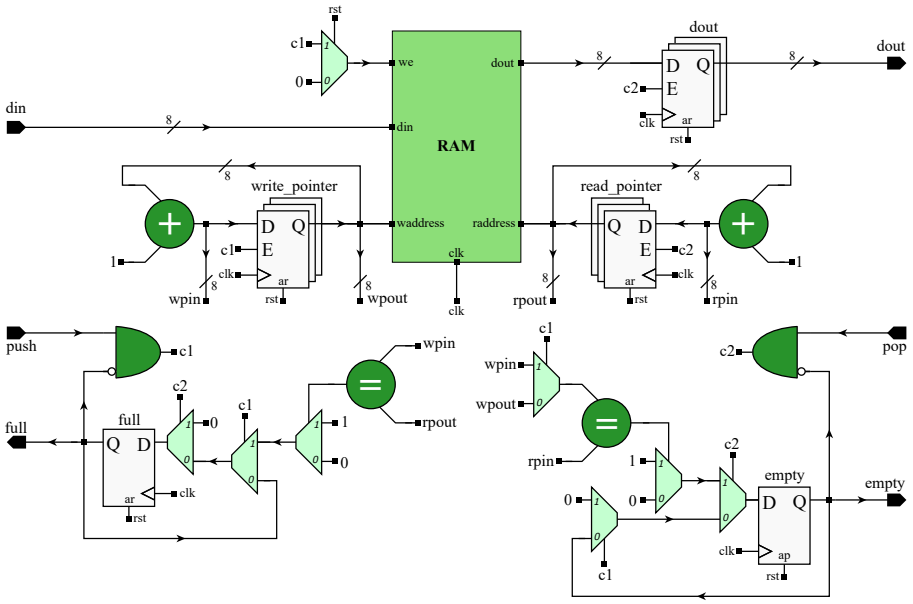
4-54. ábra. *FIFO tároló HDL modelljei*

áramkör, a memória esetén pedig maga a kiolvasó interfész az az erőforrás, amelynek vezérelt újrafelhasználása vagy duplikálása áll szemben egymással. Az RTL tervező feladata eldönteni, hogy az adott alkalmazásban az erőforrásigény vagy a műveletvégzési sebesség a fontosabb szempont.

A memóriákat gyakran használjuk FIFO tárolóként. A FIFO-kat az RTL modellekben cirkuláris bufferként valósítjuk meg, ami azt jelenti, hogy egy memóriamodul címbemeneteire ún. beíró és kiolvasó mutatókat (*write* és *read pointer*) állítunk. Beírás *(push)* művelet esetén a beíró, kiolvasás *(pop)* művelet esetén a kiolvasó mutató értékét növeljük. A FIFO állapotát leíró tele *(full)* és üres *(empty)* állapotjelzőket e két mutató egymáshoz viszonyított helyzete alapján állapítjuk meg (lásd 4-54. ábra).

Ez a leírás figyelembe veszi a memóriamodulok modellezésére vonatkozó legalapvetőbb, a 4-50. ábra példáján bemutatott szabályokat (a memóriatartalmat nem inicializáljuk reset jellel és szinkron működésű a kimenet), így a szintézis szoftverek képesek felismerni, hogy itt valójában egy „trükkösen címzett” memóriáról van szó, amit tehát egy megfelelő, beíró

és kiolvasó mutatókat tartalmazó címző logikával ellátott blokk-RAM-ként valósítanak meg (4-55. ábra).

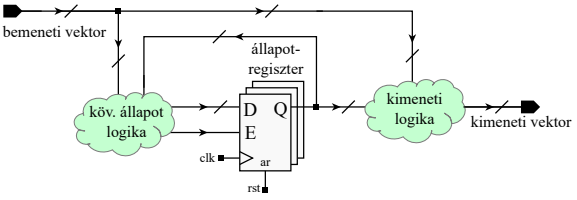


4-55. ábra. *FIFO tároló RTL kapcsolási sémája*

Ez a struktúra a bemutatott HDL modellből automatizált úton áll elő a szintézis során. Ezt a példát szintézis szempontból részletesen is megvizsgáljuk a 8.1. pontban.

4.5.5 Állapotgépek

A digitális áramkörökben az egyik legösszetettebb és egyben legsokoldalúbb vezérlési megoldás az állapotgép. Állapotgép alatt itt olyan áramkört értünk, amely az órajel minden felfutó élénél a bemeneteinek aktuális állapota, valamint saját belső állapota alapján meghatározza a kimeneteinek, valamint a saját belső állapotának következő értékét. Az állapotgépek rendszerint más áramkörök számára állítanak elő megfelelő ütemezésű vezérlőjeleket, például kiválasztójeleket multiplexerek vagy órajel-engedélyező jeleket regiszterek számára. A 4-56. ábra az állapotgépek általános szerkezetét mutatja be.



4-56. ábra. *Állapotgépek általános szerkezete*

Az állapotgépeket HDL-ek segítségével többféleképpen is leírhatjuk. Vegyünk példaként egy olyan aritmetikai áramkört, amelynek két operandusa (`a` és `b`) és egy engedélyező bemenete (`enable`) van, és amely az engedélyező bemeneten megjelenő, egyetlen órajel-periódusig aktív impulzus hatására először az `a` operandust, majd egy órajelciklussal később a két operandus összegét, egy újabb órajelciklus után pedig a két operandus különbségét teszi a kimenetére. A két operandus különbsége szintén csak egy órajel-periódus ideig van

VHDL

```
architecture rtl of fsm is

  type state_t is (s0, s1, s2);
  signal pres_state, next_state: state_t;

begin

  L_STATE: process ( clk, rst )
  begin
    if ( rst = '1' ) then
      pres_state <= s0;
    elsif ( rising_edge(clk) ) then
      pres_state <= next_state;
    end if;
  end process;

  L_NSL: process ( pres_state, enable )
  begin
    case pres_state is
      when s0     =>  if ( enable = '1' ) then
                        next_state <= s1;
                      else
                        next_state <= s0;
                      end if;
      when s1     =>  next_state <= s2;
      when s2     =>  next_state <= s0;
      when others =>  null;
    end case;
  end process;

  L_OL: process ( pres_state, a, b )
  begin
    case pres_state is
      when s0     =>  y <= a;
      when s1     =>  y <= a + b;
      when s2     =>  y <= a - b;
      when others =>  null;
    end case;
  end process;

end architecture rtl;
```

Verilog

```
module fsm (...);

  typedef enum logic [1:0] { s0, s1, s2 } state_t;
  state_t pres_state, next_state;

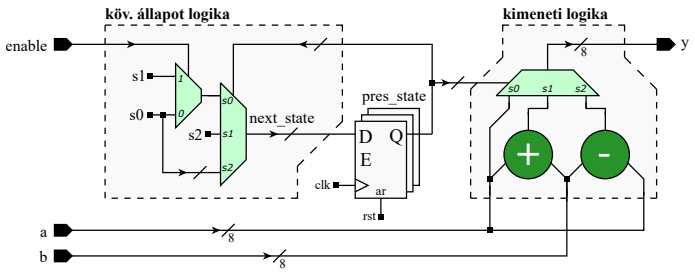
  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) pres_state <= s0;
    else pres_state <= next_state;
  end

  always @ ( pres_state, enable )
  begin
    case ( pres_state )
      s0:       if ( enable == 1'b1 )
                  next_state <= s1;
                else next_state <= s0;
      s1:       next_state <= s2;
      s2:       next_state <= s0;
      default:  ;
    endcase
  end

  always @ ( pres_state, a, b )
  begin
    case ( pres_state )
      s0:       y <= a;
      s1:       y <= a + b;
      s2:       y <= a - b;
      default:  ;
    endcase
  end

endmodule
```

Áramköri struktúra



4-57. ábra. *Állapotgép leírása három különálló utasítással*

VHDL

```
architecture rtl of fsm is
  type state_t is (s0, s1, s2);
  signal pres_state, next_state: state_t;
begin

  L_STATE: process ( clk, rst )
  begin
    if ( rst = '1' ) then
      pres_state <= s0;
    elsif ( rising_edge(clk) ) then
      pres_state <= next_state;
    end if;
  end process;

  L_LOGIC: process ( pres_state, enable, a, b )
  begin
    case pres_state is
      when s0     =>  y <= a;
                      if ( enable = '1' ) then
                        next_state <= s1;
                      else
                        next_state <= s0;
                      end if;
      when s1     =>  y <= a + b;
                      next_state <= s2;
      when s2     =>  y <= a - b;
                      next_state <= s0;
      when others =>  null;
    end case;
  end process;

end architecture rtl;
```

Verilog

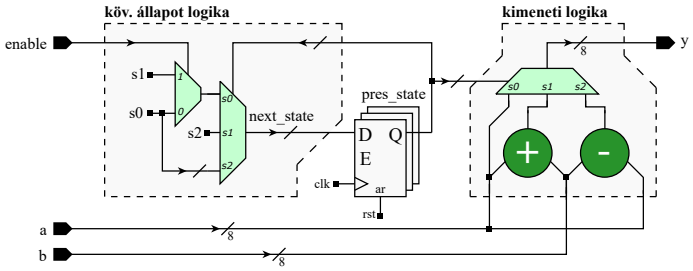
```
module fsm (...);
  typedef enum logic [1:0] { s0, s1, s2 } state_t;
  state_t pres_state, next_state;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) pres_state <= s0;
    else pres_state <= next_state;
  end

  always @ ( pres_state, enable, a, b )
  begin
    case ( pres_state )
      s0: begin
            y <= a;
            if ( enable == 1'b1 )
              next_state <= s1;
            else next_state <= s0;
          end
      s1: begin
            y <= a + b;
            next_state <= s2;
          end
      s2: begin
            y <= a - b;
            next_state <= s0;
          end
      default:  ;
    endcase
  end

endmodule
```

Áramköri struktúra



4-58. ábra. *Állapotgép leírása két különálló utasítással*

a kimeneten, ezután az áramkör alapállapotba tér vissza, ahol megint az a operandus kerül a kimenetre. A 4-56. ábrára tekintve kézenfekvőnek tűnik, hogy az állapotgépek három összetevőjét egy-egy különálló HDL utasítással írjuk le. A 4-57. ábra ezt a megoldást mutatja be.

Ez a megoldás tehát az állapotgépek három strukturális elemét, az állapotregisztert, a következő állapotot meghatározó kombinációs hálózatot *(next-state logic)* és a kimenetek értékét meghatározó kombinációs hálózatot *(output logic)* három különálló folyamat segítségével írja le. Az is megfigyelhető, hogy az állapotregiszter különböző értékeit az RTL modellben felsorolt típusú jelekkel modellezzük. Az állapotregiszter különböző értékeihez tehát az RTL modellben – általános esetben – még nem rendelünk bináris reprezentációt.[40]

[40] Nagymegbízhatóságú alkalmazásokban egyes technológiákon (pl. Microsemi RTAX család) jellemző, hogy az állapotkódolást nem bízzuk a szintézis algoritmusra, hanem „belekódoljuk” a HDL modellbe. Ilyenkor az állapotváltozó egy egyszerű logikai vektor, amelynek különböző értékeit konstansok segítségével elnevezzük a kód jobb olvashatósága érdekében.

VHDL

```
architecture rtl of fsm is

  type state_t is (s0, s1, s2);
  signal state: state_t;

begin

  L_STATE: process ( clk, rst )
  begin
    if ( rst = '1' ) then
      state <= s0;
      y <= X"00";
    elsif ( rising_edge(clk) ) then
      case state is
        when s0     =>  y <= a;
                        if ( enable = '1' ) then
                          state <= s1;
                        end if;
        when s1     =>  y <= a + b;
                        state <= s2;
        when s2     =>  y <= a - b;
                        state <= s0;
        when others =>  null;
      end case;
    end if;
  end process;

end architecture rtl;
```

Verilog

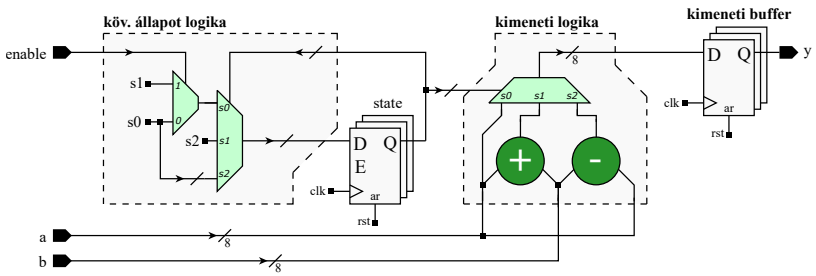
```
module fsm (...);

  typedef enum logic [1:0] { s0, s1, s2 } state_t;
  state_t state;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) begin
      state <= s0;
      y <= 8'd0;
    end else begin
      case ( state )
        s0: begin
              y <= a;
              state <= s1;
            end
        s1: begin
              y <= a + b;
              state <= s2;
            end
        s2: begin
              y <= a - b;
              state <= s0;
            end
        default:  ;
      endcase
    end
  end

endmodule
```

Áramköri struktúra



4-59. ábra. *Állapotgép leírása egyetlen utasítással*

Ezt a hozzárendelést, az ún. állapotkódolást az esetek többségében az RTL szintézis algoritmus végzi el a szintézis során (lásd 8.2.1. pont).

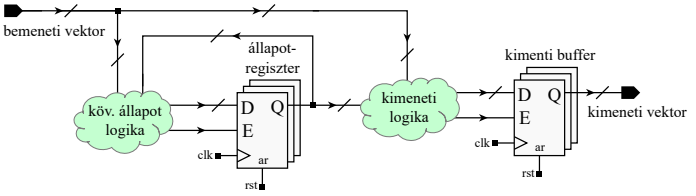
A három folyamatot alkalmazó leírásnál némileg tömörebb modellhez jutunk, ha az állapotgép kombinációs részét, vagyis a következő állapotot meghatározó és a kimenetek értékét meghatározó logikát egy közös folyamatban írjuk le. Erre a megoldásra mutat példát a 4-58. ábra.

A három és a kettő folyamatot használó leírások között sem szimulációs, sem szintézis szempontból nincs semmi különbség, a két leírás teljesen azonosnak tekinthető.

A harmadik állapotgép-leírási mód még tovább egyszerűsíti a HDL modellt azáltal, hogy a három szerkezeti elemet egyetlen folyamatba tömörítve írja le (4-59. ábra). Ezt a megoldást *one-shot* állapotgép-leírásnak is nevezzük.

Ez a megoldás a három közül a lehető legtömörebb, de mielőtt kineveznénk ezt a módszert az állapotgépek legjobb leírási módjának, vizsgáljuk meg a HDL kód által kifejezett áramköri struktúrát egy kicsit közelebbről. Tekintve, hogy az állapotregiszter

tulajdonképpen egy sor flip-flop – függetlenül attól, hogy egyelőre a pontos bináris kódolás még nem meghatározott –, az egyetlen folyamat, amit a teljes állapotgép leírására használunk, szinkron folyamat kell hogy legyen. Ugyanakkor ebben a folyamatban az állapotgép kimeneteinek is értéket kell adnunk. Ez azt jelenti, hogy ebben modellben – a három, illetve két folyamatot használó megoldásokkal ellentétben – a kimenetek olyan jelek lesznek, amelyeket szinkron folyamat hajt meg, tehát valójában minden kimenet egy-egy flip-flopként fog megvalósulni a szintézis során. Ezt úgy is kifejezhetjük, hogy a one-shot típusú állapotgép leírás az állapotgép három alapvető szerkezeti elemén, az állapotregiszteren, a következő állapotot meghatározó logikán és a kimenetek értékét meghatározó logikán kívül implicit módon tartalmaz egy-egy kimeneti buffert is (lásd 4-60. ábra).



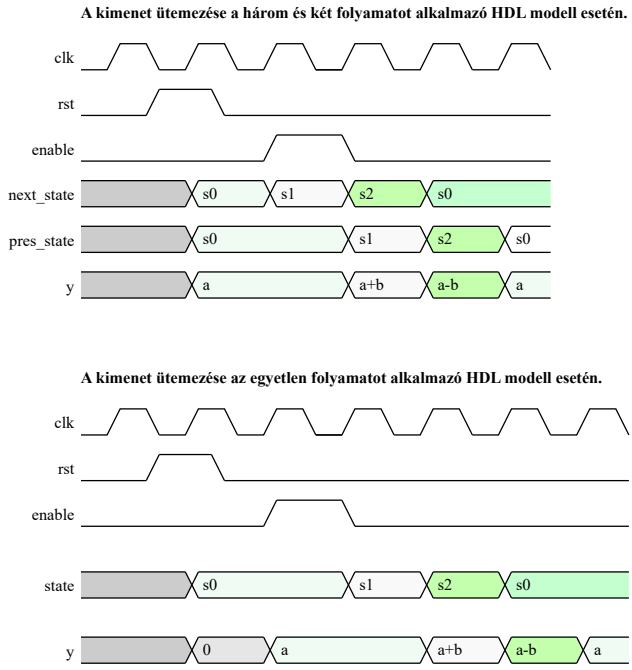
4-60. ábra. *One-shot technikával leírt állapotgépek általános szerkezete*

Ez a sajátosság két fontos következményt von maga után:

- A one-shot típusú leírásból szintetizált áramkörök erőforrásigénye nagyobb lesz, hiszen minden kimenethez egy-egy többlet flip-flop fog tartozni.
- A külső vezérlőjelek – a példa esetén az `enable` jel – szempontjából a kimenetek értékének változásai egy órajelciklussal később jelennek meg, mint a másik két megoldás esetén (lásd 4-61. ábra).

Vegyük észre, hogy bár a bemeneti vezérlő jelhez viszonyítva a kimenet megváltozása valóban el van csúszva egy órajelciklussal, de a kimenetek egymáshoz viszonyított változási üteme pontosan megegyezik a három és két folyamatot használó megoldásoknál tapasztalttal. Ez annyit jelent, hogy önmagában az, hogy egyetlen folyamatot használunk az áramkör működésének leírására, nem jelent semmiféle korlátot a funkció megvalósíthatósága szempontjából. Ezt az ütemezési sajátosságot egész egyszerűen figyelembe lehet venni a tervezés során; ha feltétlenül szükségünk van arra, hogy az áramkör kimenete úgy változzon, ahogy a másik két leírás esetén, akkor a kimeneti értékadásokat át kell rendeznünk. Ekkor természetesen a kimeneti logika nem fog teljesen megegyezni a három és két folyamatot alkalmazó megoldásokból szintetizált kimeneti logikával, de ennek különösebb jelentősége nincs.

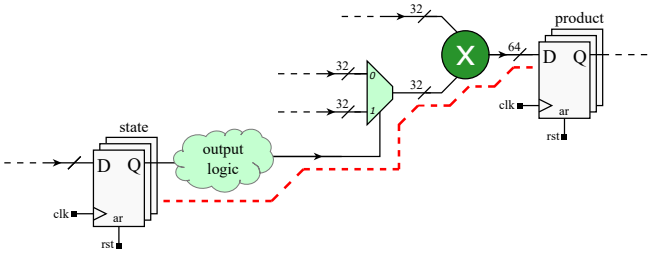
Megállapítható tehát, hogy a felhasznált erőforrások szempontjából a három és két folyamatot alkalmazó megoldásokhoz képest a one-shot leírásból szintetizált áramkör lényegében csak a kimeneti bufferek jelenlétében különbözik. A kimeneti bufferek árnyoldala magától értetődő; az áramkör több flip-flopot igényel, nagyobb területet foglal, nagyobb lesz a fogyasztása. Vizsgáljuk meg a kimeneti bufferek használatának előnyeit is!



4-61. ábra. *Ütemezési különbség a három/kettő és az egy folyamatot alkalmazó állapotgép-leírások között*

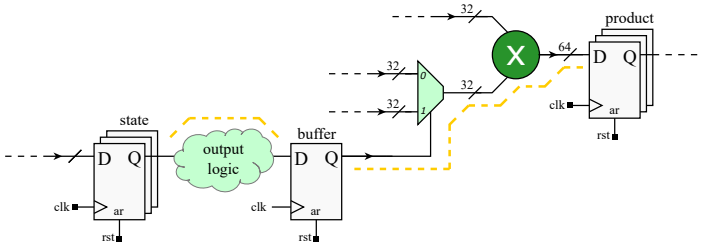
A 4-56. ábrán azt látjuk, hogy általános esetben az állapotgépek kimeneti logikáját egyes bemenetek – az ún. Mealy-bemenetek – közvetlenül meghajtják. Ez azt jelenti, hogy ha a bemenetek változásai nincsenek szinkronizálva az állapotgép órajeléhez, akkor az állapotgép által előállított kimeneteken ezek a Mealy-bemenetek glitch-eket okozhatnak, ami a vezérelt rendszerben funkcionális hibákat eredményezhet. Ha kimeneti buffereket alkalmazunk, akkor ezek a glitch-ek nem juthatnak az állapotgép kimenetére.

A kimeneti bufferek az áramkör időzítési tulajdonságait is javíthatják. Tegyük fel, hogy az állapotgépünk egy összetett aritmetikai műveleteket végző másik áramkört vezérel a 4-62. ábrán látható séma szerint.



4-62. ábra. *Nagy késleltetésű lokális adatút az állapotgép és a vezérelt áramkör között*

Kimeneti bufferek hiányában kialakul egy ún. lokális adatút (közvetlen összeköttetés két flip-flop között, részletesen lásd 6. fejezet) az állapotgép és az általa vezérelt, nagy



4-63. ábra. *Kimeneti bufferek, mint pipeline regiszterek az állapotgép és a vezérelt áramkör között*

erőforrásigényű műveletvégző elemek között. Ennek a lokális adatútnak a késleltetése meglehetősen nagy lehet, hiszen nemcsak az állapotgép kimeneti logikáját, hanem egy multiplexert és egy szorzó áramkört is magában foglal.

Kimeneti bufferek alkalmazásával – ahogy az a 4-63. ábrán is megfigyelhető – tulajdonképpen egy pipeline regiszterréteget iktatunk be ebbe a lokális adatútba, így kisebb valószínűséggel alakul ki itt a kritikus út, vagyis a legnagyobb késleltetésű, az áramkörünk maximális órajel-frekvenciáját meghatározó adatút.

Az egy folyamatot alkalmazó leírás egy további előnyt rejt a másik két megoldással szemben, nevezetesen hogy lehetővé teszi az adatfeldolgozó rendszereknek egy sajátos leírási módját, amelyben a műveletvégzésért és a vezérlésért felelős áramköri részletek viselkedését egy jól áttekinthető, összevont modellben írhatjuk le. Ennek részleteit a 4.8. pontban tárgyaljuk.

Egyes szakirodalmi források a fenti három megoldás közül sokszor csak az első kettővel foglalkoznak, sőt, az is előfordul, hogy a kimeneti bufferek miatti erőforrásigény-növekményből adódóan a one-shot típusú leírási módot egyenesen hibás kódolási technikának aposztrofálják, nem részletezve annak előnyeit. A fentiek alapján azonban belátható, hogy a három leírási módszer közül nem tudunk kiválasztani egyetlen egyet, amely minden körülmények között a legjobb. Az egyes megoldásokból szintetizált áramkörök tulajdonságait ismerve kell eldöntenünk, hogy egy adott esetben melyik a legpraktikusabb.[41] Amikor egy leírás „praktikusságáról" beszélünk, akkor természetes, hogy elsősorban a szintetizált áramkör erőforrásigényére és számítási teljesítményére gondolunk, de vegyük figyelembe, hogy a kódbázis olvashatósága, érthetősége sok esetben szintén fontos tényező lehet, különösen ha csapatban dolgozunk, és az általunk írt kódot később esetleg egy másik RTL tervezőnek kell megértenie vagy javítania. Az olvashatóság ugyanakkor szubjektív dolog, ráadásul elképzelhető, hogy egy bizonyos funkció esetén az egyik, másik funkció esetén a másik leírási módszer bizonyul átláthatóbbnak.

[41] A HDL tervezésre általánosságban igaz, hogy szerencsére könnyedén kaphatunk azonnali visszajelzést a helyes kódolási technikával kapcsolatban a végszó kimondására jogosult szintézis szoftverektől. Ha nem vagyunk biztosak abban, hogy egy kódrészlet, egy nyelvi szerkezet szintetizálható-e, vagy hogy úgy szintetizálódik-e, ahogy elképzeltük, akkor készítsünk belőle egy „lecsupasztított" példamodellt – az ebben a könyvben bemutatott példák többsége is ilyen –, és vessük alá az automatizált szintézis próbájának. Ha kellően leegyszerűsítjük a problémát, akkor a szintetizált netlista vizsgálatával meggyőződhetünk róla, hogy helyesen jártunk-e el a kód megszerkesztésénél.

4.6 Újrafelhasználhatóság, generikus modellek

A digitális rendszerek fejlesztési folyamatának hatékonysága, gyorsasága szempontjából az újrafelhasználható áramköri részleteknek, az ún. IP magoknak *(Intellectual Property core)* kulcsfontosságú szerepük van. Ha egy adott funkciót megvalósító áramkör RTL modelljét egyszer már elkészítettük, és ami talán még fontosabb, hogy verifikáltuk is, tehát igazoltuk, hogy helyesen működik, akkor azt a modellt célszerű újra és újra felhasználni, nem érdemes minden alkalommal újra feltalálni a kereket. Az újrafelhasználhatóságra tehát minden HDL modell esetén érdemes hangsúlyt fektetni. Két dolgot kell szem előtt tartanunk, ha HDL modelljeink újrafelhasználhatóságát biztosítani akarjuk:

- **Generikus megfogalmazás**: Törekedjünk arra, hogy HDL modelljeinkben, amit csak lehet, paraméteresen fogalmazzunk meg. Ha egy egyszerű összeadóról van szó, akkor az operandusok mérete legyen paraméter, ha egy memóriáról, akkor a címbusz és az adatbusz mérete legyen változtatható. Ez a fajta generikusság az objektumorientált programozási nyelvekből már ismerős lehet, ott az ilyen jellegű rugalmasságot az ún. sablonosztályok segítségével érhetjük el. A HDL-ek biztosítanak nyelvi elemeket, amelyekkel generikus modelleket állíthatunk elő. Ezeknek a nyelvi elemeknek a használatát ebben az alfejezetben példákkal fogjuk szemléltetni.
- **Technológiafüggetlenség**: HDL modelljeinkben csak olyan nyelvi szerkezeteket használjunk az áramkörök leírására, amelyek standard cellás és programozható logikai eszközben is megvalósíthatók, továbbá – amennyire lehet – kerüljük a konkrét cellakönytárbeli elemek közvetlen példányosítását. A technológiafüggetlenség kritériumának eleget tenni sokkal nehezebb, mint generikussá tenni egy modellt. Ennek az az oka, hogy a különböző technológiákon megvalósított áramkörök némileg eltérhetnek; amit megtehetünk egy standard cellás áramkörben, az esetleg problémát okozhat egy FPGA-ban és fordítva. Tovább nehezíti a helyzetet, hogy a különböző gyártóktól származó programozható logikai eszközök architektúrája sem egyezik meg teljesen, így egy adott HDL modell hatékony lehet az egyik eszközben, míg a másikban kevésbé. A hab a tortán pedig az, hogy a különböző logikai szintézis szoftverek „ízlése" is eltérhet a HDL modellekben alkalmazott nyelvi szerkezetekkel kapcsolatban, amire kiváló példa a VHDL 2008-as verziójában bevezetett több, igen hasznos és kényelmes újítás, amit egyes szintézisszoftverek máig nem támogatnak, igencsak megnehezítve a technológiafüggetlen, újrafelhasználható RTL modellek fejlesztőinek életét.

A 4-51. ábrán bemutatott dual-port memóriamodul esetén kézenfekvő a tárolt bejegyzések számát és azok bitszélességét is ún. szintézis-paraméterként megfogalmazni, amit VHDL-ben a `generic`, Verilogban pedig a `parameter` utasítás tesz lehetővé. Ezt mutatja be a 4-64. ábra.

Az újrafelhasználható HDL modellekkel kapcsolatban fontos különbséget tenni a példában is látható szintézis-paraméterek, valamint a futásidejű paraméterek között. A szintézis-paraméterek a HDL modell azon tulajdonságai, amelyeket a példányosításkor,

VHDL

```vhdl
entity dual_port_ram is
  generic (
    awidth: integer range 2 to 16;
    dwidth: integer range 4 to 32
  );

  port (
    clk, we:    in  std_logic;
    waddress:   in  unsigned (awidth-1 downto 0);
    raddress:   in  unsigned (awidth-1 downto 0);
    data_in:    in  std_logic_vector (dwidth-1 downto 0);
    data_out:   out std_logic_vector (dwidth-1 downto 0)
  );
end entity dual_port_ram;

architecture rtl of dual_port_ram is

  type content_t is array (0 to 2**awidth-1) of
    std_logic_vector (dwidth-1 downto 0);
  signal content: content_t;

begin

  process ( clk )
  begin
    if ( rising_edge(clk) ) then
      if ( we = '1' ) then
        content(to_integer(waddress)) <= data_in;
      end if;
      data_out <= content(to_integer(raddress));
    end if;
  end process;

end architecture rtl;
```

Verilog

```verilog
module dual_port_ram
#(
  parameter awidth,
  parameter dwidth
)(
  input   logic clk, we,
  input   logic [awidth-1:0] waddress,
  input   logic [awidth-1:0] raddress,
  input   logic [dwidth-1:0] data_in,
  output  logic [dwidth-1:0] data_out
);

  logic [dwidth-1:0] content [2**awidth-1:0];

  always @ ( posedge clk )
  begin
    if ( we == 1'b1 )
      content[waddress] <= data_in;
    data_out <= content[raddress];
  end

endmodule
```

4-64. ábra. *Generikus dual-port memória*

a toplevel integráció során kell meghatároznunk (pl. a memória mérete), míg a futásidejű paraméterek olyan változtatható beállítások, amelyeket a fizikailag megvalósított áramkör működése közben, valamely külső interfészen keresztül módosíthatunk (pl. egy UART soros–párhuzamos átalakító bitsebessége). A különbség abban áll, hogy a szintézis-paraméterek konkrét értékét a logikai szintézis előtt határozzuk meg, így tulajdonképpen a szintetizált netlistában már semmi nem utal arra, hogy ezek a paraméterek a HDL modellben még léteztek (erről az automatizált logikai szintézis folyamata során, az ún. *generic folding* nevű művelet gondoskodik, lásd 8.2.1.1. pont), míg a futásidejű paramétereket megvalósító áramköri részletek a szintetizált áramkörnek is részét képezik. A szintézis- és futásidejű paraméterek közötti különbség szemléltetéséhez vegyük például a 4-65. ábrán látható számláló áramkört.

Ebben a modellben a számlálás irányát futásidejű paraméterként valósítottuk meg, hiszen azt egy külső vezérlőjel segítségével, a szintetizált áramkör működése során bármikor megváltoztathatjuk. A 4-66. ábrán ugyanannak a számláló áramkörnek a modellje látható, de ezúttal a számlálás irányát szintézis-paraméterként valósítottuk meg.

Szintézis előtt a szintézis-paramétereknek konkrét értéket kell adnunk. Abból adódóan, hogy a paraméterek meghatározása után azok valójában csupán egy-egy konstans értéket képviselnek az előfeldolgozott kódban, a szintézis eredménye jelentősen el fog térni attól, amit akkor kapunk, ha ugyanazokat a paramétereket futásidejű paraméterként valósítanánk meg. A 4-67. ábra a számláló áramkör két verziója által meghatározott áramköri

VHDL

```vhdl
entity counter is
  port (
    clk, rst:           in  std_logic;
    load, ena_cnt, dir: in  std_logic;
    cin:                in  unsigned (7 downto 0);
    cout:               out unsigned (7 downto 0)
  );
end entity gen_counter_vhdl;

architecture rtl of counter is
begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then cout <= X"00";
    elsif ( rising_edge(clk) ) then
      if ( load = '1' ) then cout <= cin;
      else
        if ( ena_cnt = '1' ) then
          if ( dir = '1' ) then cout <= cout + 1;
          else cout <= cout - 1;
          end if;
        end if;
      end if;
    end if;
  end process;

end architecture rtl;
```

Verilog

```verilog
module counter (
  input   logic clk, rst,
  input   logic load, ena_cnt, dir,
  input   logic [7:0] cin,
  output  logic [7:0] cout
);

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) cout <= 8'd0;
    else
      if ( load == 1'b1 ) cout <= cin;
      else
        if ( ena_cnt == 1'b1 )
          if ( dir == 1'b1 ) cout <= cout + 8'd1;
          else cout <= cout - 8'd1;
  end

endmodule
```

4-65. ábra. *A számlálás iránya futásidejű paraméter*

VHDL

```vhdl
entity counter is
  generic (dir:         std_logic);

  port (
    clk, rst:       in  std_logic;
    load, ena_cnt:  in  std_logic;
    cin:            in  unsigned (7 downto 0);
    cout:           out unsigned (7 downto 0)
  );
end entity counter;

architecture rtl of counter is
begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then cout <= X"00";
    elsif ( rising_edge(clk) ) then
      if ( load = '1' ) then cout <= cin;
      else
        if ( ena_cnt = '1' ) then
          if ( dir = '1' ) then cout <= cout + 1;
          else cout <= cout - 1;
          end if;
        end if;
      end if;
    end if;
  end process;

end architecture rtl;
```

Verilog

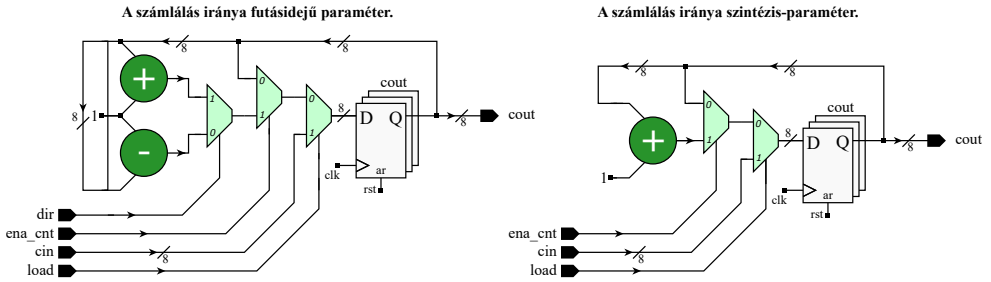
```verilog
module counter
#(parameter dir)
(
  input   logic clk, rst,
  input   logic load, ena_cnt,
  input   logic [7:0] cin,
  output  logic [7:0] cout
);

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) cout <= 8'd0;
    else
      if ( load == 1'b1 ) cout <= cin;
      else
        if ( ena_cnt == 1'b1 )
          if ( dir == 1'b1 ) cout <= cout + 8'd1;
          else cout <= cout - 8'd1;
  end

endmodule
```

4-66. ábra. *A számlálás iránya szintézis-paraméter*

szerkezetet hasonlítja össze úgy, hogy a második esetben a `dir` paraméter értékét a példányosításkor logikai magas szintre állítottuk.



4-67. ábra. *A szintézis- és futásidejű paraméterek hatása az áramköri szerkezetre*

A fenti példa alapján belátható, hogy a futásidejű paraméterek megvalósításához nincs szükségünk speciális nyelvi elemekre. Figyeljük meg, hogy mindkét nyelv a modell interfészének részeként kezeli a szintézis-paramétereket, hiszen ezeket a modul felhasználójának kell meghatároznia a beültetéskor, mint ahogy az interfészjelek bekötése is ekkor történik meg. A 4-68. ábra a generikus számláló áramkör példányosítását mutatja be.

VHDL

```
...
COUNTER_1:  entity work.counter(rtl)
              generic map (dir => '1')
              port map (
                clk      => global_clock,
                rst      => global_reset,
                load     => load_counter_1,
                ena_cnt  => enable_counter_1,
                cin      => cin_2_counter_1,
                cout     => counter_1_cout
              );
...
```

Verilog

```
...
counter #(.dir(1'b1))
COUNTER_1 (
  .clk(global_clock),
  .rst(global_reset),
  .load(load_counter_1),
  .ena_cnt(enable_counter_1),
  .cin(cin_2_counter_1),
  .cout(counter_1_cout)
)
...
```

4-68. ábra. *A generikus számláló példányosítása*

Ebben a példában azt figyelhetjük meg, hogy a szintézis-paraméter konkrét értéke közvetetten meghatározza a szintetizált áramköri szerkezetet. Az ún. feltételes példányosítás utasítások segítségével ugyanezt sokkal közvetlenebb módon is kifejezhetjük. Tegyük fel, hogy a fejezet bevezetőjében említett, négy egész szám összeadására képes akkumulátor áramkörök mindegyikének HDL modellje rendelkezésünkre áll, feladatunk pedig egy újrafelhasználható és a lehető legszélesebb körben alkalmazható IP elkészítése, amelyben szükségünk van erre az akkumulátor funkcióra. Mivel nem ismerjük az általunk készített HDL modell leendő felhasználásának pontos körülményeit, nem tudhatjuk, hogy a három lehetséges akkumulátor-modell közül melyik lesz a legjobb; a minimális ciklusszámú (`acc_nopipeline`), a maximális órajelfrekvenciájú (`acc_pipeline`) vagy a minimális erőforrásigényű (`acc_iter`) megoldás. Ebben az esetben a feltételes példányosítás segítségével létrehozhatunk egy burkolómodult, amelyben egy szintézis-paraméter értékétől tesszük függővé, hogy éppen melyik akkumulátort példányosítjuk (lásd 4-69. ábra).

VHDL

```vhdl
entity acc_wrapper is
  generic (
    adder_version:      integer range 0 to 2
  );

  port (
    clk, rst, start: in  std_logic;
    finish:          out std_logic;
    A, B , C, D:     in  unsigned (31 downto 0);
    SUM:             out unsigned (31 downto 0);
    Z:               out std_logic
  );
end entity acc_wrapper;

architecture rtl of acc_wrapper is
begin

  ACC_GENERATE: if adder_version = 0 generate
    A1: entity work.acc_nopipeline(rtl)
          port map (clk,rst,A,B,C,D,SUM,Z);
    finish <= '0';
  elsif adder_version = 1 generate
    A1: entity work.acc_pipeline(rtl)
          port map (clk,rst,A,B,C,D,SUM,Z);
    finish <= '0';
  elsif adder_version = 2 generate
    A1: entity work.acc_iter(rtl)
          port map (clk,rst,start,finish,
                    A,B,C,D,SUM,Z);
  end generate;

end architecture rtl;
```

Verilog

```verilog
module acc_wrapper
#(
  parameter adder_version
)(
  input   logic         clk, rst, start,
  output  logic         finish,
  input   logic [31:0]  A, B, C, D,
  output  logic [31:0]  SUM,
  output  logic         Z
);

  generate

    if ( adder_version == 0 ) begin
      acc_nopipeline A1(clk,rst,A,B,C,D,SUM,Z);
      assign finish = 1'b0;
    end else begin
      if ( adder_version == 1 ) begin
        acc_pipeline A1(clk,rst,A,B,C,D,SUM,Z);
        assign finish = 1'b0;
      end else acc_iter A1(clk,rst,start,finish,
                           A,B,C,D,SUM,Z);
    end

  endgenerate

endmodule
```

4-69. ábra. *Akkumulátor burkolómodul feltételes példányosítással*

VHDL

```vhdl
entity shiftreg is
  generic (n:  integer range 4 to 32);

  port (
    clk, rst: in  std_logic;
    ser_in:   in  std_logic;
    par_out:  out std_logic_vector (n-1 downto 0)
  );
end entity shiftreg;

architecture rtl of shiftreg is
begin

  FIRST_REG:  entity work.d_flip_flop(rtl)
                  port map (clk, rst,
                            ser_in, par_out(0));

  GEN: for i in 1 to n-1 generate
    REG:  entity work.d_flip_flop(rtl)
            port map (clk, rst,
                      par_out(i-1), par_out(i));
  end generate;

end architecture rtl;
```

Verilog

```verilog
module shiftreg #(parameter n = 8)
(
  input   logic clk, rst,
  input   logic ser_in,
  output  logic [n-1:0] par_out
);

  d_flip_flop FIRST_REG(clk, rst,
                        ser_in, par_out[0]);

  genvar i;
  generate
    for ( i = 1; i < n; i = i + 1 ) begin: GEN
      d_flip_flop REG(clk, rst,
                      par_out[i-1], par_out[i]);
    end
  endgenerate

endmodule
```

4-70. ábra. *Generikus léptetőregiszter `for`-generate utasítással megvalósítva*

A fenti példánál persze figyelembe kell venni, hogy a három akkumulátor viselkedése, sőt interfésze is kissé eltér, ami azt jelenti, hogy a burkolómodullal kölcsönhatásba lépő, azt vezérlő, vagy az által meghajtott egyéb HDL moduloknak képesnek kell lenniük bármelyikkel együttműködni.

A feltételes példányosítás mellett létezik ciklikus példányosítás (`for-generate`) utasítás is, amellyel hatékonyan tudunk létrehozni sok példányt egy modulból úgy, hogy a példányosított modulok száma szintézis-paraméter is lehet. A 4-70. ábra egy léptetőregiszter generikus modelljét mutatja be, amelyben a regiszter flip-flopjainak a számát egy szintézis-paraméter határozza meg (vö. 4-27. ábra). Mivel itt is példányosítás utasításról van szó, természetesen feltételezzük, hogy a `d_flip_flop` nevű modul létezik.

4.7 Számláló-alapú funkcionális egységek

A gyakorlatban rengeteg olyan problémával találkozhatunk, amelyek egyszerű számlálók segítségével kezelhetők. Az alábbi fejezetben a teljesség igénye nélkül bemutatunk néhány gyakran használt, az előzőekhez képest kicsit összetettebb funkcionális egységet, amelyek központi eleme egy-egy számláló áramkör.

A 4-71. ábrán látható *watchdog* áramkör feladata, hogy egy normális működési körülmények között periodikus négyszögjelről megállapítsa, hogy aktív-e vagy sem, vagyis hogy egy adott időintervallumon belül detektálható-e rajta felfutó vagy lefutó él. Periodikus négyszögjeleket gyakran használunk annak jelzésére, hogy egy adott funkcionális egység működik-e, az ilyen jeleket *„heartbeat"* jeleknek hívjuk.

A watchdog egy felfutó és lefutó élekre egyaránt érzékeny éldetektorból és egy lefelé számláló áramkörből áll. A `signal_in` bemeneten tapasztalható bármilyen változás a számláló áramkört annak alapértékére állítja be. Amíg a `signal_in` bemeneten nincs változás, addig a számláló értéke minden órajel-felfutóélnél eggyel csökken. Ha a `signal_in` bemeneten olyan sokáig nem történik változás, hogy a számláló el tudja érni a nulla értéket, akkor a `signal_in_active` kimenet logikai alacsony szintre áll be, és úgy is marad mindaddig, amíg a `signal_in` jel meg nem változik. Figyeljük meg, hogy amint a számláló elérte a nulla értéket, a visszafelé számlálás leáll. Azokat a számlálókat, amelyek egy szélsőérték elérése után nem fordulnak át, hanem leállnak, szaturációs számlálóknak nevezzük. A watchdog áramkör egyetlen szintézis-paramétere a számláló alapértéke (`counter_max`), amelyet a `signal_in` jel változásai hatására betöltünk, és amely a lokális órajel periódusidejével együtt meghatározza, hogy a `signal_in` jelnek mekkora időtartamig kell változatlannak lennie ahhoz, hogy a watchdog áramkör jelezze az inaktivitást. E példaáramkörnél – és az ebben az alfejezetben bemutatott többi megoldás esetén is – fontos kiemelni, hogy bemenete az áramkör szempontjából ún. statikus jel, vagyis a lokális órajelünk periódusideje sokkal kisebb, mint a vizsgált jel változásai, élei között eltelt jellemző időtartamok.

A legösszetettebb digitális rendszer is „prototípusként kezdi pályafutását". A HDL-ekre épülő tervezési módszer, a könnyen, akár ingyenesen is hozzáférhető szoftverrendszerek, valamint a programozható logikai eszközök és az azokhoz kifejlesztett, ugyancsak olcsón elérhető fejlesztőkártyák együtt lehetővé teszik azt, hogy akár igen összetett digitális áramköröket is olcsón, gyakorlatilag bármiféle laborfelszerelés nélkül meg tudjunk valósítani. Az ilyen, programozható logikai eszközökben megvalósított prototípusokban szinte mindig találunk nyomógombokat, amelyekkel próbaáramkörünk működését befolyásolni igyekszünk. A nyomógombokról azt feltételezzük, hogy megnyomott állapotban az egyik,

VHDL

```
architecture rtl of watchdog is

  signal signal_in_d:       std_logic;
  signal signal_in_event:   std_logic;
  signal counter: integer range 0 to counter_max;

begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      signal_in_d <= '0';
    elsif ( rising_edge(clk) ) then
      signal_in_d <= signal_in;
    end if;
  end process;
  signal_in_event <= signal_in xor signal_in_d;

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      counter <= counter_max;
    elsif ( rising_edge(clk) ) then
      if ( signal_in_event = '1' ) then
        counter <= counter_max;
      else
        if ( counter > 0 ) then
          counter <= counter - 1;
        end if;
      end if;
    end if;
  end process;
  signal_in_active <= '1' when counter /= 0 else
                      '0';

end architecture rtl;
```

Verilog

```
module watchdog #(...)(...);

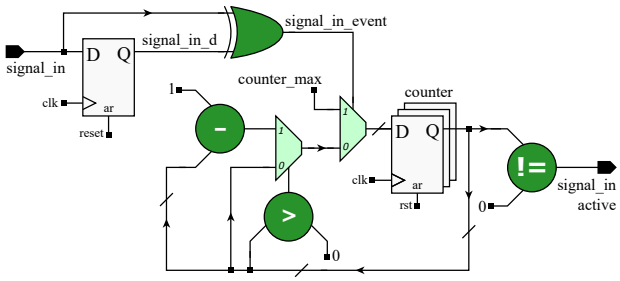
  logic signal_in_d;
  logic signal_in_event;
  logic [$clog2(counter_max)-1:0] counter;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) signal_in_d <= 1'b0;
    else signal_in_d <= signal_in;
  end
  assign signal_in_event = signal_in ^ signal_in_d;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) counter <= 0;
    else begin
      if ( signal_in_event == 1'b1 ) begin
        counter <= counter_max;
      end else begin
        if ( counter > 0 ) counter <= counter - 1;
      end
    end
  end
  assign signal_in_active = (counter != 0) ?
                            1'b1 : 1'b0;

endmodule
```

Áramköri struktúra



4-71. ábra. *Watchdog áramkör*

elengedett állapotban a másik logikai szintet kapcsolják áramkörünk bemenetére. A helyzet az esetek többségében sajnos ennél kényelmetlenebb. Mechanikai kialakításukból adódóan a nyomógombok megnyomás vagy felengedés hatására nem azonnal változtatják meg kimenetük értékét, hanem „peregnek”, a logikai magas és alacsony szintek sorozatát produkálják, mielőtt beállnának egy stabil állapotba. A pergés frekvenciája és időtartama is függ a konkrét nyomógombtípustól, de a fent említett fejlesztőkártyákon gyakran alkalmazott, felületszerelt eszközök esetén 10 kHz-es nagyságrendű frekvenciájú, néhányszor tíz periódus hosszúságú pergésre lehet számítani. Áramkörünk belső logikája nem feltétlenül alkalmas ennek a viselkedésnek a tolerálására, ezért a nyomógomb-bemeneteket ún.

VHDL

```
architecture rtl of debouncer is

  signal signal_in_d:     std_logic;
  signal signal_in_dd:    std_logic;
  signal signal_in_ddd:   std_logic;
  signal signal_in_event: std_logic;
  signal counter: integer range 0 to counter_max;

begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      signal_in_d <= '0';
      signal_in_dd <= '0';
      signal_in_ddd <= '0';
    elsif ( rising_edge(clk) ) then
      signal_in_d <= signal_in;
      signal_in_dd <= signal_in_d;
      signal_in_ddd <= signal_in_dd;
    end if;
  end process;
  signal_in_event <= signal_in_dd xor signal_in_ddd;

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      counter <= 0;
    elsif ( rising_edge(clk) ) then
      if ( signal_in_event = '1' ) then
        counter <= 0;
      elsif ( counter < counter_max ) then
        counter <= counter + 1;
      end if;
    end if;
  end process;

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      signal_out <= '0';
    elsif ( rising_edge(clk) ) then
      if ( counter = counter_max ) then
        signal_out <= signal_in_ddd;
      end if;
    end if;
  end process;

end architecture rtl;
```

Verilog

```
module debouncer #(...)(...);

  logic signal_in_d, signal_in_dd, signal_in_ddd;
  logic signal_in_event;
  logic [$clog2(counter_max)-1:0] counter;

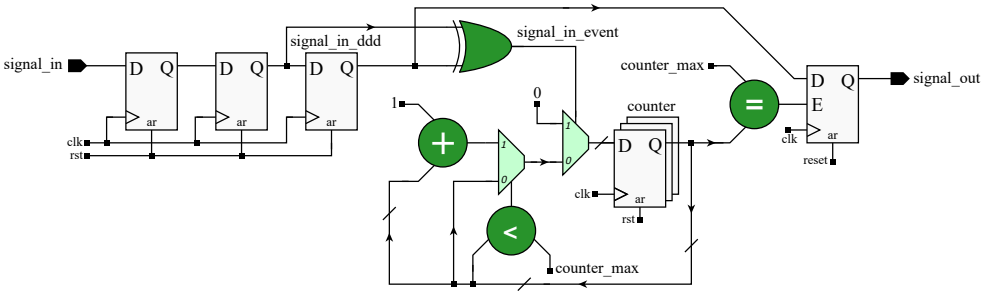
  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) begin
      signal_in_d <= 1'b0;
      signal_in_dd <= 1'b0;
      signal_in_ddd <= 1'b0;
    end else begin
      signal_in_d <= signal_in;
      signal_in_dd <= signal_in_d;
      signal_in_ddd <= signal_in_dd;
    end
  end
  assign signal_in_event = signal_in_dd ^
                           signal_in_ddd;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) counter <= 0;
    else begin
      if ( signal_in_event == 1'b1 ) counter <= 0;
      else if ( counter < counter_max )
        counter <= counter + 1;
    end
  end

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) signal_out <= 1'b0;
    else if ( counter == counter_max )
      signal_out <= signal_in_ddd;
  end

endmodule
```

Áramköri struktúra



4-72. ábra. *Pergésmentesítő áramkör*

pergésmentesítő *(debouncer)* áramkörökkel szoktuk „megszelídíteni". A pergésmentesítő áramkörök lényege, hogy a bemenetükön tapasztalható tranzienst, a pergést elnyomják, kimenetükön már csak egyetlen jelváltás jelenik meg. A 4-72. ábrán látható példa egy ilyen pergésmentesítő áramkört mutat be.

A pergésmentesítő áramkör egy, az előző példában láthatóval azonos éldetektorból, egy inkrementáló szaturációs számlálóból és egy kimeneti bufferből áll (a bemeneti éldetektor elé kapcsolt két flip-flop szerepét a 6.3. alfejezetben részletezzük). Reset hatására az áramkör összes flip-flopját nullára inicializáljuk. A reset jel elengedését követően a számláló felfelé kezd számolni. Ha elérte a szintézis-paraméterként megadott maximális értékét, akkor a komparátor kimenete egybe vált, ami által az éldetektor flip-flopjának kimenete – gyakorlatilag a bemenet értéke – a kimeneti bufferbe íródik át. Ha mindez idő alatt a bemenet nem változott meg, tehát nem történt gombnyomás, akkor a kimenet értéke tulajdonképpen végig nulla marad. Ha ekkor a bemeneten felfutó vagy lefutó él jelenik meg, akkor az éldetektor nullázza a számlálót, így a kimeneti komparátor tartás üzemmódba kapcsolja át a kimeneti buffert, a bemenet megváltozása tehát nem jut ki a kimenetre. Mindaddig, amíg a bemenet pereg, a számláló folyton nullázódik, így a kimeneti buffer is folyamatosan tartás üzemmódban lesz. Ha a bemeneten tapasztalható változások között eltelik annyi idő, hogy a számláló ez alatt el tudja érni maximális értékét, akkor a pergés véget ért, a komparátor kimenete ismét egybe vált, és a kimeneti flip-flop mintát vesz az éldetektor flip-flopjáról, ami az immár stabil bemenetet jelenti. A számlálónak a `counter_max` szintézis-paraméterben megadott maximális értékét a lokális órajel periódusidejének ismeretében úgy kell megválasztani, hogy a nyomógombra jellemző pergési periódus rövidebb legyen, mint az az időtartam, ami alatt a számláló eléri maximális értékét.

A 4-73. ábrán látható példaáramkör segítségével események közötti időtartam mérése sét végezhetjük el úgy, hogy az eseményeket képviselő felfutó élek közötti időtartamban megszámoljuk az ismert periódusidejű lokális órajelünk felfutó éleit. Ha a bemenetre kapcsolt jel periodikus, akkor tulajdonképpen ez a módszer frekvenciamérésként is felfogható, amely a mért jel periódusidejének növekedésével, vagy – ami ezzel egyenértékű – a lokális órajel periódusidejének csökkenésével egyre pontosabb.

Az áramkör a már jól ismert felfutóél-detektorból, egy inkrementáló szaturációs számlálóból és egy kimeneti bufferből áll. Reset hatására mind a számlálót, mind a kimeneti buffert nullázzuk. A reset elengedésekor a számláló értéke minden órajel-felfutóélnél eggyel nő. Ha a számlálás közben egy felfutó él jelenik meg a bemeneten, akkor az éldetektor kimenetén megjelenő impulzus hatására a számláló aktuális értéke áttöltődik a kimeneti bufferbe, és ezzel egyidőben a számláló értéke nullázódik. A pontos működéshez természetesen a bemeneten gyorsabban kell hogy kövessék egymást a felfutó élek, mint amilyen gyorsan a számláló eléri maximális értékét. A fenti áramkör használható például frekvenciamodulált jelek demodulációjára, ahogy azt a következő példában is láthatjuk.

Tegyük fel, hogy két, egymástól nagy távolságra lévő alrendszer között kell átvinnünk egy kétállapotú engedélyező jelet. Nagymegbízhatóságú rendszerekben fel kell készülnünk arra az esetre, hogy az összeköttetést megvalósító alkatrészek hibájából adódóan az engedélyező jelet hordozó vezeték tápfeszültségre vagy földpotenciálra kerül a meghajtó rendszer kimenetének állapotától függetlenül (pl. egy vonali meghajtó tápfeszültsége megszűnik;

VHDL

```
architecture rtl of delay_meter is

  signal signal_in_d:       std_logic;
  signal signal_in_dd:      std_logic;
  signal signal_in_ddd:     std_logic;
  signal signal_in_rising:  std_logic;
  signal counter: integer range 0 to counter_max;

begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      signal_in_d <= '0';
      signal_in_dd <= '0';
      signal_in_ddd <= '0';
    elsif ( rising_edge(clk) ) then
      signal_in_d <= signal_in;
      signal_in_dd <= signal_in_d;
      signal_in_ddd <= signal_in_dd;
    end if;
  end process;
  signal_in_rising <= signal_in_dd and
                      not signal_in_ddd;

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      counter <= 0;
    elsif ( rising_edge(clk) ) then
      if ( signal_in_rising = '1' ) then
        counter <= 0;
      else
        if ( counter < counter_max ) then
          counter <= counter + 1;
        end if;
      end if;
    end if;
  end process;

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      delay <= X"0000";
    elsif ( rising_edge(clk) ) then
      if ( signal_in_rising = '1' ) then
        delay <= to_unsigned(counter, 16);
      end if;
    end if;
  end process;

end architecture rtl;
```

Verilog

```
module delay_meter #(...)(...);

  logic signal_in_d, signal_in_dd, signal_in_ddd;
  logic signal_in_rising;
  logic [$clog2(counter_max)-1:0] counter;

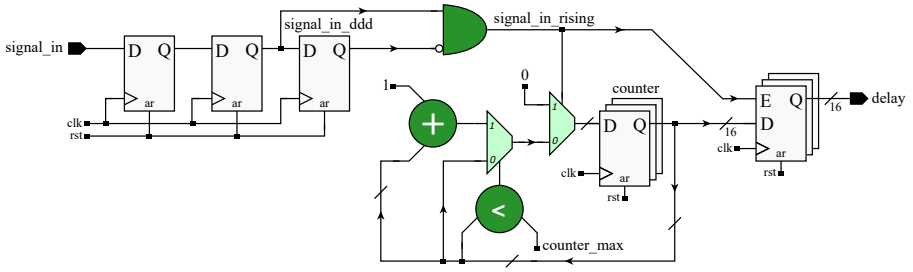
  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) begin
      signal_in_d <= 1'b0;
      signal_in_dd <= 1'b0;
      signal_in_ddd <= 1'b0;
    end else begin
      signal_in_d <= signal_in;
      signal_in_dd <= signal_in_d;
      signal_in_ddd <= signal_in_dd;
    end
  end
  assign signal_in_rising = signal_in_dd &
                            ~signal_in_ddd;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) counter <= 0;
    else
      if ( signal_in_rising == 1'b1 ) counter <= 0;
      else if ( counter < counter_max )
        counter <= counter + 1;
  end

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) delay <= 0;
    else if ( signal_in_rising == 1'b1 )
      delay <= counter;
  end

endmodule
```

Áramköri struktúra



4-73. ábra. *Késleltetésmérő áramkör*

egy csatlakozóban valamilyen fizikai behatás miatt rövidzár keletkezik stb.). A rendszer megbízhatóságát jelentősen megnöveli, ha a jelet fogadó áramkör meg tudja állapítani, hogy az adó oldal szándéka szerint, vagy a fentihez hasonló hiba okán látja azt a fogadott jelen, amit lát. Vegyük észre, hogy ebben az esetben valójában két bit információt akarunk átvinni egyetlen vezetéken: egyrészt hogy mi a logikai érték, másrészt pedig hogy ez szándékos-e. Két bit információ átvitele egyetlen vezetéken nem kivitelezhetetlen, megtehetjük, hogy az adó áramkör nem egyszerűen egy jelszintet bocsát ki, hanem egy négyszögjelet, a logikai szintekhez egy-egy jól megkülönböztethető frekvenciaértéket rendelve. Ez gyakorlatilag a kétállapotú frekvenciamoduláció (*Binary Frequency Shift Keying*, BFSK) elvének egy megvalósítása. A vevő oldal a korábban bemutatott watchdog és késleltetésmérő áramkörök kombinálásával meg tudja állapítani, hogy a jel aktív-e, és meg tudja mérni

VHDL

```
architecture rtl of fsk_logic_modulator is

  signal counter: integer range 0 to counter_max_low;

begin

  process ( clk, rst )
    variable counter_max:
             integer range 15 to counter_max_low;
  begin
    if ( rst = '1' ) then
      counter <= 0;
      fsk_out <= '0';
    elsif ( rising_edge(clk) ) then

      if ( logic_in = '0' ) then
        counter_max := counter_max_low;
      else
        counter_max := counter_max_high;
      end if;

      if ( counter < counter_max ) then
        counter <= counter + 1;
      else
        fsk_out <= not fsk_out;
        counter <= 0;
      end if;

    end if;
  end process;

end architecture rtl;
```

Verilog

```
module fsk_logic_modulator #(...)(...);

  logic [$clog2(counter_max_low):0] counter;
  logic [$clog2(counter_max_low):0] counter_max;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) begin
      counter <= 0;
      fsk_out <= 1'b0;
    end else begin

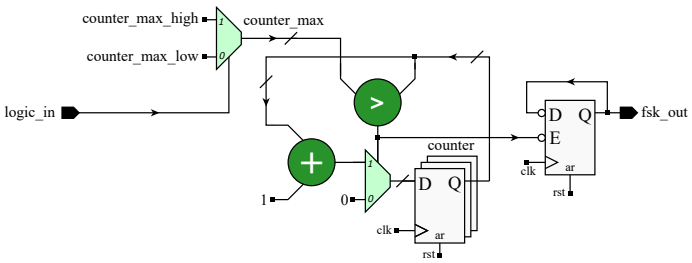
      if ( logic_in == 1'b0 )
       counter_max = counter_max_low;
      else counter_max = counter_max_high;

      if ( counter < counter_max ) counter <=
                                     counter + 1;
      else begin
        fsk_out <= ~fsk_out;
        counter <= 0;
      end

    end
  end

endmodule
```

Áramköri struktúra



4-74. ábra. *BFSK logikai modulátor áramkör*

annak frekvenciáját is, amiből a jel logikai értéke is kiderül. A 4-74. ábra egy ilyen áramkör-páros adó-oldali megvalósítását mutatja be.

A modulátor magja ezúttal is egy számláló, amely az órajel minden felfutó élénél eggyel inkrementálódik. Amikor a számláló elérte maximális értékét, a kimeneti buffert megnegáljuk, így tulajdonképpen a kimeneten egy, a számláló maximális értékétől függő frekvenciájú, leosztott órajelet látunk. A számláló maximális értéke az áramkör bemenetét jelentő logikai jeltől függ, tehát a kimeneti jel frekvenciája két diszkrét értéket vehet fel. E két értéket (`counter_max_high`, `counter_max_low`) szintézis-paraméterként megvalósítva az áramkör két jellemző kimeneti frekvenciáját az adott alkalmazás igényei szerint beállíthatjuk.

A 4-75. ábra a vevő-oldali, lényegében BFSK demodulációt végző áramkört mutatja be.

A demodulátor a modulált jelet egy felfutó és lefutó élekre is érzékeny éldetektorral fogadja. Az éldetektor kimenetét arra használjuk, hogy amikor változást észlelünk a bemeneti jelen, akkor egy számláló értékét nullázzuk. Ha két nullázás között a számláló nem tudja elérni a `counter_max_low` értéket, tehát a bemeneti jel frekvenciája kellően magas, akkor az általa képviselt logikai értéket magasnak tekintjük, ellenkező esetben alacsonynak. A `logic_out` kimeneti buffert minden éldetektáláskor eszerint frissítjük. Ha két nullázás között olyan sok idő telik el, hogy a számláló a `counter_max_lost` értéket is el tudja érni, akkor a `signal_lost` kimeneti buffert egybe állítjuk, jelezve, hogy a bemeneti jel valószínűleg megszűnt (vagy legalábbis túlságosan lecsökkent a frekvenciája), így a logikai kimenet értéke nem érvényes.

A fent bemutatott modulátor és demodulátor áramkörök szintézis-paramétereit természetesen megfelelően össze kell hangolni, nem elegendő egyszerűen azonos értékre állítani őket, annál is inkább, mivel az adó és a vevő általános esetben két külön eszközben (pl. két külön FPGA-ban) működik, így órajelük sem feltétlenül egyezik meg.

Az itt bemutatott, számláló-alapú áramkörök újrafelhasználhatóságát az biztosítja, hogy a bennük lévő számlálók szaturációs értékei szintézis-paraméterként az adott alkalmazás igényei szerint beállíthatók. Látszólag tehát csak az korlátozza alkalmazhatóságukat, hogy a vizsgált jelek változásai között eltelt jellemző időtartamok csökkenésével a belső működésükből adódó késleltetések arányaiban egyre jelentősebbek lehetnek, tehát ahogy a külső, vizsgált események gyorsulnak, úgy ezek az áramkörök egyre kevésbé lesznek alkalmasak a probléma kezelésére. Ugyanakkor más alkalmazásokban az is elképzelhető, hogy a vizsgált események a lokális órajel periódusidejéhez viszonyítva nagyon lassúak. Ez azt jelenti, hogy a belső számlálók szaturációs értékeit nagyon nagyra kell állítanunk, tehát maguk a számláló áramkörök nagy méretűek lesznek. A nagyméretű számlálók hátránya, hogy a bennük lévő, kombinációs hálózatként megvalósított összeadó/kivonó logikák késleltetése igen nagyra adódhat, ami korlátozhatja az áramkör maximális órajel-frekvenciáját. Ez a probléma elvileg könnyen kezelhető úgy, hogy a számlálót tartalmazó áramkör lokális órajelét lelassítjuk, ún. órajel-előosztó fokozatokkal, amelyek a leosztatlan órajel minden n-edik felfutó élénél generálnak egy felfutó élt a kimenetükön. Elvileg megtehetjük, hogy az órajel-előosztó kimenetét közvetlenül a lelassítandó áramkör flip-flopjainak órajel-bemenetére kapcsoljuk, így egy adott időtartam alatt kevesebb órajelet kell „megszámlálni", mintha a leosztatlan, gyors órajelet használnánk, tehát kisebb méretű számlálókra lesz szükségünk. Azonban az áramkörök fizikai megvalósításából adódó nehézségek miatt az órajel-előosztó áramkörök kimenetét gyakran nem arra használjuk, hogy közvetlenül

VHDL

```
architecture rtl of fsk_logic_demodulator is
  signal fsk_in_d:        std_logic;
  signal fsk_in_dd:       std_logic;
  signal fsk_in_ddd:      std_logic;
  signal fsk_in_event:    std_logic;
  signal counter: integer range 0 to
                          counter_max_lost;
begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      fsk_in_d <= '0'; fsk_in_dd <= '0';
      fsk_in_ddd <= '0';
    elsif ( rising_edge(clk) ) then
      fsk_in_d <= fsk_in; fsk_in_dd <= fsk_in_d;
      fsk_in_ddd <= fsk_in_dd;
    end if;
  end process;
  fsk_in_event <= fsk_in_dd xor fsk_in_ddd;

  process ( clk, rst )
  begin
    if ( rst = '1' ) then counter <= 0;
    elsif ( rising_edge(clk) ) then
      if ( fsk_in_event = '1' ) then counter <= 0;
      else if ( counter < counter_max_lost ) then
             counter <= counter + 1;
           end if;
      end if;
    end if;
  end process;

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      logic_out <= '0'; signal_lost <= '1';
    elsif ( rising_edge(clk) ) then

      if ( counter >= counter_max_lost ) then
        signal_lost <= '1';
      else signal_lost <= '0'; end if;

      if ( fsk_in_event = '1' ) then
        if ( counter >= counter_max_low ) then
          logic_out <= '0';
        else logic_out <= '1'; end if;
      end if;

    end if;
  end process;

end architecture rtl;
```

Verilog

```
module fsk_logic_demodulator #(...)(...);

  logic fsk_in_d, fsk_in_dd, fsk_in_ddd;
  logic fsk_in_event;
  logic [$clog2(counter_max_lost):0] counter;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) begin
      fsk_in_d <= 1'b0;
      fsk_in_dd <= 1'b0;
      fsk_in_dd <= 1'b0;
    end else begin
      fsk_in_d <= fsk_in;
      fsk_in_dd <= fsk_in_d;
      fsk_in_ddd <= fsk_in_dd;
    end
  end
  assign fsk_in_event = fsk_in_dd ^ fsk_in_ddd;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) counter <= 0;
    else begin
      if ( fsk_in_event == 1'b1 ) begin
        counter <= 0;
      end else begin
        if ( counter < counter_max_lost )
          counter <= counter + 1;
      end
    end
  end

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) begin
      logic_out <= 1'b0; signal_lost <= 1'b1;
    end else begin

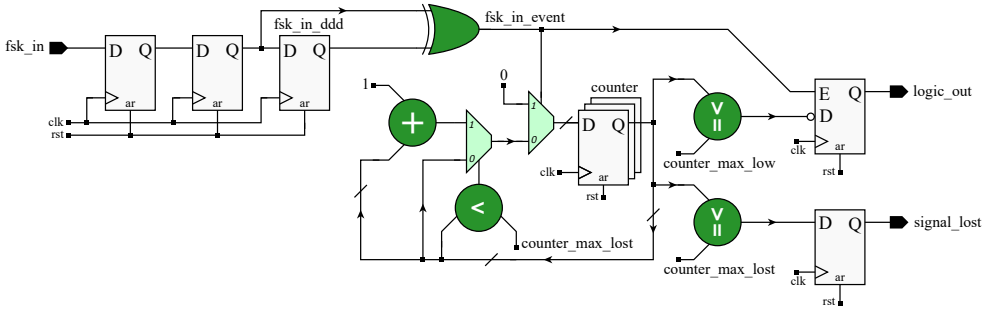
      if ( counter >= counter_max_lost )
        signal_lost <= 1'b1;
      else signal_lost <= 1'b0;

      if ( fsk_in_event == 1'b1 )
        if ( counter >= counter_max_low )
          logic_out <= 1'b0;
        else logic_out <= 1'b1;

    end
  end

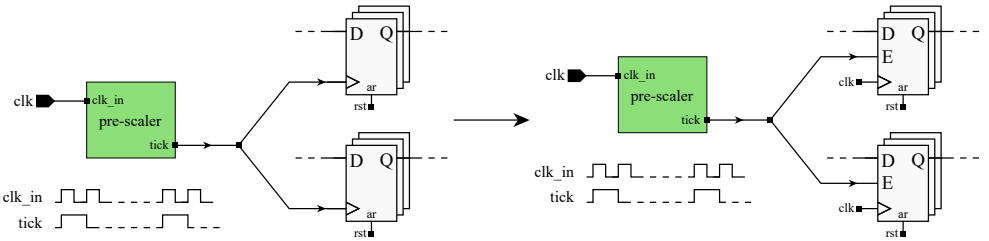
endmodule
```

Áramköri struktúra



4-75. ábra. *BFSK logikai demodulátor áramkör*

flip-flopok órajel-bemenetét hajtsuk vele, ehelyett a lelassítandó áramkör flip-flopjait továbbra is a leosztatlan órajellel hajtjuk, az órajel-előosztó kimenetét pedig a lelassítandó flip-flopok órajel-engedélyező bemenetére kötjük (lásd 4-76. ábra).[42]



4-76. ábra. *Leosztott órajelek felhasználása*

Ebben az esetben az órajel-előosztó tulajdonképpen maga is egy egyszerű számláló áramkör, amely adott időközönként egy, a leosztatlan órajel egyetlen periódusáig fennálló impulzust állít elő (`tick`), amit másodlagos órajelként is felfoghatunk, hiszen ez az az ütemjel, amely meghatározza, hogy a lelassítandó áramkörök flip-flopjai mikor „vesznek tudomást" a leosztatlan órajel felfutó éléről és mikor nem, így érve el azt a hatást, mintha maga az órajel lassult volna le. Az órajel-előosztó kimeneti impulzusai közötti időtartamot az újrafelhasználhatóság érdekében itt is érdemes szintézis-paraméterként megvalósítani (lásd 4-77. ábra).

Az órajel-előosztót tüzetesebben megvizsgálva felmerülhet a kérdés, hogy miért szükséges a kimenetét éldetektorral ellátni, hiszen a komparátor kimenete amúgy is egy egyetlen periódus hosszúságú negatív impulzus, ezt tehát elvileg elegendő lenne egyszerűen megnegálva a kimenetre kapcsolni. Abban az esetben, ha ez az áramkör önmagában elegendő a megfelelő leosztás megvalósításához, akkor ez valóban így is van. Ha azonban olyan mértékű leosztást kell létrehoznunk, amelyben már az órajel-előosztóban lévő számláló is túlságosan nagy lenne, akkor ún. többfokozatú, vagy kaszkád előosztót kell megvalósítanunk. A 4-77. ábrán látható áramkör engedélyező bemenete és kimeneti éldetektora azt a célt szolgálja, hogy kaszkád előosztót hozhassunk létre a 4-78. ábrán látható séma szerint sorba kapcsolva ennek az egyfokozatú előosztónak több példányát.

Az n-edik fokozat kimenetét az n+1-edik fokozat engedélyező bemenetére kapcsoljuk, az első fokozat engedélyező bemenetét pedig vagy logikai magas szintre állítjuk, vagy arra használjuk, hogy letilthatóvá tegyük az egész órajelosztó struktúra működését. A kaszkád előosztó eredő leosztásának mértéke így az egyes fokozatok által megvalósított leosztás szorzata lesz, tehát igen nagy leosztás érhető el úgy, hogy az egyes fokozatokban lévő számlálók nem lesznek nagy méretűek. A kimeneti éldetektor tehát azért szükséges, hogy a belső fokozatokra is igaz legyen, hogy a leosztatlan órajel egyetlen periódusáig aktív impulzusokat állítanak elő.

[42] Ez a technika inkább FPGA megvalósítás esetén jellemző. Az FPGA-kban a kifejezetten az órajel és a reset terjesztésére használható, alacsony késleltetésű és elcsúszású *(low-skew)* vezetékhálózatok, az ún. órajelfák száma (lásd 7. fejezet) korlátozott. Ha egy órajel-előosztó áramkör kimenetét közvetlenül flip-flopok órajelbemenetére kapcsoljuk, az azt jelenti, hogy már két, egymástól független órajelfát használtunk fel. Sok független, belső órajel esetén a rendelkezésre álló órajelfák száma könnyen túl kevésnek bizonyulhat.

VHDL

```
architecture rtl of prescaler is

  signal counter: integer range 0 to counter_max;
  signal limit:   std_logic;
  signal limit_d: std_logic;

begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then counter <= 0;
    elsif ( rising_edge(clk) ) then
      if ( ena = '1' ) then
        if ( counter < counter_max ) then
          counter <= counter + 1;
        else
          counter <= 0;
        end if;
      end if;
    end if;
  end process;
  limit <= '0' when counter < counter_max else '1';

  process ( clk, rst )
  begin
    if ( rst = '1' ) then limit_d <= '1';
    elsif ( rising_edge(clk) ) then
      limit_d <= limit;
    end if;
  end process;
  tick <= limit and not limit_d;

end architecture rtl;
```

Verilog

```
module prescaler #(...)(...);

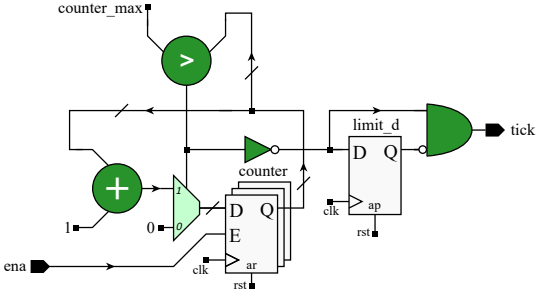
  logic [$clog2(counter_max):0] counter;
  logic limit, limit_d;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) counter <= 0;
    else
      if ( ena == 1'b1 )
        if ( counter < counter_max )
          counter <= counter + 1;
        else counter <= 0;
  end
  assign limit = ( counter < counter_max ) ?
                 1'b0 : 1'b1;

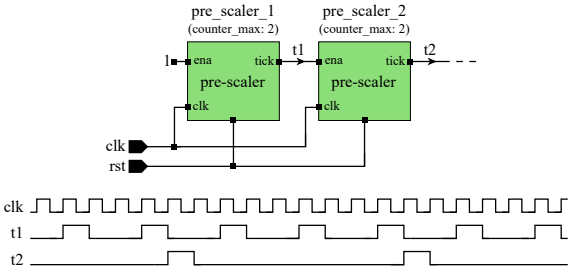
  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) limit_d <= 1'b1;
    else limit_d <= limit;
  end
  assign tick = limit & ~limit_d;

endmodule
```

Áramköri struktúra



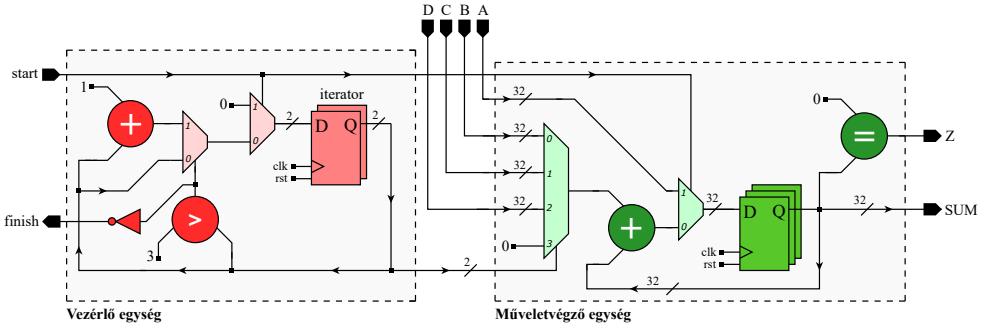
4-77. ábra. *Órajel-előosztó áramkör*



4-78. ábra. *Kaszkád előosztó*

4.8 Adatfeldolgozó rendszerek RTL modelljei

Adatfeldolgozó rendszer alatt azokat a digitális áramköröket értjük, amelyek valamilyen bemenő adathalmazon megfelelő ütemezéssel aritmetikai és/vagy logikai transzformációkat végeznek, majd a feldolgozott adatokat a kimenetükre juttatják. Adatfeldolgozó rendszernek tekinthető tehát például a 4.4.4. pontban bemutatott egyszerű iteratív akkumulátor, de egy bonyolult utasításkészletet megvalósító általános célú mikroprocesszor is. Az ilyen rendszerek belső részáramkörei feladatkörüket tekintve két csoportba sorolhatók. Az egyik csoport azokat az alkatrészeket tartalmazza, amelyek a rendszer funkciójának elvégzéséhez szükséges aritmetikai, logikai és tároló alapfeladatokat látják el (regiszterek, ALU, memória stb.). Ezek az alkatrészek a műveletvégző elemek, megfelelően összekapcsolt hálózatuk pedig a műveletvégző egység *(datapath)*. A másik csoportba tartozó komponensek összessége az ún. vezérlő egység *(control unit)*, amelynek feladata, hogy a megfelelő ütemezéssel előállítsa a műveletvégző egység számára szükséges vezérlő jeleket. A 4-79. ábra az iteratív akkumulátor áramkör kapcsolási sémáját mutatja be, kihangsúlyozva, hogy ebben a példaáramkörben mely erőforrások alkotják a vezérlő egységet (piros), és melyek a műveletvégző egységet (zöld).



4-79. ábra. *Vezérlő és műveletvégző egység az iteratív akkumulátorban*

Az iteratív akkumulátor esetén a vezérlő egység szerepét egy egyszerű felfelé számláló áramkör tölti be. Ezt a megoldást számlálós vezérlésnek nevezzük. Ezt a kevéssé rugalmas vezérlési struktúrát csak olyan esetekben alkalmazzuk, amikor a vezérlőjelek viselkedését meghatározó algoritmus viszonylag egyszerű, nem tartalmaz elágazásokat, ciklusokat, amelyeket a műveletvégző egységben feldolgozott adatok befolyásolnának, tehát nincsenek visszacsatolások, ún. állapotjelek a műveletvégző egység és a vezérlő egység között.

Vizsgáljunk meg egy kicsit összetettebb példát is a számlálós vezérlés megvalósítására. A 4-80. ábrán látható pszeudó-kód egy előjel nélküli egész számokon végzett, iteratív osztás algoritmust, az ún. *shift&subtract* algoritmust írja le, ami tulajdonképpen nem más, mint a jól ismert, decimális számok osztására alkalmas *long division* algoritmus bináris megfelelője.

Az itt bemutatott algoritmus a `w` biten ábrázolt `dividend` és `divisor` számok hányadosát (`quotient`), valamint az osztás maradékát (`remainder`) számítja ki iteratív úton. A 4-81.

Pseudo

```
remainder = 0
quotient = 0

for i in w-1 downto 0 loop
  remainder = remainder << 1
  remainder[0] = dividend[i]
  if ( remainder >= divisor ) then
    remainder = remainder - divisor
    quotient[i] = 1
  end if
end loop

return quotient, remainder
```

4-80. ábra. *Előjel nélküli egész osztás* shift&subtract *algoritmussal*

ábrán az algoritmust megvalósító, számlálós vezérléssel ellátott áramkör VHDL és Verilog modellje látható.

Figyeljük meg, hogy bár ez az áramkör nyilvánvalóan bonyolultabb, mint az iteratív akkumulátor, itt is igaz, hogy a vezérlő jelek viselkedését meghatározó algoritmus tulajdonképpen nagyon egyszerű; az `iterator` számláló értékét – ami egyúttal egy közvetlen vezérlő jel is a műveletvégző egység számára – minden órajelciklusban eggyel csökkentjük, amíg el nem érjük a nullát, ami a számítás végét jelenti.

Összetettebb viselkedésű rendszerek esetén a számlálós vezérlésnél rugalmasabb megoldásra van szükség. Tegyük fel, hogy a műveletvégzést vezérlő áramkör nemcsak felfelé és lefelé számlálást tud megvalósítani, hanem a saját aktuális állapotát, a külső vezérlő jeleket, sőt a műveletvégző egységből visszacsatolt, ún. állapotjeleket akár minden órajelciklusban ki tudja értékelni, ezek alapján meg tudja állapítani, hogy mi legyen a vezérlő jelek soron következő értéke, és mi legyen magának a vezérlő egységnek az állapotát leíró regiszternek a következő értéke. Ez a viselkedés már ismerős lehet a 4.5.5. pontból, hiszen pontosan ezek azok a képességek, amelyek egy állapotgépre jellemzők. Az állapotgépes vezérlést a *shift&subtract* osztó áramkörnek egy továbbfejlesztett verzióján mutatjuk be. A 4-81. ábrán látható megoldáshoz képest a következő funkciókkal bővítjük ki az áramkört:

- Előjel nélküli egész számok helyett ezúttal előjeles, kettes komplemens kódolású számokkal dolgozunk.
- A számítás megkezdése előtt megvizsgáljuk, hogy az osztó nulla-e. Ha igen, akkor hibajelet generálunk.
- Hand-shake interfészt valósítunk meg a felhasználó felé.

A 4-82. ábra azt mutatja be, hogy a nullával való osztás detektálásának és a negatív számok kezelésének igénye hogyan változtatja meg a leendő áramkörünk által megvalósított algoritmust.

A két pszeudó-kódot összevetve látható, hogy az előjel nélküli osztás algoritmust változatlan formában felhasználtuk, de annak operandusait egy előfeldolgozás lépésben állítjuk elő (`divisor_abs`, `dividend_abs`), majd eredményét (`remainder_core`, `quotient_core`) csak egy utófeldolgozás után juttatjuk a kimenetre (`remainder`, `quotient`). Így a korábban elkészített, csak előjel nélküli egész számok osztására képes áramkörünk műveletvégző egységét fel tudjuk használni. A műveletvégzést tehát három fázisra osztjuk:

VHDL

```
architecture rtl of divisor is
  signal iterator: integer range 0 to w-1;
begin

  L_CONTROL: process ( clk, rst )
  begin
    if ( rst = '1' ) then iterator <= w-1; finish <= '0';
    elsif ( rising_edge(clk) ) then
      if ( enable = '1' ) then
        if ( iterator > 0 ) then iterator <= iterator - 1;
        end if;
      else iterator <= w-1; end if;

      if ( iterator = 0 ) then finish <= '1';
      else finish <= '0'; end if;
    end if;
  end process;

  L_DATAPATH: process ( clk, rst )
    variable tmp: unsigned (w-1 downto 0);
  begin
    if ( rst = '1' ) then
      remainder <= (others => '0');
      quotient <= (others => '0');
    elsif ( rising_edge(clk) ) then
      if ( enable = '0' ) then
        remainder <= (others => '0');
        quotient <= (others => '0');
      else
        if ( finish = '0' ) then
          tmp := shift_left(remainder, 1);
          tmp(0) := dividend(iterator);
          if ( tmp >= divisor ) then
            remainder <= tmp - divisor;
            quotient(iterator) <= '1';
          else remainder <= tmp; end if;
        end if;
      end if;
    end if;
  end process;

end architecture rtl;
```

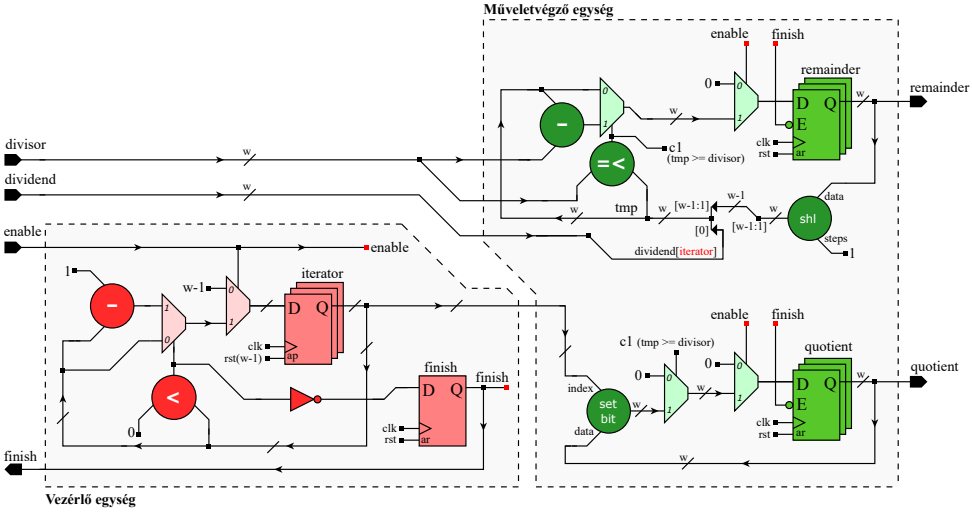
Verilog

```
module divisor #(...)(...);
  logic [w-1:0] iterator;
  logic [w-1:0] tmp;

  // Control Unit
  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) begin
      iterator <= w-1; finish <= 1'b0;
    end else begin
      if ( enable == 1'b1 ) begin
        if ( iterator > 0 )
          iterator <= iterator - 1;
      end
      else iterator <= w-1;
      if ( iterator == 0 ) finish <= 1'b1;
      else finish <= 1'b0;
    end
  end

  // Datapath
  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) begin
      remainder <= 0; quotient <= 0;
    end else begin
      if ( enable == 1'b0 ) begin
        remainder <= 0;
        quotient <= 0;
      end else begin
        if ( finish == 1'b0 ) begin
          tmp = remainder << 1;
          tmp[0] = dividend[iterator];
          if ( tmp >= divisor ) begin
            remainder <= tmp - divisor;
            quotient[iterator] <= 1'b1;
          end else remainder <= tmp;
        end
      end
    end
  end
endmodule
```

Áramköri struktúra



4-81. ábra. *Számlálós vezérléssel megvalósított* shift&subtract *osztó áramkör*

Pseudo

```
remainder_core = 0;
quotient_core = 0;

// pre-processing...
if ( divisor == 0 ) then zero_div = 1; return; end if;
divisor_abs = abs(divisor);
dividend_abs = abs(dividend);

// core division...
for i in w-1 donwto 0 loop
  remainder_core = remainder_core << 1;
  remainder_core[0] = dividend_abs[i];
  if ( remainder_core >= divisor_abs ) then
    remainder_core = remainder_core - divisor_abs;
    quotient_core[i] = 1;
  end if;
end loop;

// post-processing...
if ( dividend[w-1] != divisor[w-1] ) then quotient = -1 * quotient_core; end if;
if ( dividend[w-1] == 1 ) then remainder = -1 * remainder_core; end if;

return quotient, remainder;
```

4-82. ábra. *Előjeles egész osztás* shift&subtract *algoritmussal*

VHDL

```
L_PRE_PROCESSOR: process ( clk, rst )
begin
  if ( rst = '1' ) then
    divisor_abs <= (others => '0');
    dividend_abs <= (others => '0');
  elsif ( rising_edge(clk) ) then
    if ( ena_pre = '1' ) then
      if ( divisor(w-1) = '1' ) then
        divisor_abs <= signed(not divisor) + 1;
      else
        divisor_abs <= unsigned(divisor);
      end if;

      if ( dividend(w-1) = '1' ) then
        dividend_abs <= signed(not dividend) + 1;
      else
        dividend_abs <= signed(dividend);
      end if;
    end if;
  end if;
end process;
```

Verilog

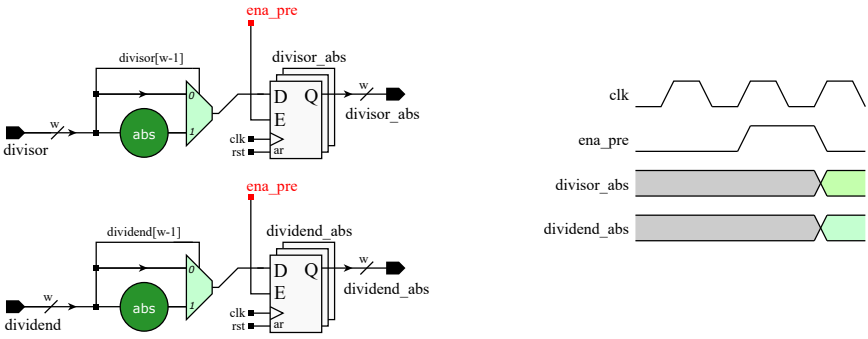
```
// PRE_PROCESSOR
always @ ( posedge clk, posedge rst )
begin
  if ( rst == 1'b1 ) begin
    divisor_abs <= 0;
    dividend_abs <= 0;
  end else begin
    if ( ena_pre == 1'b1 ) begin

      if ( divisor[w-1] == 1'b1 )
        divisor_abs <= ~divisor + 1;
      else divisor_abs <= divisor;

      if ( dividend[w-1] == 1'b1 )
        dividend_abs <= ~dividend + 1;
      else dividend_abs <= dividend;

    end
  end
end
```

Áramköri struktúra



4-83. ábra. *Az előfeldolgozó áramkör*

1. **Előfeldolgozás**: Képezzük az operandusok abszolútértékét, előkészítve azokat az előjel nélküli osztást végző algoritmus számára.
2. **Osztás**: Elvégezzük a korábban bemutatott előjel nélküli osztást, immár az operandusok abszolútértékén.
3. **Utófeldolgozás**: Az eredeti operandusok előjelétől függően utólag megfelelően módosítjuk a hányados és a maradék előjelét.

A fenti részfeladatoknak megfelelően az áramkör műveletvégző egysége is három részből, előfeldolgozóból (`pre-processor`), az előjel nélküli osztást megvalósító magból (`division core`) és egy utófeldolgozóból áll (`post-processor`). E három részáramkört egy-egy

VHDL

```
L_DIVISION_CORE: process ( clk, rst )
  variable tmp: unsigned (w-1 downto 0);
begin
  if ( rst = '1' ) then
    remainder_core <= (others => '0');
    quotient_core <= (others => '0');
  elsif ( rising_edge(clk) ) then
    if ( reset_core = '1' ) then
      remainder_core <= (others => '0');
      quotient_core <= (others => '0');
    else
      if ( ena_core = '1' ) then
        tmp := shift_left(remainder_core, 1);
        tmp(0) := dividend_abs(iterator);
        if ( tmp >= divisor_abs ) then
          remainder_core <= tmp - divisor_abs;
          quotient_core(iterator) <= '1';
        else
          remainder_core <= tmp;
        end if;
      end if;
    end if;
  end if;
end process;
```

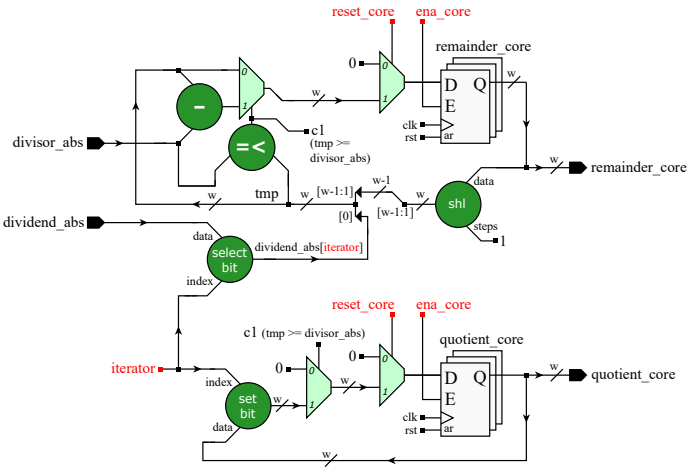
Verilog

```
// DIVISION_CORE
always @ ( posedge clk, posedge rst )
begin
  if ( rst == 1'b1 ) begin
    remainder_core <= 0; quotient_core <= 0;
  end else begin
    if ( reset_core == 1'b1 ) begin
      remainder_core <= 0; quotient_core <= 0;
    end else begin

      if ( ena_core == 1'b1 ) begin
        tmp = remainder_core << 1;
        tmp[0] = dividend_abs[iterator];
        if ( tmp >= divisor_abs ) begin
          remainder_core <= tmp - divisor_abs;
          quotient_core[iterator] <= 1'b1;
        end else begin
          remainder_core <= tmp;
        end
      end

    end
  end
end
```

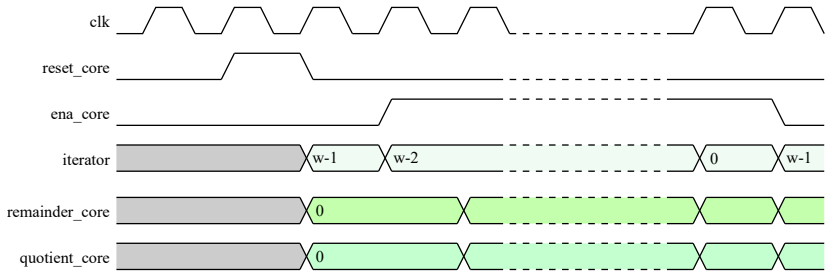
Áramköri struktúra



4-84. ábra. *Az előjel nélküli osztást megvalósító áramkör*

különálló HDL utasítás, egy-egy szinkron folyamat segítségével írjuk le. A 4-83. ábra az előfeldolgozó részáramkört és annak viselkedését szemlélteti a megfelelő vezérlő jelek rendelkezésre állása esetén.

Az előjel nélküli osztó áramkör műveletvégző egységét felhasználjuk, azonban vezérlő jeleinek nevét megváltoztattuk. A 4-84. ábra ezt az előjel nélküli osztást végző részáramkört mutatja be.



4-85. ábra. *Az előjel nélküli osztást végző áramkör működése*

VHDL

```
L_POST_PROCESSOR: process ( clk, rst )
begin
  if ( rst = '1' ) then
    remainder <= (others => '0');
    quotient <= (others => '0');
  elsif ( rising_edge(clk) ) then
    if ( ena_post = '1' ) then
      if ( dividend(w-1) /= divisor(w-1) ) then
        quotient <= signed((not quotient_core) + 1);
      else quotient <= signed(quotient_core);
      end if;

      if ( dividend(w-1) = '1' ) then
        remainder <= signed((not remainder_core) + 1);
      else
        remainder <= signed(remainder_core);
      end if;
    end if;
  end if;
end process;
```

Verilog

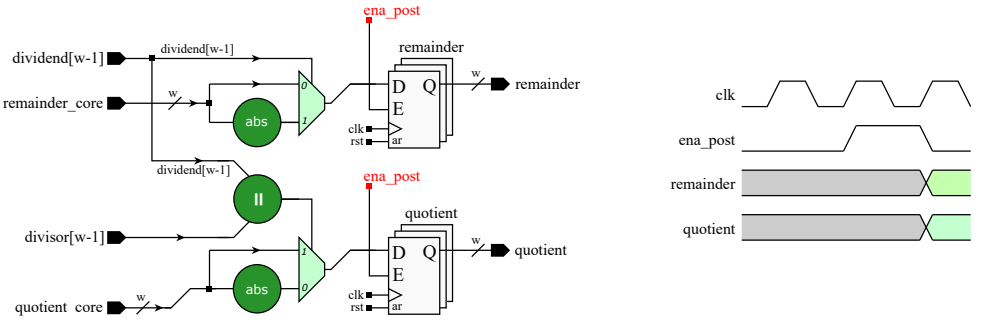
```
// POST_PROCESSOR
always @ ( posedge clk, posedge rst )
begin
  if ( rst == 1'b1 ) begin
    remainder <= 0;
    quotient <= 0;
  end else begin
    if ( ena_post == 1'b1 ) begin

      if ( dividend[w-1] != divisor[w-1] )
        quotient <= ~quotient_core + 1;
      else quotient <= quotient_core;

      if ( dividend[w-1] == 1'b1 )
        remainder <= ~remainder_core + 1;
      else remainder <= remainder_core;

    end
  end
end
```

Áramköri struktúra



4-86. ábra. *Az utófeldolgozó áramkör*

A 4-85. ábra az előjel nélküli osztást végző részáramkör viselkedését szemlélteti a megfelelő vezérlő jelek rendelkezésre állása esetén.

A 4-86. ábra az utófeldolgozó részáramkört és annak viselkedését szemlélteti a megfelelő vezérlő jelek rendelkezésre állása esetén.

A műveletvégző egység természetesen önmagában nem működőképes, hiszen vezérlő jeleit a megfelelő ütemezéssel meg kell hajtani. Ezt a feladatot a példában egy olyan állapotgép fogja ellátni, amelyet egyetlen szinkron folyamat segítségével valósítunk meg (lásd 4-87. ábra).

VHDL

```
L_CONTROL: process ( clk, rst )
begin
  if ( rst = '1' ) then
    state <= s0; rdy <= '1'; zero_div <= '0';
    reset_core <= '0'; ena_pre <= '0';
    ena_core <= '0'; ena_post <= '0'; iterator <= 0;
  elsif ( rising_edge(clk) ) then
    case state is
      when s0     =>  if ( rqst = '1' ) then
                        if ( divisor = 0 ) then
                          zero_div <= '1'; rdy <= '1';
                          state <= hold_rdy;
                        else
                          zero_div <= '0'; rdy <= '0';
                          reset_core <= '1';
                          state <= s1;
                        end if;
                      end if;

      when s1     =>  reset_core <= '0';
                      ena_pre <= '1';
                      iterator <= w-1;
                      state <= s2;

      when s2     =>  ena_pre <= '0';
                      if ( iterator > 0 ) then
                        ena_core <= '1';
                        iterator <= iterator - 1;
                      else
                        ena_core <= '0';
                        iterator <= w-1;
                        ena_post <= '1';
                        state <= s3;
                      end if;

      when s3     =>  ena_post <= '0'; rdy <= '1';
                      state <= hold_rdy;

      when hold_rdy => state <= s0;
      when others   => null;
    end case;
  end if;
end process;
```

Verilog

```
// CONTROL
always @ ( posedge clk, posedge rst )
begin
  if ( rst == 1'b1 ) begin
    state <= s0; rdy <= 1'b1; zero_div <= 1'b0;
    reset_core <= 1'b0; ena_pre <= 1'b0;
    ena_core <= 1'b0; ena_post <= 1'b0;
    iterator <= 0;
  end else
    case ( state )
      s0: if ( rqst == 1'b1 )
            if ( divisor == 0 ) begin
              zero_div <= 1'b1; rdy <= 1'b1;
              state <= hold_rdy;
            end else begin
              zero_div <= 1'b0; rdy <= 1'b0;
              reset_core <= 1'b1; state <= s1;
            end

      s1: begin
            reset_core <= 1'b0; ena_pre <= 1'b1;
            iterator <= w-1; state <= s2;
          end

      s2: begin
            ena_pre <= 1'b0;
            if ( iterator > 0 ) begin
              ena_core <= 1'b1;
              iterator <= iterator - 1;
            end else begin
              ena_core <= 1'b0; iterator <= w-1;
              ena_post <= 1'b1; state <= s3;
            end
          end

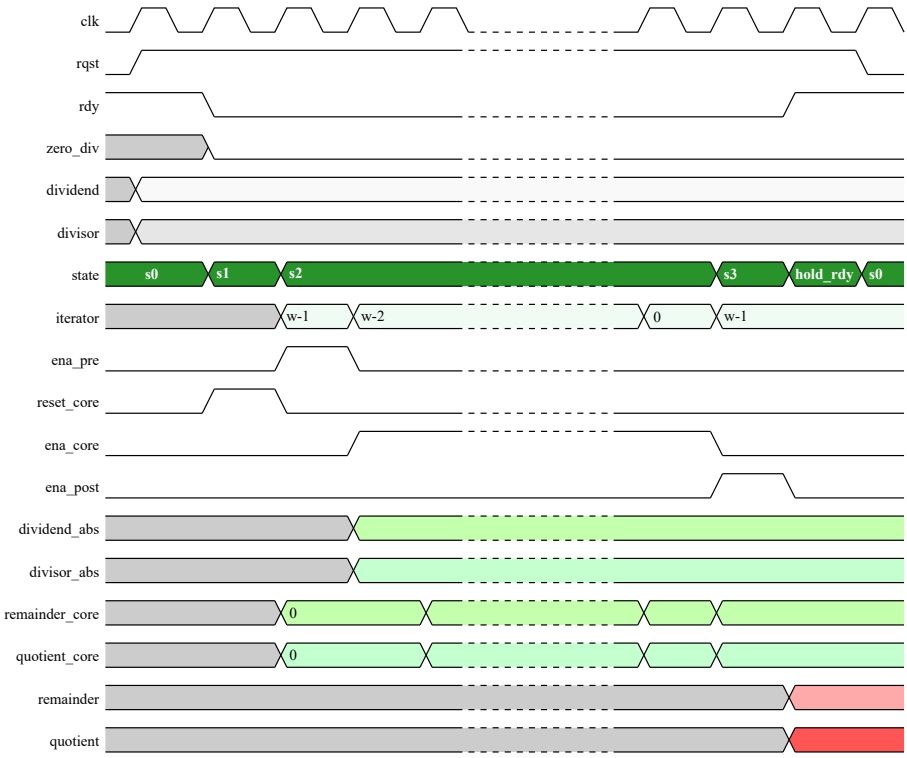
      s3: begin
            ena_post <= 1'b0; rdy <= 1'b1;
            state <= hold_rdy;
          end

      hold_rdy: state <= s0;
      default:  ;
    endcase;
end
```

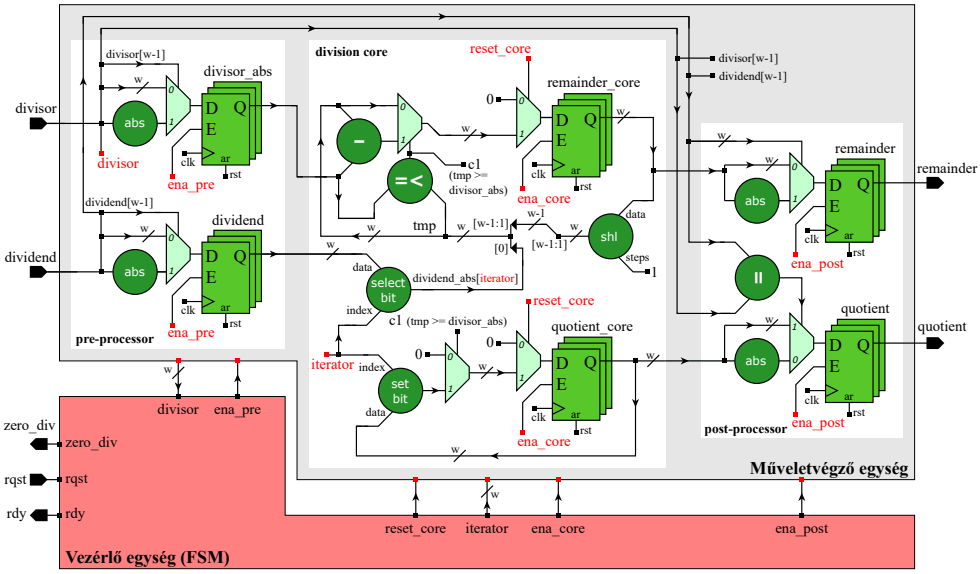
4-87. ábra. *A vezérlő állapotgép*

A 4-88. ábra a vezérlő és műveletvégző egység teljes belső állapotának, valamint az áramkör interfészjeleinek ütemezését mutatja be.

A teljesség kedvéért a 4-89. ábrán megmutatjuk a teljes áramkört egyben. Az ábra nem fejti ki a vezérlő egység szerepét betöltő állapotgép belső szerkezetét, az ehhez hasonló állapotgép-modelleket áramköri struktúra szempontjából a 4.5.5. pontban vizsgáltuk részletesebben.



4-88. ábra. *Az osztó áramkör külső és belső jeleinek időbeli viselkedése*



4-89. ábra. *A teljes osztó áramkör szerkezete*

VHDL

```
architecture rtl of accumulator_fsm is

  type state_t is ( s0, s1, s2, s3, s4 );
  signal state: state_t;

begin

  L_FSM: process ( clk, rst )
  begin
    if ( rst = '1' ) then
      sum <= X"00";
      zero <= '0';
      finish <= '1';
      state <= s0;
    elsif ( rising_edge(clk) ) then
      case state is
        when s0     =>  if ( start = '1' ) then
                          finish <= '0';
                          sum <= a;
                          state <= s1;
                        end if;
        when s1     =>  sum <= sum + b;
                        state <= s2;
        when s2     =>  sum <= sum + c;
                        state <= s3;
        when s3     =>  sum <= sum + d;
                        state <= s4;
        when s4     =>  if ( sum = X"00" ) then
                          zero <= '1';
                        else zero <= '0';
                        end if;
                        finish <= '1';
                        state <= s0;
        when others =>  null;
      end case;
    end if;
  end process;

end architecture rtl;
```

Verilog

```
module accumulator_fsm (...);

  typedef enum { s0, s1, s2, s3, s4 } state_t;
  state_t state;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) begin
      sum <= 8'd0; zero <= 1'b0;
      finish <= 1'b1; state <= s0;
    end else
      case ( state )
        s0: begin
          if ( start == 1'b1 ) begin
            finish <= 1'b0; sum <= a;
            state <= s1;
          end
        end

        s1: begin
          sum <= sum + b; state <= s2;
        end

        s2: begin
          sum <= sum + c; state <= s3;
        end

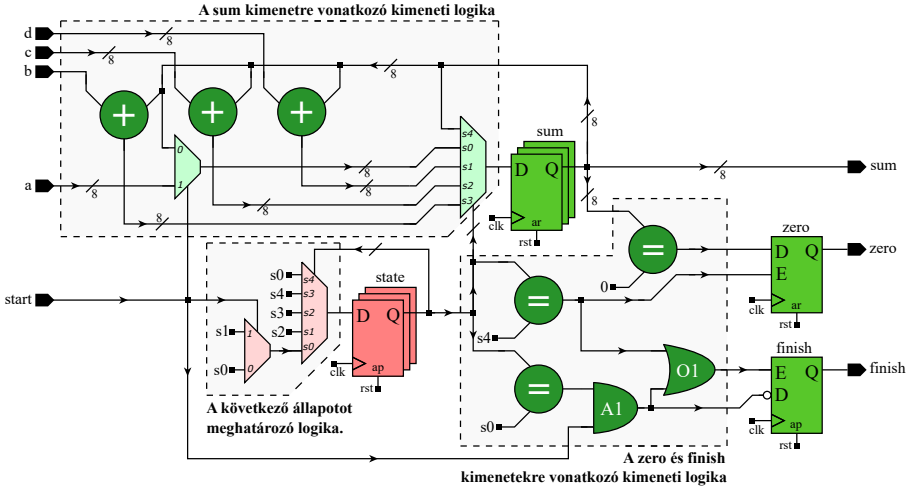
        s3: begin
          sum <= sum + d; state <= s4;
        end

        s4: begin
          if ( sum == 8'd0 ) zero <= 1'b1;
          else zero <= 1'b0;
          finish <= 1'b1; state <= s0;
        end

        default:  ;
      endcase
  end

endmodule
```

Áramköri struktúra

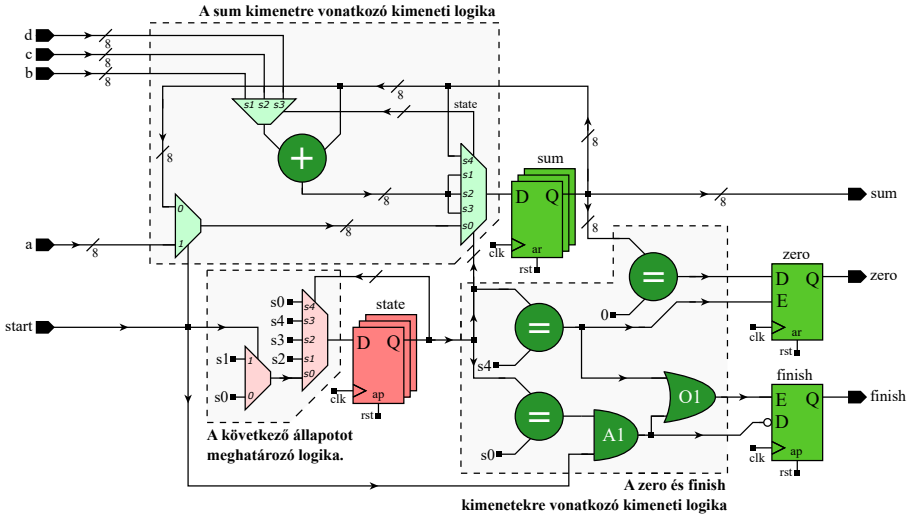


4-90. ábra. *Az iteratív akkumulátor összevont állapotgépes modellje*

Az iteratív akkumulátort és az osztó áramkört bemutató példában is külön HDL kódrészletek írták le a vezérlő és műveletvégző egységet. A 4.5.5. pontban bemutatott, ún. one-shot állapotgép-leírási módszer, amely egyetlen szinkron folyamatba tömörítve fejezi ki az állapotgépek összes szerkezeti elemének viselkedését, egy sajátos megoldást tesz lehetővé az adatfeldolgozó rendszerek modellezésére. Ebben a megoldásban a vezérlő egység szerepét betöltő állapotgép és a műveletvégző egység erőforrásainak működését mind egyetlen szinkron folyamatba sűrítve fejezhetjük ki. A 4-90. ábra az iteratív akkumulátoron mutatja be, miként lehet alkalmazni ezt a technikát.

Ebben a példában a one-shot módszerrel elkészített állapotgép-leírás nemcsak az állapotgép viselkedésére vonatkozó értékadásokat tartalmazza, hanem az állapottól függően más-más aritmetikai műveleteket kifejező értékadásokat is, tehát jellemzően a műveletvégző egységek feladatkörébe tartozó funkciót is megfogalmaz. Ezeknek a műveleteknek az eredményét olyan regiszterek tárolják el, amelyek a one-shot állapotgép kimeneti bufferei nek tekinthetők (`sum`, `zero`, `finish`), az azok bemeneteit meghajtó kombinációs hálózatok pedig tulajdonképpen az állapotgép kimeneti logikáját képviselik. E módszer előnye, hogy a HDL modell tömör, jól átlátható, olvasmányos, tehát könnyen javítható, módosítható, akár igen bonyolult algoritmusokat megvalósító áramkörök RTL modelljét is gyorsan elő tudjuk állítani vele. Ugyanakkor – ahogy már a korábbi példák esetén is láthattuk – ingyen ebéd ezúttal sincs, lássuk, milyen hátrányai vannak az aritmetikai műveletek állapotgép-leírásokba való beágyazásának.

A 4-90. ábra kapcsolási sémájára tekintve jól látható a HDL modellben lévő, három összeadás műveletnek megfelelő három darab összeadó áramkör. A szintézis eszköz HDL leírásunkat feldolgozva láthatóan nem ismerte fel azt a tényt, hogy a három összeadó áramkört soha nem használjuk egyazon időpillanatban, kimeneteik közül bármely órajel-felfutóélnél csak az egyikre van szükség. Ez azt jelenti, hogy három különálló összeadó helyett



4-91. ábra. *Erőforrás-megosztás alkalmazása az állapotgéppel megvalósított, iteratív akkumulátor esetén*

a feladat elvileg egyetlen, multiplexált bemenetű összeadóval is megoldható. A szintézis szoftverek persze igen kifinomultak, egy ilyen apróság nem fog ki rajtuk. A megfelelő beállítás, az ún. *„resource sharing”*, vagyis erőforrás-megosztás funkció bekapcsolása esetén képesek felismerni ezt a szituációt, és a kisebb erőforrásigényű, a 4-91. ábrán látható áramköri struktúrát alakítják ki (lásd még: 8.2.1. pont).

Az állapotgép-leírásba ágyazott aritmetikai műveletekkel kapcsolatban tehát fontos mindig szem előtt tartanunk, hogy az erőforrás-megosztási stratégiát egy ilyen HDL modellel tulajdonképpen teljesen a szintézis szoftverre bízzuk, tehát az eredményként előálló szintetizált netlista igen nagy mértékben függhet a szintézis szoftver beállításaitól. Nem véletlen, hogy az újrafelhasználható IP magok tervezői rendszerint nemcsak a HDL modelleket, hanem a hozzájuk tartozó szimulációs és szintézis parancsfájlokat is rendelkezésünkre bocsátják.

4.9 A MiniCore mikroprocesszor RTL modellje

Ebben az alfejezetben a MiniCore mikroprocesszor mag 3.1. alfejezetben bemutatott rendszerterve és viselkedési szintű modellje alapján elkészítjük annak RTL modelljét, igazodva az adatfeldolgozó rendszerekre jellemző, korábban bemutatott „vezérlő egység + műveletvégző egység” sémához.

Korábban is utaltunk rá, hogy bár az RTL modelleket formálisan a hardverleíró nyelvek segítségével rögzítjük, a tervezés viszont nem rögtön a HDL kód előállításával kezdődik. Így lesz ez a MiniCore processzor esetén is; a HDL modell elkészítése előtt egy sor tervezői döntést kell meghoznunk, amelyek olyan megvalósítási részletekre vonatkoznak, amelyeket sem a rendszerterv, sem a viselkedési modell nem részletez.

Számábrázolás • A MiniCore mikroprocesszor által megvalósított utasításkészlet meglehetősen egyszerű, MISP *(Minimal Instruction Set Processor)* jellegű, aminek hátterében természetesen elsősorban az áll, hogy egy valós, általános célú utasításkészletet megvalósító processzor teljes RTL modelljének bemutatására terjedelmi korlátok miatt aligha lenne lehetőség, és tulajdonképpen felesleges is lenne, hiszen az RTL tervezés jellegzetességei már ezen az egyszerű példán is jól megfigyelhetők lesznek. Az egyszerűség azonban nem jelenti azt, hogy ez a processzor ne lehetne a gyakorlatban is használható. Az utasításkészlet tartalmazza a legalapvetőbb adatmozgató, aritmetikai/logikai és vezérlő utasításokat, tehát elvileg bármilyen algoritmust megvalósíthatunk vele, korlátot tulajdonképpen csak a véges memóriaméretek jelentenek. Az utasításkészlet tartalmazza továbbá az osztás műveletet is, vagyis a processzorunkban lesz egy dedikált, a szoftveres megvalósításnál lényegesen gyorsabb működésű osztó áramkör, ami egy általános processzorra egyáltalán nem jellemző.

Sem a rendszerterv, sem a viselkedési modell nem rögzíti egyértelműen a processzor adatútjának bitszélességét, vagyis azt, hogy a processzor hány bites operandusokkal dolgozzon. RTL tervezőként hozzuk meg azt a döntést, hogy a processzorunk 32 bites operandusokon fog dolgozni, beleértve a szorzás és az osztás műveleteket is. Egy ilyen processzor egy általános célú 8 bites mikroprocesszoros rendszerben használható például nagypontosságú,

aritmetikai társprocesszorként előjel nélküli vagy kettes komplemens kódolású előjeles egész, sőt fixpontos aritmetikai jellegű algoritmusrészletek hatékony végrehajtására.[43]

Utasításkódolás • A viselkedési leírás a gépi utasításoknak egy elvont modelljét használja, a valós áramkörben azonban minden egyes utasítás egy-egy bitsorozat lesz. A viselkedési modellből egyértelműen kiderül, hogy az egyes gépi utasításokban milyen információnak kell benne lennie, de az nem, hogy ezt az információt pontosan hogyan kódoljuk, vagyis hogy az adott utasítást leíró bitsorozat hány bites és melyik bitje mit jelent. A 4-4. táblázat az utasításkészletnek egy olyan definícióját mutatja be, amely már az utasításkódolást is részletesen tartalmazza.

Mnemonik	Gépi utasítás funkciója	Állapotjelzők
loadimm (0x0)	RF[IR[13:11]] = IR[10:0]	Z, N
memread (0x1)	RF[IR[13:11]] = DM[RF[IR[10:8]] + IR[7:0]]	Z, N
memwrite (0x2)	DM[RF[IR[10:8]] + IR[7:0]] = RF[IR[13:11]]	-
add (0x3)	RF[IR[13:11]] = RF[IR[10:8]] + RF[IR[7:5]]	Z, N
sub (0x4)	RF[IR[13:11]] = RF[IR[10:8]] - RF[IR[7:5]]	Z, N
mul (0x5)	RF[IR[13:11]] = (RF[IR[10:8]] * RF[IR[7:5]]) [31:0] RF[IR[13:11]+1] = (RF[IR[10:8]] * RF[IR[7:5]]) [63:32]	-
div (0x6)	RF[IR[13:11]] = RF[IR[10:8]] / RF[IR[7:5]] RF[IR[13:11]+1] = RF[IR[10:8]] mod RF[IR[7:5]]	-
and (0x7)	RF[IR[13:11]] = RF[IR[10:8]] & RF[IR[7:5]]	Z, N
or (0x8)	RF[IR[13:11]] = RF[IR[10:8]] | RF[IR[7:5]]	Z, N
xor (0x9)	RF[IR[13:11]] = RF[IR[10:8]] ^ RF[IR[7:5]]	Z, N
shl (0xA)	RF[IR[13:11]] = RF[IR[10:8]] >> RF[IR[7:5]]	Z, N
shr (0xB)	RF[IR[13:11]] = RF[IR[10:8]] << RF[IR[7:5]]	Z, N
jump (0xC)	IP = IR[9:0]	-
branchzero (0xD)	if (Z == 1) then IP = IR[9:0] else IP = IP + 1	-
branchnegative (0xE)	if (N == 1) then IP = IR[9:0] else IP = IP + 1	-
stop (0xF)	stop the execution	-

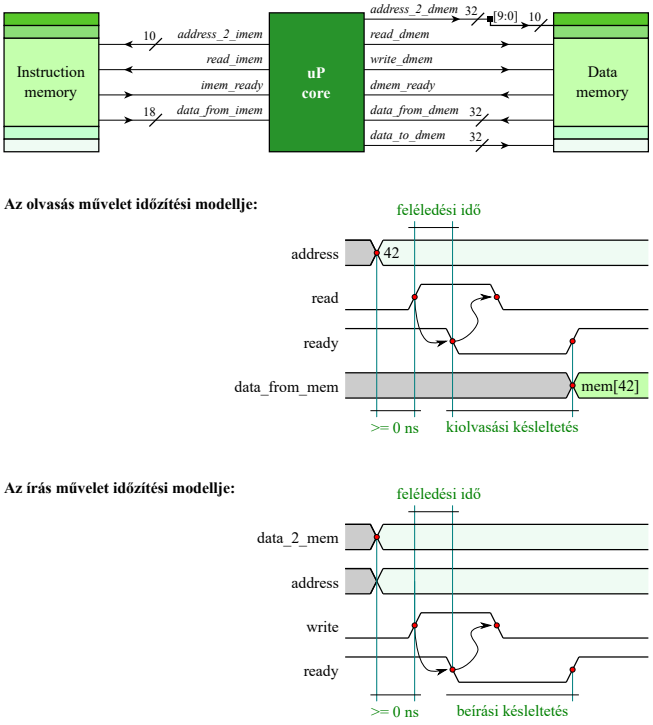
4-4. táblázat. *A mikroprocesszor utasításkészlete és utasításkódolása (vö. 3-1. táblázat)*

E táblázatból kiderül, hogy a tizenhat darab gépi utasítás megkülönböztetésére négy bitet, az utasítások argumentumainak kódolására pedig további tizennégy bitet használunk fel, ami azt jelenti, hogy a gépi utasítások tizennyolc biten ábrázolhatók.

Memória-interfészek • A rendszerterv a külső utasításmemóriával és adatmemóriával kapcsolatban csak annyit rögzít, hogy előbbi 1024 darab utasítást, utóbbi pedig 1024 darab adatbejegyzést tartalmaz. A 32-bites számábrázolás és a 18-bites utasítások ezeknek a memóriáknak a konkrét méretét, valamint adat- és címbuszaik bitszélességeit már meghatározzák, de az RTL modell elkészítése előtt specifikálnunk kell a mikroprocesszor

[43] Az ilyen egyszerű, kis számú, de egyedi gépi utasítást tartalmazó utasításkészletet megvalósító processzorokat is alkalmazás-specifikus processzoroknak (*Application-Specific Instruction-set Processor*, ASIP) tekinthetjük.

mag és a memóriamodulok közötti kommunikációs interfészek jeleit és azok elvárt viselkedését is. Nincs információnk arról, hogy ezeket a memóriamodulokat fizikailag milyen áramkörök fogják majd megvalósítani, ezért egy egyedi, nagyon általános és egyszerű hand-shake interfészt definiálunk mind az utasításmemória, mind az adatmemória felé (lásd 4-92. ábra).



4-92. ábra. *A memóriainterfészek jelei és időzítési modelljei*

Ez az interfészspecifikáció teljesen egyedi, nem felel meg semmilyen memóriainterfész-szabványnak sem, ugyanakkor – mivel magukkal a memóriamodulokkal szemben semmiféle speciális követelményt (például megengedhető maximális késleltetés) nem fogalmaz meg – bármilyen szabványos, soros vagy párhuzamos interfésszel (pl. WISHBONE, AXI, SPI stb.) rendelkező, fizikailag is létező memóriamodul illeszthető hozzá a megfelelő interfészadapter segítségével. Figyeljük meg azt is, hogy bár a rendszerterv szerint az adatmemória mindössze 1024 bejegyzés tárolására alkalmas, magának a processzornak az adatmemóriát címző busza 32-bites, és ennek a 32 bitnek csak az alsó 10 bitjét használjuk az adatmemória címzésére. Ez a megoldás lehetővé teszi az adatmemória bővítését, vagy akár a processzor címtartományára leképezett perifériavezérlő rendszer (*Memory-Mapped I/O*, MMIO) kiépítését.

A MiniCore mikroprocesszor egy adatfeldolgozó rendszer, amelynek RTL modelljét a 4.8. pontban bemutatott példáknál is alkalmazott módszerrel, tehát a műveletvégző és a vezérlő egység különálló megtervezésével készítjük el.

4.9.1 A műveletvégző egység

A műveletvégző egység megtervezésénél a viselkedési modellre támaszkodunk ugyan, de az abban megfogalmazott, a leendő áramkör szerkezetére vonatkozó „célzásoknak” csak egy részét fogjuk valóban megvalósítani. Az egyszerűbb felépítés érdekében nem valósítunk meg utasítás-átlapolást *(pipeline)*, így a viselkedési modellben a dekódolás fázisban expliciten megjelenő pipeline regisztereket (`immediate`, `op_a_index`, `op_b_index`, `destination_index`) az RTL modell – és ez által a leendő áramkör – nem fogja tartalmazni. Így a dekódolás fázis tulajdonképpen a memóriareferens utasítások végrehajtásához szükséges regiszterek, a memória-címregiszter (`mar`) és a memória-adatregiszter (`mdr`) frissítésére egyszerűsödik. A viselkedési modell tárgyalásánál a belső regisztertömb fizikai megvalósítására vonatkozó szempontokat azonban figyelembe vesszük; olyan regisztertömböt használunk, amely két különálló interfészen olvasható, és egy interfészen írható, ami gyakorlatilag megegyezik a 4-52. és 4-53. ábrákon bemutatott, true-dual-port memóriával. Az utasításkódolást leíró táblázat adatai alapján belátható, hogy egy ilyen regisztertömb megfelelő lesz számunkra; ha jobban szemügyre vesszük az utasítások definícióit, láthatjuk, hogy bár a regisztertömb írása két eltérő írási címmel is szerepel (IR[13:11] és ennek inkrementált értéke), tehát elvileg két különálló beíró interfészre lenne szükségünk, de e két beírás műveletet – a cím- és adatbemenetek megfelelő multiplexálásával – elvégezhetjük egymás után is, egyazon beíró interfészen keresztül. Hasonló a helyzet a kiolvasó interfészekkel

VHDL

```vhdl
L_IR_IP: process ( clk, rst )
begin
  if ( rst = '1' ) then
    ir <= (others => '0'); ip <= (others => '0');
  elsif ( rising_edge(clk) ) then

    if ( ce_ir = '1' ) then ir <= data_from_imem; end if;

    if ( ce_ip = '1' ) then
      if ( sel_ip = '0' ) then
        ip <= std_logic_vector(unsigned(ip) + 1);
      else ip <= ir(9 downto 0);
      end if;
    end if;

  end if;
end process;

address_2_imem <= ip;
```

Verilog

```verilog
// IR, IP
always @ ( posedge clk, posedge rst )
begin
  if ( rst == 1'b1 ) begin
    ir <= 18'd0; ip <= 10'd0;
  end else begin

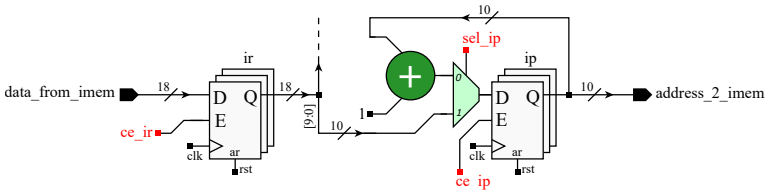
    if ( ce_ir == 1'b1 ) ir <= data_from_imem;

    if ( ce_ip == 1'b1 )
      if ( sel_ip == 1'b0 ) ip <= ip + 1;
      else ip <= ir[9:0];

  end
end

assign address_2_imem = ip;
```

Áramköri struktúra



4-93. ábra. *Az utasításfelhozáshoz szükséges erőforrások*

VHDL

```
mx_rfb <=
  ir(13 downto 11) when sel_rfb = '0' else ir(7 downto 5);

L_RF: process ( clk )
begin
  if ( rising_edge(clk) ) then
    if ( we_rf = '1' ) ...
    rf_douta <=
      rf_content(to_integer(unsigned(ir(10 downto 8))));
    rf_doutb <= rf_content(to_integer(unsigned(mx_rfb)));
  end if;
end process;

L_MAR_MDR: process ( clk, rst )
begin
  if ( rst = '1' ) then
    mar <= (others => '0'); mdr <= (others => '0');
  elsif ( rising_edge(clk) ) then
    mar <= std_logic_vector(unsigned(rf_douta) +
                            unsigned(ir(7 downto 0)));
    mdr <= rf_doutb;
  end if;
end process;

address_2_dmem <= mar;
data_2_dmem <= mdr;
```

Verilog

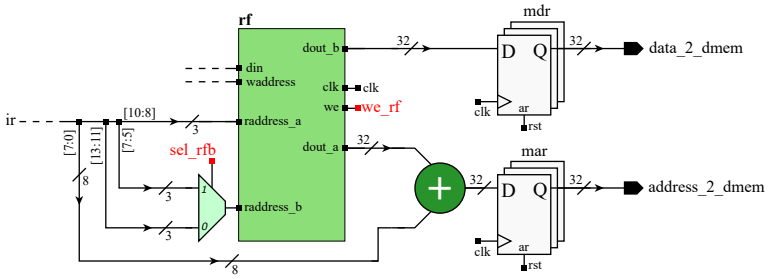
```
assign mx_rfb =
  ( sel_rfb == 1'b0 ) ? ir[13:11] : ir[7:5];

// RF
always @ ( posedge clk )
begin
  if ( we_rf == 1'b1 ) ...
  rf_douta <= rf_content[ir[10:8]];
  rf_doutb <= rf_content[mx_rfb];
end

// MAR, MDR
always @ ( posedge clk, posedge rst )
begin
  if ( rst == 1'b1 ) begin
    mar <= 32'd0; mdr <= 32'd0;
  end else begin
    mar <= rf_douta + ir[7:0];
    mdr <= rf_doutb;
  end
end

assign address_2_dmem = mar;
assign data_2_dmem = mdr;
```

Áramköri struktúra



4-94. ábra. *A dekódolás fázis végrehajtásához szükséges erőforrások és összeköttetéseik*

is; a regisztertömböt az utasításregiszter három különböző mezője címzi ([13:11], [10:8], [7:5]), de az e címek által jelölt bejegyzések közül valójában mindig csak kettőre lesz szükségünk egyszerre; a dekódolás fázisban a `memwrite` utasítás esetén az utasításregiszter [10:8] mezője által megcímzett báziscímre és a [13:11] mezők által címzett, a külső memóriába kiírandó adatbejegyzésre, az aritmetikai műveletek esetén pedig a [10:8] és a [7:5] mezők által címzett operandusokra. E két-két bejegyzés kiolvasása egymás után, rendre a dekódolás és a végrehajtás fázisban történik, tehát a címbemenetek multiplexálása itt is megoldható. Vegyük észre továbbá, hogy az utasításregiszter [10:8] mezője mindkét esetben szerepel, így tulajdonképpen elegendő a dual-port regisztertömb két kiolvasó címbemenete közül az egyiket multiplexálni.

Kezdjük a műveletvégző egység összeállítását az utasításfelhozás megvalósításához szükséges erőforrásokkal. Az utasításfelhozás fázisban a külső utasításmemóriát az utasításmutató (IP) regiszterrel megcímezzük, és végrehajtunk egy olvasás műveletet a 4-92. ábrán látható időzítési modellnek megfelelően. Az ehhez a folyamathoz tartozó vezérlő jelek (`read_imem` és `imem_ready`) kezelése a vezérlést végző állapotgép feladata lesz, a műveletvégző

VHDL

```
mx_rfdin <=
  alu_result(31 downto 0) when sel_rfdin = "00" else
  alu_result(63 downto 32) when sel_rfdin = "01" else
  div_quotient when sel_rfdin = "10" else
  div_remainder;

mx_rfw <= ir(13 downto 11) when sel_rfw = '0' else
          std_logic_vector((unsigned(ir(13 downto 11))+1));

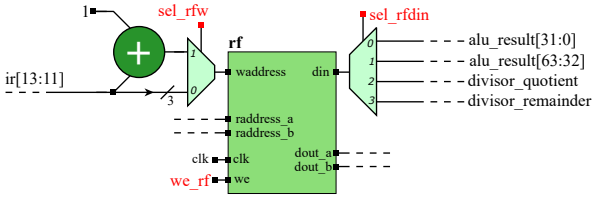
L_RF: process ( clk )
begin
  if ( rising_edge(clk) ) then
    if ( we_rf = '1' ) then
      rf_content(to_integer(unsigned(mx_rfw))) <= mx_rfdin;
    end if;
    rf_douta <= ...
    rf_doutb <= ...
  end if;
end process;
```

Verilog

```
always @ ( sel_rfdin, div_remainder,
           alu_result, div_quotient )
begin
  case ( sel_rfdin )
    2'b00:    mx_rfdin <= alu_result[31:0];
    2'b01:    mx_rfdin <= alu_result[63:32];
    2'b10:    mx_rfdin <= div_quotient;
    default:  mx_rfdin <= div_remainder;
  endcase
end
assign mx_rfw = ( sel_rfw == 1'b0 ) ?
       ir[13:11] : (ir[13:11] + 1);

// RF
always @ ( posedge clk )
begin if ( we_rf == 1'b1 )
        rf_content[mx_rfw] <= mx_rfdin;
      rf_douta <= ...
      rf_doutb <= ...
end
```

Áramköri struktúra



4-95. ábra. *A végrehajtás fázisban előálló eredmények regisztertömbbe való visszaírásához szükséges összeköttetések*

egységbe csak a címet tároló utasításmutatót és a kiolvasott utasítást tároló utasításregisztert építjük be (lásd 4-93. ábra).

Az utasításregiszterbe minden esetben az utasításmemória adatkimenetének értékét töltjük, az utasításmutató bemenetét azonban multiplexálnunk kell, hiszen annak következő értéke a vezérlésátadó utasítások esetén az utasításregiszter alsó 10 bitje (IR[9:0], az ugrási cím), a többi utasítás esetén viszont magának az utasításmutatónak az inkrementált értéke (IP + 1).

Az utasításregiszter betöltése után a dekódolás fázis következik, amely során a memóriareferens utasításokhoz szükséges regiszterek, a memória-címregiszter (`mar`) és a memória-adatregiszter (`mdr`) frissítésére kerül sor. Az eltolt regiszter-indirekt címzés miatt a memória-címregiszter értékének előállításához egy, a regisztertömbnek az utasításregiszter [10:8] mezője által címzett bejegyzésére, valamint az utasításregiszter [7:0] mezőjében eltárolt eltolás mértékére is szükségünk van. A teljes memóriacím e két érték összegeként áll elő. A memória-adatregiszterbe a regisztertömbnek az utasításregiszter [13:11] mezője által megcímzett bejegyzésének értékét írjuk. A dekódolás fázis végrehajtásához szükséges erőforrásokat és azok összeköttetéseit a 4-94. ábra szemlélteti.

A dekódolás során csak két regisztertömbbeli bejegyzést olvasunk ki; egyiket az utasításregiszter [13:11], a másikat a [10:8] mezői által kódolt címről. Ennek végrehajtásához elegendő lenne a regisztertömb két kiolvasó interfészének címbemeneteit közvetlenül meghajtani, de mivel az aritmetikai utasítások végrehajtása során egy harmadik, az

VHDL

```
mx_alu <= data_from_dmem when sel_alu = "00" else
(X"00000" & "0" & ir(10 downto 0)) when sel_alu = "01" else
rf_doutb;

L_ALU: process ( ir(17 downto 14), rf_douta, mx_alu )
  variable tmp32: signed (31 downto 0);
  variable tmp64: signed (63 downto 0);
begin

  tmp64 := signed(rf_douta) * signed(mx_alu);

  case ir(17 downto 14) is
    when i_add => tmp32 := signed(rf_douta) + signed(mx_alu);
    when i_sub => tmp32 := signed(rf_douta) - signed(mx_alu);
    when i_and => tmp32 := signed(rf_douta and mx_alu);
    when i_or  => tmp32 := signed(rf_douta or mx_alu);
    when i_xor => tmp32 := signed(rf_douta xor mx_alu);
    when i_shl => tmp32 := shift_left(
                             signed(rf_douta),
                             to_integer(unsigned(mx_alu))
                           );
    when i_shr => tmp32 := shift_right(
                             signed(rf_douta),
                             to_integer(unsigned(mx_alu))
                           );
    when others => tmp32 := signed(mx_alu);
  end case;

  if ( ir(17 downto 14) = i_mul ) then
    alu_result <= std_logic_vector(tmp64);
  else alu_result <= (X"00000000" & std_logic_vector(tmp32));
  end if;

  if ( tmp32 = 0 ) then alu_z <= '1'; else alu_z <= '0'; end if;
  alu_n <= tmp32(31);

end process;
```

Verilog

```
always @ ( sel_alu, rf_doutb,
           data_from_dmem, ir[10:0] )
begin
  case ( sel_alu )
    2'b00: mx_alu <= data_from_dmem;
    2'b01: mx_alu <= ir[10:0];
    default: mx_alu <= rf_doutb;
  endcase
end

// ALU
always @ ( ir[17:14], rf_douta,
           mx_alu )
begin

  tmp64 = $signed(rf_douta) *
          $signed(mx_alu);

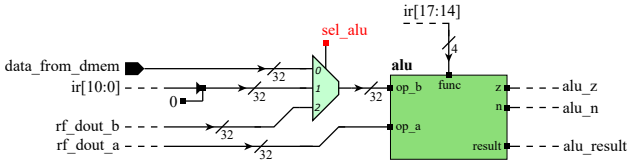
  case ( ir[17:14] )
    i_add: tmp32 = rf_douta + mx_alu;
    i_sub: tmp32 = rf_douta - mx_alu;
    i_and: tmp32 = rf_douta & mx_alu;
    i_or:  tmp32 = rf_douta | mx_alu;
    i_xor: tmp32 = rf_douta ^ mx_alu;
    i_shl: tmp32 = rf_douta << mx_alu;
    i_shr: tmp32 = rf_douta >> mx_alu;
    default: tmp32 = mx_alu;
  endcase

  if ( ir[17:14] == i_mul )
    alu_result <= tmp64;
  else alu_result <= tmp32;

  if ( tmp32 == 0 ) alu_z <= 1'b1;
  else alu_z <= 1'b0;
  alu_n <= tmp32[31];

end
```

Áramköri struktúra



4-96. ábra. *Az aritmetikai/logikai egység megvalósítása*

utasításregiszter [7:5] mezője által kódolt címen lévő bejegyzésre is szükségünk van, ezért a regisztertömb egyik olvasási címbemenetét kénytelenek vagyunk multiplexálni.

A regisztertömb beíró interfészét az aritmetikai/logikai utasítások (`add`, `sub`, `mul`, `div`, `and`, `or`, `xor`, `shl`, `shr`), valamint a `loadimm` és a `memread` utasítások végrehajtási fázisában használjuk. Ezeknek az utasításoknak az eredményeit két különálló műveletvégző elem, az aritmetikai/logikai egység (*Arithmetic/Logic Unit*, ALU) és – az osztás utasítás esetén – a 4.8. alfejezetben ismertetett osztó (`divisor`) áramkör szolgáltatja (lásd 4-95. ábra).

Mindkét aritmetikai egységre jellemző, hogy 2 × 32 bit eredményt állít elő. Az ALU 64-bites kimenetének felső 32 bitje az utasítások többsége esetén nem releváns, azok csak a szorzás utasítás esetén íródnak vissza a regisztertömbbe. A regisztertömb bejegyzései

32 bitesek, ami azt jelenti, hogy a 64 bites eredmény visszaírásához két bejegyzés felülírására van szükség. Az utasításregiszter [13:11] mezője jelöli ki a 64 bites szorzat alsó 32 bitjének regisztertömbbeli helyét, a felső 32 bit visszaírása az eggyel nagyobb indexre történik a beíró interfész címbemenetének multiplexálásával, egy órajelciklussal később. Az osztó áramkör szintén két darab 32 bites eredményt, az operandusok egész osztásával előálló hányadost (`quotient`) és maradékot (`remainder`) állítja elő. Az utasításregiszter [13:11] mezője ebben az esetben a hányados helyét jelöli ki, az eggyel nagyobb indexű rekeszbe a maradékot írjuk vissza.

A 4-96. ábra az aritmetikai/logikai egység megvalósítását mutatja be.

A `loadimm` és a `memread` utasítások ugyan nem igényelnek semmilyen aritmetikai/logikai transzformációt, de mivel a Z és N állapotjelzőket ezek esetén is elő kell állítani, ezért ezeket az utasításokat is az ALU hajtja végre. A `loadimm` esetén az utasításregiszter [10:0] mezőjének 32 bitre kiegészített értékét, a `memread` esetén pedig a külső memóriából kiolvasott, 32-bites adat értékét kapcsoljuk az ALU `op_b` bemenetére, amit aztán változtatás nélkül a Z és N kimeneteket előállító dekóderekre juttatunk. Az aritmetikai/logikai egység egyszerű kombinációs hálózatként van megvalósítva, ezért a Z és N kimeneteinek értékét egy-egy különálló flip-flopban eltároljuk a végrehajtás fázisban (lásd 4-97. ábra).

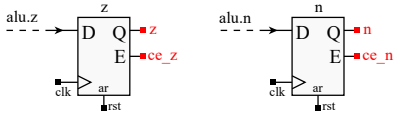
VHDL

```
L_Z_N: process ( clk, rst )
begin
  if ( rst = '1' ) then z <= '0'; n <= '0';
  elsif ( rising_edge(clk) ) then
    if ( ce_z = '1' ) then z <= alu_z; end if;
    if ( ce_n = '1' ) then n <= alu_n; end if;
  end if;
end process;
```

Verilog

```
// Z, N
always @ ( posedge clk, posedge rst )
begin
  if ( rst == 1'b1 ) begin z <= 1'b0; n <= 1'b0;
  end else begin
    if ( ce_z == 1'b1 ) z <= alu_z;
    if ( ce_n == 1'b1 ) n <= alu_n;
  end
 end
```

Áramköri struktúra



4-97. ábra. *Az ALU Z és N kimeneteit egy-egy flip-flopban eltároljuk*

Az osztó áramkör belső felépítését és működését a 4.8. pontban részleteztük, ezt a mikroprocesszor műveletvégző egységében 32 bites operandusmérettel egyszerűen példányosítjuk (lásd 4-98. ábra).

Összeállt a mikroprocesszor teljes műveletvégző egysége (lásd 4-99. ábra), ami azonban önmagában még nem képes semmiféle utasássorozat végrehajtására. A következő feladat annak a vezérlőáramkörnek a megtervezése, amely gondoskodik a műveletvégző egység vezérlő jeleinek megfelelő ütemezésű meghajtásáról, ezáltal valósítva meg a mikroprocesszorra jellemző, az utasításfelhozás, dekódolás és végrehajtás fázisokat ismétlő, ciklikus viselkedést.

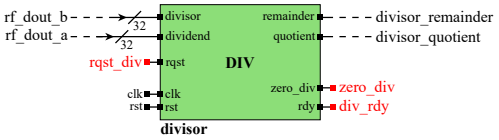
VHDL

```
L_DIVISOR: entity work.divisor(rtl) generic map (32)
           port map (
             clk       => clk,
             rst       => rst,
             rqst      => rqst_div,
             rdy       => div_rdy,
             dividend  => rf_douta,
             divisor   => rf_doutb,
             zero_div  => zero_div,
             quotient  => div_quotient,
             remainder => div_remainder
           );
```

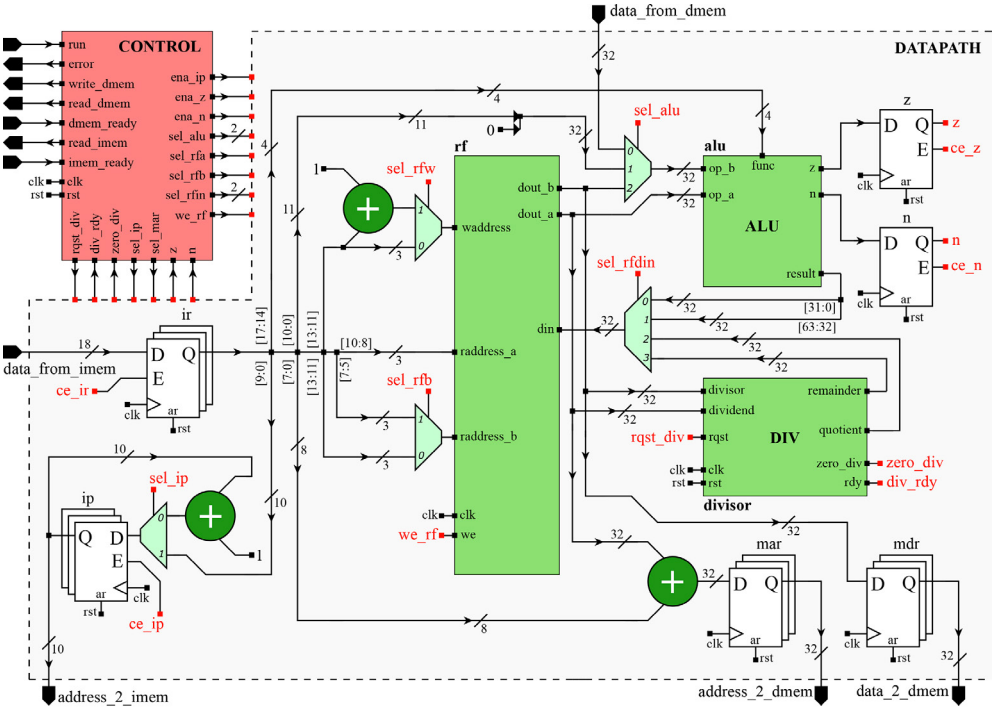
Verilog

```
// DIVISOR
divisor #(.w(32)) L_DIVISOR (
  .clk(clk),
  .rst(rst),
  .rqst(rqst_div),
  .rdy(div_rdy),
  .dividend(rf_douta),
  .divisor(rf_doutb),
  .zero_div(zero_div),
  .quotient(div_quotient),
  .remainder(div_remainder)
);
```

Áramköri struktúra



4-98. ábra. *A 4.8. pontban bemutatott osztó áramkör példányosítása a processzor műveletvégző egységében*



4-99. ábra. *A MiniCore mikroprocesszor teljes RTL kapcsolási sémája*

4.9.2 A vezérlő egység

A MiniCore mikroprocesszor vezérlő egységének szerepét egy one-shot módszerrel leírt állapotgép tölti be, amelynek vázát a 4-100. ábra szemlélteti.

VHDL

```vhdl
L_CONTROL: process ( clk, rst )
begin
  if ( rst = '1' ) then
    state <= fetch_1; run <= '0'; error <= '0';
    read_imem <= '0'; write_dmem <= '0';
    read_dmem <= '0'; ce_ir <= '0'; sel_ip <= '0';
    ce_ip <= '0'; sel_rfdin <= "00"; sel_rfw <= '0';
    sel_rfb <= '0'; we_rf <= '0'; sel_alu <= "00";
    ce_z <= '0'; ce_n <= '0'; rqst_div <= '0';
  elsif ( rising_edge(clk) ) then
    case state is
      when fetch_1  =>  ...
      when fetch_2  =>  ...

      when decode_1 =>  ...
      when decode_2 =>  ...

      when execute  =>  ...

      when s_memread  =>  ...

      when s_memwrite =>  ...

      when s_alu  =>  ...

      when s_mul_1  =>  ...
      when s_mul_2  =>  ...

      when s_div_1  =>  ...
      when s_div_2  =>  ...
      when s_div_3  =>  ...
      when s_div_4  =>  ...

      when deassert =>  ...

      when others   => null;
    end case;
  end if;
end process;
```

Verilog

```verilog
// CONTROL
always @ ( posedge clk, posedge rst )
begin
  if ( rst == 1'b1 ) begin
    state <= fetch_1; run <= 1'b0; error <= 1'b0;
    read_imem <= 1'b0; write_dmem <= 1'b0;
    read_dmem <= 1'b0; ce_ir <= 1'b0; sel_ip <= 1'b0;
    ce_ip <= 1'b0; sel_rfdin <= 2'b00;
    sel_rfw <= 1'b0; sel_rfb <= 1'b0; we_rf <= 1'b0;
    sel_alu <= 2'b00; ce_z <= 1'b0; ce_n <= 1'b0;
    rqst_div <= 1'b0;
  end else begin
    case ( state )
      fetch_1:  ...
      fetch_2:  ...

      decode_1: ...
      decode_2: ...

      execute:  ...

      s_memread:  ...

      s_memwrite: ...

      s_alu:  ...

      s_mul_1:  ...
      s_mul_2:  ...

      s_div_1:  ...
      s_div_2:  ...
      s_div_3:  ...
      s_div_4:  ...

      deassert: ...

      default:  ;
    endcase
  end
end
```

4-100. ábra. *A vezérlő állapotgép HDL modelljének váza*

E kódrészletek egyelőre csak az állapotgép reset-funkcióját tartalmazzák, amelyben magát az állapotgépet és az általa meghajtott összes külső és belső vezérlő jelet alapállapotba állítjuk. Az aszinkron reset elengedése után az állapotgép megkezdi az első utasítás felhozását az utasításmemóriából. Mivel az utasításmutató regiszter (`ip`) reset hatására nulla értéket vesz fel (lásd 4-93. ábra), ezért a programvégrehajtás erről a címről kezdődik. A `fetch_1` állapotban az állapotgép mindaddig várakozik, ameddig a külső `start` jel magas szintű nem lesz. Ekkor a `run` állapotjelzőt magas szintre állítja, jelezve a külvilág felé, hogy a program futtatása megkezdődött, és elindít egy olvasási ciklust az utasításmemória felé, az utasításmutató aktuális értékét felhasználva olvasási címként (lásd 4-101. ábra).

Az olvasási ciklus után, ahogy az `imem_ready` jel magas szintre vált, az állapotgép az utasításregiszter értékét a `ce_ir` vezérlő jelen előállított, egyetlen órajelperiódusig fennálló pozitív impulzussal frissíti, miközben átlép a dekódolás állapotba. Ahogy korábban említettük,

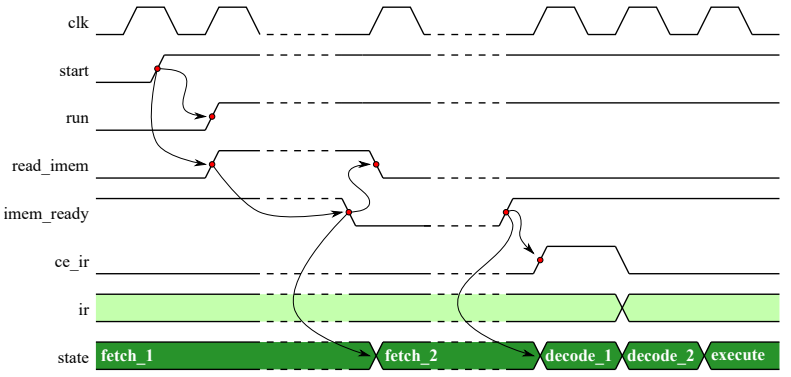
VHDL

```
when fetch_1  => if ( start = '1' ) then
                   run <= '1'; read_imem <= '1';
                   if ( imem_ready = '0' ) then
                     read_imem <= '0';
                     state <= fetch_2;
                   end if;
                 else read_imem <= '0';
                 end if;
when fetch_2  => if ( imem_ready = '1' ) then
                   ce_ir <= '1'; state <= decode_1;
                 end if;
                 sel_rfb <= '0';
-------------------------------------------------------
when decode_1 => ce_ir <= '0'; state <= decode_2;
when decode_2 => state <= execute;
```

Verilog

```
fetch_1: if ( start == 1'b1 ) begin
           run <= 1'b1; read_imem <= 1'b1;
           if ( imem_ready == 1'b0 ) begin
             read_imem <= 1'b0; state <= fetch_2;
           end
         end else read_imem <= 1'b0;
fetch_2: begin
           if ( imem_ready == 1'b1 ) begin
             ce_ir <= 1'b1; state <= decode_1;
           end
           sel_rfb <= 1'b0;
         end
//-----------------------------------------------------
decode_1: begin ce_ir <= 1'b0; state <= decode_2; end
decode_2: state <= execute;
```

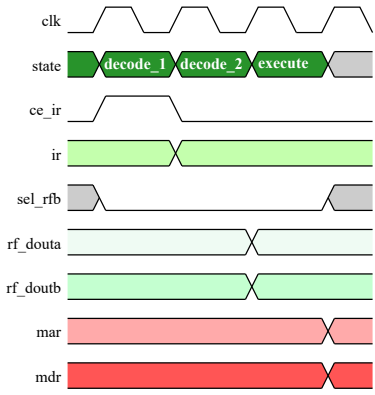
A belső állapot változása



4-101. ábra. *Az utasításfelhozás és dekódolás fázisokhoz tartozó állapotok*

a viselkedési modellben expliciten benne lévő `immediate`, `op_a_index`, `op_b_index` és `destination_index` regiszterek megspórolásával – amit a nem-átlapolt utasítás-végrehajtás tett lehetővé – a dekódolás fázis tulajdonképpen a memóriareferens utasításokhoz kapcsolódó `mar` és `mdr` regiszterek frissítésére egyszerűsödött. Mivel sem a `mar`, sem az `mdr` regiszternek nincs órajel-engedélyező bemenete, ezért a vezérlő állapotgép szempontjából a dekódolás lényegében annyit jelent, hogy a regisztertömb `raddress_b` bemenetére a `sel_rfb` vezérlő jel segítségével rákapcsoljuk az utasításregiszter [13:11] mezőjét, ami az `mdr` regiszterbe beírandó bejegyzést címzi (`fetch_2` állapot), majd várunk két órajelciklust; az elsőben a regisztertömb kimenetei, a másodikban pedig azok alapján a `mar` és `mdr` regiszterek frissülnek (lásd 4-102. ábra).

Az utasítások felhozása és dekódolása után következhet a végrehajtás fázisa, ami abban különbözik az előző két fázistól, hogy itt az előállított vezérlő jelek és azok ütemezése függ attól, hogy éppen milyen utasítást hajtunk végre. Ezért a végrehajtás fázisa az utasításkód vizsgálatával kezdődik, és ennek a vizsgálatnak az eredménye alapján – vagyis az alapján, hogy milyen utasításról van szó – a vezérlés elágazik. Az elágazást az állapotgép HDL modelljében egy `case`-szerkezettel valósítjuk meg (lásd 4-103. ábra).



4-102. ábra *– A belső állapot változása a dekódolás során*

VHDL

```
when execute => case ir(17 downto 14) is
                  when i_loadimm  => ...
                  when i_memread  => ...
                  when i_memwrite => ...
                  when i_add | i_sub |
                       i_and | i_or  | i_xor |
                       i_shl | i_shr => ...
                  when i_mul => ...
                  when i_div => ...
                  when i_jump =>  ...
                  when i_branchzero => ...
                  when i_branchnegative => ...
                  when i_stop => ...
                  when others => null;
                end case;
```

Verilog

```
execute: case ( ir[17:14] )
           i_loadimm: ...
           i_memread: ...
           i_memwrite: ...
           i_add, i_sub,
           i_and, i_or, i_xor,
           i_shl, i_shr: ...
           i_mul: ...
           i_div: ...
           i_jump: ...
           i_branchzero: ...
           i_branchnegative: ...
           i_stop: ...
           default: ;
         endcase
```

4-103. ábra. *A végrehajtáshoz tartozó első állapot; az utasítástípus megállapítása az utasításregiszter [17:14] mezője alapján*

Az egyes gépi utasítások végrehajtása több órajelciklust igényelhet, ami segédállapotok felvételét teszi szükségessé (lásd 4-100. ábra: `s_memread`, `s_memwrite`, `s_alu`, `s_mul_1/2`, `s_div_1/2/3/4`, `deassert`, `s_error`). A felhozott utasítás kódja határozza meg, hogy az állapotgép az `execute` állapot után melyik segédállapotban folytatja a végrehajtást.

A `loadimm` utasítás végrehajtása során az utasításregiszter [10:0] mezőjében tárolt konstanst a magasabb helyiértékű bitekbe nullát írva 32 bitre kiegészítjük, majd az így kapott értéket visszaírjuk a regisztertömbnek az utasításregiszter [13:11] mezője által címzett rekeszébe. Az utasítás specifikációja alapján a 32 bitre kiegészített értékről meg kell állapítanunk, hogy nulla vagy negatív-e, és ez alapján a Z és N állapotjelzőket is frissítenünk kell. Az e két állapotjelző értékének meghatározását végző logika az ALU-ba van beépítve, ezért – bár a `loadimm` utasítás esetén nincs szükség aritmetikai/logikai műveletek elvégzésére – a 32 bitre kiegészített konstans regisztertömbbe való visszaírása az ALU-n keresztül valósul meg. A vezérlő állapotgépnek tehát a `sel_alu` vezérlő jel segítségével rá kell kapcsolnia ezt a 32 bites konstanst az ALU `op_b` bemenetére. Az ALU `func` bemenetére az utasításkód (`ir[17:14]`) statikusan rá van kapcsolva, így az ALU képes megállapítani, hogy épp a `loadimm` utasítás végrehajtása van folyamatban, tehát a feladata annyi, hogy az

VHDL

```
when execute => case ir(17 downto 14) is
                  when  i_loadimm  =>
                    sel_alu <= "01";
                    sel_rfdin <= "00";
                    sel_rfw <= '0'; sel_ip <= '0';
                    ce_z <= '1'; ce_n <= '1';
                    we_rf <= '1'; ce_ip <= '1';
                    state <= deassert;
                  ...
                  end case;
...
when deassert => ce_z <= '0'; ce_n <= '0';
                 we_rf <= '0'; ce_ip <= '0';
                 state <= fetch_1;
```

Verilog

```
execute: case ( ir[17:14] )
           i_loadimm: begin
             sel_alu <= 2'b01; sel_rfdin <= 2'b00;
             sel_rfw <= 1'b0; sel_ip <= 1'b0;
             ce_z <= 1'b1; ce_n <= 1'b1;
             we_rf <= 1'b1; ce_ip <= 1'b1;
             state <= deassert;
           end
           ...
         endcase
...
deassert: begin
            ce_z <= 1'b0; ce_n <= 1'b0;
            we_rf <= 1'b0; ce_ip <= 1'b0;
            state <= fetch_1;
          end
```

4-104. ábra. *A `loadimm` utasítás végrehajtása*

`op_b` bemenetet változtatás nélkül a kimenetére kapcsolja, és meghatározza a Z és N állapotjelzők értékét.

Az ALU `op_b` bemenetére kapcsolandó operandus kiválasztásán felül a vezérlő állapotgépnek a regisztertömb visszaírási címének (`waddress`) és a visszaírandó adat forrásának (`din`) kijelölését is el kell végeznie rendre a `sel_rfw` és `sel_rfdin` vezérlő jelek megfelelő beállításával. Minden utasítás esetén meg kell határozni továbbá az utasításmutató regiszter következő értékét, ami vagy az aktuális érték eggyel megnövelve, vagy – az ugró utasítások esetén – az utasításregiszter [9:0] mezőjében tárolt konstans. A vezérlő állapotgép e két érték közül a `sel_ip` vezérlő jel megfelelő beállításával választhat. Ezzel a műveletvégző egység összes multiplexerének kiválasztó bemenetét beállítottuk, az órajel következő felfutó élénél tehát minden multiplexer kimenetén a megfelelő érték lesz jelen. Még ugyanebben a vezérlési állapotban magas szintre kell állítanunk azoknak a regisztereknek az órajel/beírás-engedélyező bemenetét, amelyeket a `loadimm` utasítás során frissíteni szükséges, vagyis a Z és N állapotjelzőkét (`ce_z`, `ce_n`), a regisztertömbét (`we_rf`) és az utasításmutatóét (`ce_ip`). A `loadimm` utasítás végrehajtása a `deassert` állapotban folytatódik, amelyben az összes órajel-engedélyező bemenetet nullára állítjuk, így érve el, hogy ezeken a jeleken mindig csak egy, egyetlen órajel-periódusig fennálló, pozitív impulzus jelenjen meg. A `deassert` állapot az állapotgépet a `fetch_1` állapotba irányítja, amellyel új utasítás-végrehajtási ciklus kezdődik.

A `memread` utasítás hasonló a `loadimm` utasításhoz, tulajdonképpen az egyetlen különbség az, hogy az ALU `op_b` bemenetére ezúttal nem az utasításregiszterből közvetlenül kiolvasott adatot, hanem az adatmemóriából kiolvasott, 32 bites bejegyzést kapcsoljuk, vagyis a `sel_alu` vezérlő jel értékét másképpen állítjuk be. Természetesen ahhoz, hogy a visszaírandó adat a `sel_alu` jellel vezérelt multiplexer kimenetén valóban rendelkezésre álljon, az adatmemória 4-92. ábrán bemutatott időzítési modellje alapján először végre kell hajtani egy olvasás műveletet, amelyhez az olvasási címet a dekódolás fázisában már előkészítettük a `mar` regiszterben (lásd 4-105. ábra).

A memóriaolvasás után a `loadimm` utasításnál látottakhoz hasonlóan ezúttal is a Z és N állapotjelzők, a regisztertömb, valamint az utasításmutató regiszter órajel/beírás-engedélyező jelein állítunk elő egy-egy pozitív impulzust az `s_memread` (felfutó él) és a `deassert` állapotokban (lefutó él).

VHDL

```
when execute => case ir(17 downto 14) is
                  ...
                  when i_memread =>
                    sel_rfw <= '0'; sel_alu <= "00";
                    sel_rfdin <= "00"; sel_ip <= '0';
                    read_dmem <= '1';
                    if ( dmem_ready = '0' ) then
                      read_dmem <= '0';
                      state <= s_memread;
                    end if;
                  ...
                end case;
...
when s_memread => if ( dmem_ready = '1' ) then
                    we_rf <= '1'; ce_z <= '1';
                    ce_n <= '1'; ce_ip <= '1';
                    state <= deassert;
                  end if;
...
when deassert => ce_z <= '0'; ce_n <= '0';
                 we_rf <= '0'; ce_ip <= '0';
                 state <= fetch_1;
```

Verilog

```
execute: case ( ir[17:14] )
           ...
           i_memread: begin
             sel_rfw <= 1'b0; sel_alu <= 2'b00;
             sel_rfdin <= 2'b00; sel_ip <= 1'b0;
             read_dmem <= 1'b1;
             if ( dmem_ready == 1'b0 ) begin
               read_dmem <= 1'b0;
               state <= s_memread;
             end
           end
           ...
         endcase
...
s_memread: if ( dmem_ready == 1'b1 ) begin
             we_rf <= 1'b1; ce_z <= 1'b1;
             ce_n <= 1'b1; ce_ip <= 1'b1;
             state <= deassert;
           end
...
deassert: begin
            ce_z <= 1'b0; ce_n <= 1'b0;
            we_rf <= 1'b0; ce_ip <= 1'b0;
            state <= fetch_1;
          end
```

4-105. ábra. *A* ***memread*** *utasítás végrehajtása*

A `memwrite` utasítás az eddigieknél egyszerűbb, mivel ez esetben mindössze egy memóriaírási ciklust kell végrehajtanunk a már előkészített `mar` és `mdr` regiszterek segítségével, a regisztertömbbe visszaírás nem történik, a Z és N állapotjelzőket nem frissítjük. Az egyetlen regiszter, amelynek értékét be kell állítanunk, az az utasításmutató, amelyet ezúttal is eggyel megnövelünk a következő utasítás felhozása előtt (lásd 4-106. ábra).

Az utasításkészlet reguláris jellege, vagyis az egyes utasítások szerkezetének és végrehajtásuk folyamatának „hasonlósága” egyszerűbb, ezáltal kisebb és gyorsabb áramkörökhöz vezet. A MiniCore mikroprocesszor esetén is megfigyelhető ennek az elvnek az érvényesülése; azáltal, hogy az aritmetikai/logikai utasítások – a `mul` utasítás kivételével – lényegében ugyanúgy néznek ki, csupán a konkrét műveletben különböznek, lehetővé válik, hogy a vezérlő állapotgép szempontjából tulajdonképpen egyformának tekintsük őket.

VHDL

```
when execute => case ir(17 downto 14) is
                  ...
                  when i_memwrite =>
                    sel_ip <= '0'; write_dmem <= '1';
                    if ( dmem_ready = '0' ) then
                      write_dmem <= '0';
                      state <= s_memwrite;
                    end if;
                  ...
                end case;
...
when s_memwrite => if ( dmem_ready = '1' ) then
                     ce_ip <= '1'; state <= deassert;
                   end if;
...
when deassert => ce_z <= '0'; ce_n <= '0';
                 we_rf <= '0'; ce_ip <= '0';
                 state <= fetch_1;
```

Verilog

```
execute: case ( ir[17:14] )
           ...
           i_memwrite: begin
             sel_ip <= 1'b0; write_dmem <= 1'b1;
             if ( dmem_ready == 1'b0 ) begin
               write_dmem <= 1'b0;
               state <= s_memwrite;
             end
           end
           ...
         endcase
...
s_memwrite: if ( dmem_ready == 1'b1 ) begin
              ce_ip <= 1'b1; state <= deassert;
            end
...
deassert: begin
            ce_z <= 1'b0; ce_n <= 1'b0;
            we_rf <= 1'b0; ce_ip <= 1'b0;
            state <= fetch_1;
          end
```

4-106. ábra. *A* ***memwrite*** *utasítás végrehajtása*

A 4-107. ábrán látható, hogy hét különböző aritmetikai utasítás végrehajtását vontuk össze egyetlen vezérlésiállapot-sorozatba.

VHDL

```
when execute => case ir(17 downto 14) is
                  ...
                  when i_add | i_sub | i_and | i_or |
                       i_xor | i_shl | i_shr =>
                              sel_rfb <= '1';
                              sel_rfw <= '0';
                              sel_rfdin <= "00";
                              sel_alu <= "10";
                              sel_ip <= '0';
                              state <= s_alu;
                  ...
                end case;
...
when s_alu => we_rf <= '1'; ce_z <= '1';
              ce_n <= '1'; ce_ip <= '1';
              state <= deassert;
...
when deassert => ce_z <= '0'; ce_n <= '0';
                 we_rf <= '0'; ce_ip <= '0';
                 state <= fetch_1;
```

Verilog

```
execute: case ( ir[17:14] )
           ...
           i_add, i_sub, i_and, i_or,
           i_xor, i_shl, i_shr: begin
             sel_rfb <= 1'b1; sel_rfw <= 1'b0;
             sel_rfdin <= 2'b00;
             sel_alu <= 2'b10; sel_ip <= 1'b0;
             state <= s_alu;
           end
           ...
         endcase
...
s_alu: begin
         we_rf <= 1'b1; ce_z <= 1'b1;
         ce_n <= 1'b1; ce_ip <= 1'b1;
         state <= deassert;
       end
...
deassert: begin
            ce_z <= 1'b0; ce_n <= 1'b0;
            we_rf <= 1'b0; ce_ip <= 1'b0;
            state <= fetch_1;
          end
```

4-107. ábra. *Az aritmetikai utasítások végrehajtása összevont vezérlésiállapot-sorozattal*

A vezérlési szempontból összevont utasítások közötti különbség csupán az ALU belsejében érvényesül azáltal, hogy az utasításregiszter [17:14] mezőjében kódolt utasításkód minden utasítás esetén egyedi, az ALU tehát pontosan tudja, hogy a különböző aritmetikai utasítások esetén mit kell tennie, miközben a vezérlő egység nem tesz különbséget ezek között.

A `mul` utasítást szintén az ALU hajtja végre, de ebben az esetben az eredmény 64 bites, amit csak két részletben tudunk visszaírni a 32 bites bejegyzéseket tároló regisztertömbbe. A két visszaírás két, egymást követő órajelciklusban tud megtörténni, amelyek vezérlési

VHDL

```
when execute => case ir(17 downto 14) is
                  ...
                  when  i_mul => sel_rfb <= '1';
                                 sel_rfw <= '0';
                                 sel_rfdin <= "00";
                                 sel_alu <= "10";
                                 sel_ip <= '0';
                                 state <= s_mul_1;
                  ...
                end case;
...
when s_mul_1 => we_rf <= '1';
                state <= s_mul_2;
when s_mul_2 => ce_ip <= '1';
                sel_rfw <= '1'; sel_rfdin <= "01";
                state <= deassert;
...
when deassert => ce_z <= '0'; ce_n <= '0';
                 we_rf <= '0'; ce_ip <= '0';
                 state <= fetch_1;
```

Verilog

```
execute: case ( ir[17:14] )
           ...
           i_mul: begin
             sel_rfb <= 1'b1; sel_rfw <= 1'b0;
             sel_rfdin <= 2'b00;
             sel_alu <=  2'b10; sel_ip <= 1'b0;
             state <= s_mul_1;
           end
           ...
         endcase
...
s_mul_1: begin
           we_rf <= 1'b1; state <= s_mul_2;
         end
s_mul_2: begin
           ce_ip <= 1'b1; sel_rfw <= 1'b1;
           sel_rfdin <= 2'b01; state <= deassert;
         end
...
deassert: begin
            ce_z <= 1'b0; ce_n <= 1'b0;
            we_rf <= 1'b0; ce_ip <= 1'b0;
            state <= fetch_1;
          end
```

4-108. ábra. *A `mul` utasítás végrehajtását végző vezérlésiállapot-sorozat*

szempontból mindössze a visszaírási cím forrását (`sel_rfw`: `ir[13:11]` vagy `ir[13:11]+1`) és a visszaírandó adat forrását (`sel_rfdin`: `alu_result[31:0]` vagy `alu_result[63:32]`) kiválasztó vezérlő jelekben különböznek egymástól (lásd 4-108. ábra).

A `mul` utasítás még egy részletben különbözik a többi aritmetikai/logikai utasítástól, nevezetesen hogy ebben az esetben a Z és N állapotjelzők – az utasításkészlet specifikációjának megfelelően – nem frissülnek, ezek órajel-engedélyező bemenetét tehát a vonatkozó vezérlésiállapot-sorozatban nem módosítjuk. Az ALU belső megvalósításában ez egy erőforrásigény-csökkentő egyszerűsítési lehetőséget jelent, aminek azonban ára van: Ha a szoftver kifejlesztése során mégiscsak arra lenne szükségünk, hogy például egy szorzás eredményének nulla volta alapján ágaztassunk el egy algoritmust, akkor a szorzás során visszaírt, két 32 bites eredményt további utasítások segítségével kell megvizsgálnunk.

A `div` utasítás funkcióját tekintve ugyan aritmetikai utasítás, de vezérlési szempontból igencsak eltér a többi aritmetikai utasítástól, mert ezt a műveletet nem az ALU, hanem egy másik műveletvégző elem, egy iteratív osztó áramkör végzi el. A vezérlő állapotgép feladata, hogy az osztó áramkör interfészének időzítési modellje alapján „rávegye” az osztó áramkört a szükséges művelet elvégzésére, majd annak végeztével visszaírja az osztás eredményét a regisztertömbbe (lásd 4-109. ábra). A vezérlő állapotgép feladata továbbá az is, hogy valaa mi módon kezelje a nullával való osztást, amit az osztó áramkör rögtön a második operandusának megvizsgálása után a `zero_div` kimenetének magas szintre állításával jelez. Ebben az esetben a vezérlő állapotgép nem ír vissza eredményt a regisztertömbbe, hanem a processzormag `error` kimenetét magas szintre állítja, és a programvégrehajtást felfüggeszti.

VHDL

```
when execute => case ir(17 downto 14) is
                  ...
                  when i_div => sel_rfb <= '1';
                                sel_rfw <= '0';
                                sel_rfdin <= "10";
                                sel_ip <= '0';
                                state <= s_div_1;
                  ...
                end case;
...
when s_div_1 => rqst_div <= '1';
                state <= s_div_2;
when s_div_2 => state <= s_div_3;
when s_div_3 => if ( div_rdy = '1' ) then
                  rqst_div <= '0';
                  if ( zero_div = '1' ) then
                    error <= '1';
                    run <= '0';
                    state <= deassert;
                  else
                    we_rf <= '1';
                    state <= s_div_4;
                  end if;
                end if;
when s_div_4 => ce_ip <= '1'; sel_rfw <= '1';
                sel_rfdin <= "11";
                state <= deassert;
...
when deassert => ce_z <= '0'; ce_n <= '0';
                 we_rf <= '0'; ce_ip <= '0';
                 state <= fetch_1;
```

Verilog

```
execute: case ( ir[17:14] )
           ...
           i_div: begin
             sel_rfb <= 1'b1; sel_rfw <= 1'b0;
             sel_rfdin <= 2'b10;
             sel_ip <= 1'b0; state <= s_div_1;
           end
           ...
         endcase
...
s_div_1: begin
           rqst_div <= 1'b1; state <= s_div_2;
         end
s_div_2: state <= s_div_3;
s_div_3: if ( div_rdy == 1'b1 ) begin
           rqst_div <= 1'b0;
           if ( zero_div == 1'b1 ) begin
             error <= 1'b1; run <= 1'b0;
             state <= deassert;
           end else begin
             we_rf <= 1'b1; state <= s_div_4;
           end
         end
s_div_4: begin
           ce_ip <= 1'b1; sel_rfw <= 1'b1;
           sel_rfdin <= 2'b11;
           state <= deassert;
         end
...
deassert: begin
            ce_z <= 1'b0; ce_n <= 1'b0;
            we_rf <= 1'b0; ce_ip <= 1'b0;
            state <= fetch_1;
          end
```

4-109. ábra. *A `div` utasítás végrehajtását végző vezérlésiállapot-sorozat*

Ha az osztás második operandusa nem nulla, tehát az osztás elvégezhető, akkor a szorzás művelethez hasonlóan ezúttal is két 32 bites eredmény születik (hányados és maradék), tehát a visszaírásra vonatkozó vezérlési állapotok az ott látottakhoz hasonlóak.

A feltétel nélküli (`jump`) és feltételes (`branchzero`, `branchnegative`) ugró utasítások végrehajtása nem igényel segédállapotokat. Feltétel nélküli esetben egyszerűen be kell töltenünk az utasításmutató regiszterbe az utasításregiszter [9:0] mezőjében tárolt ugrási címet, a `branchzero` és `branchnegative` utasítások esetében pedig rendre a Z és N állapotjelzők aktuális értékétől függően vagy szintén az ugrási címet, vagy az utasításmutató inkrementált értékét töltjük az utasításmutató regiszterbe. Az utasításmutató regiszter új értékének forrását a `sel_ip` vezérlő jellel választhatjuk ki (lásd 4-110. ábra).

VHDL

```
when execute => case ir(17 downto 14) is
                  ...
                  when i_jump =>
                    sel_ip <= '1'; ce_ip <= '1';
                    state <= deassert;

                  when i_branchzero =>
                    if ( z = '1' ) then sel_ip <= '1';
                    else sel_ip <= '0'; end if;
                    ce_ip <= '1';
                    state <= deassert;

                  when i_branchnegative =>
                    if ( n = '1' ) then sel_ip <= '1';
                    else sel_ip <= '0'; end if;
                    ce_ip <= '1';
                    state <= deassert;
                  ...
                end case;
...
when deassert => ce_z <= '0'; ce_n <= '0';
                 we_rf <= '0'; ce_ip <= '0';
                 state <= fetch_1;
```

Verilog

```
execute: case ( ir[17:14] )
           ...
           i_jump: begin
             sel_ip <= 1'b1; ce_ip <= 1'b1;
             state <= deassert;
           end
           i_branchzero: begin
             if ( z == 1'b1 ) sel_ip <= 1'b1;
             else sel_ip <= 1'b0;
             ce_ip <= 1'b1; state <= deassert;
           end
           i_branchnegative: begin
             if ( n == 1'b1 ) sel_ip <= 1'b1;
             else sel_ip <= 1'b0;
             ce_ip <= 1'b1;
             state <= deassert;
             end
           ...
         endcase
...
deassert: begin
            ce_z <= 1'b0; ce_n <= 1'b0;
            we_rf <= 1'b0; ce_ip <= 1'b0;
            state <= fetch_1;
          end
```

4-110. ábra. *Az ugró utasítások végrehajtását végző vezérlési állapotok*

A `stop` utasítás szerepe az, hogy a `run` kimenet értékét nullára állítva jelezze a külvilág felé, hogy a programvégrehajtást maga a processzoron futó szoftver le akarja állítani (lásd 4-111. ábra).

VHDL

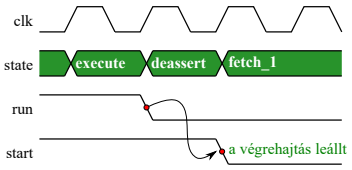
```
...
when execute => case ir(17 downto 14) is
                  ...
                  when i_stop => run <= '0';
                                 state <= deassert;
                  ...
                end case;
...
when deassert => ce_z <= '0'; ce_n <= '0';
                 we_rf <= '0'; ce_ip <= '0';
                 state <= fetch_1;
```

Verilog

```
...
execute: case ( ir[17:14] )
           ...
           i_stop: begin
             run <= 1'b0; state <= deassert;
           end
           ...
         endcase
...
deassert: begin
            ce_z <= 1'b0; ce_n <= 1'b0;
            we_rf <= 1'b0; ce_ip <= 1'b0;
            state <= fetch_1;
          end
```

4-111. ábra. *A `stop` utasítás végrehajtását végző vezérlési állapotok*

Funkcionálisan ugyan nem lenne szükség arra, hogy a `run` kimenet alacsony szintre állítása után a `stop` utasítás végrehajtása a `deassert` állapotba lépjen, ütemezési szempontból azonban fontos; ez a beiktatott órajelciklus teszi lehetővé a mikroprocesszort vezérlő külső egység számára, hogy „reagáljon” a `run` állapotjelző nullára állítására, és ugyancsak nullára állítsa a `start` bemenet értékét, megakadályozva, hogy a `fetch_1` állapotba lépve újabb utasításfelhozási ciklus kezdődjön (lásd 4-112. ábra).



4-112. ábra. *A programvégrehajtás leállításának kezdeményezése a* ***stop*** *utasítással*

5

Funkcionális verifikáció

Ha összevetjük a MiniCore mikroprocesszornak a 3.1. alfejezetbeli, C++ nyelvű leírását a 4.9. alfejezetben bemutatott VHDL és Verilog reprezentációkkal, akkor azt a következtetést vonhatjuk le, hogy egy digitális áramkör viselkedési és RTL modellje között részletgazdagság, az elvárt működést megvalósító megoldások és az azokhoz szükséges erőforrások konkretizálása szempontjából meglehetősen nagy szakadék lehet, vagyis az RTL tervezés során rengeteg megvalósítási részletet dolgozunk ki. Éppen ezért nem meglepő, hogy a tervezés során – különösen, ha olyan összetett rendszerekről van szó, mint például egy SoC áramkör – nagy hangsúlyt fektetünk a HDL leírással specifikált RTL modellek helyességének ellenőrzésére, vagyis a funkcionális verifikációra. A mai modern, nagyintegráltságú digitális rendszerek kifejlesztése során a befektetett munkának kb. hetven százalékát a funkcionális verifikáció teszi ki.

Ahogy azt az RTL tervezéssel kapcsolatos fejezetben is tettük, a funkcionális verifikáció elméleti hátterének és fogalomkészletének részletes tárgyalása előtt ezúttal is egy egyszerű példán keresztül szemléltetjük, hogy lényegében miről is szól a funkcionális verifikáció.

Tegyük fel, hogy adott egy egyszerű számláló áramkör elvárt funkcióját rögzítő specifikáció az alábbi, informális, szöveges formában:

- **REQ-001**: Az áramkör magas-aktív, aszinkron reset hatására alapállapotba áll.
- **REQ-002**: Az áramkörnek az aszinkron reset hatására beálló alapállapotában minden kimenet logikai alacsony szintű.
- **REQ-003**: Az aszinkron resetet leszámítva az áramkör teljesen szinkron működésű; az áramkör a bemeneteinek megváltozására csak az órajel felfutó éleinél reagál, kimenetei pedig szintén csak az órajel felfutó élei hatására változnak meg.
- **REQ-004**: Az áramkör 4 bites, bináris felfelé számlálást valósít meg a `cnt` kimenetén.
- **REQ-005**: A bináris 1111 érték elérése után a számláló átfordul, tehát a kimenet a bináris 0000 értéket veszi fel.
- **REQ-006**: A számláló engedélyezhető/tiltható az `ena` bemenet segítségével. Ha a számláló letiltott állapotban van, akkor a `cnt` kimenete a bemenetektől független és konstans, a tiltás előtti értéket tartja.
- **REQ-007**: Ha a számláló engedélyezve van, tehát az `ena` bemenet logikai magas értékű, akkor a `load` és `parallel_in` bemenetek segítségével tetszőleges értéket betölthetünk a számlálóba. Alacsony engedélyező bemenet esetén a párhuzamos betöltés funkció nem működik.

- **REQ-008**: Az áramkör előállít egy `even` állapotjelzőt a számláló mindenkori értéke alapján. Ha a `cnt` kimenet előjel nélküli egész számként értelmezve páros, akkor az `even` kimenet értéke magas, egyébként alacsony.

Adott továbbá a fenti specifikáció alapján készült, hardverleíró nyelvű RTL modell. A funkcionális verifikáció során a HDL modell belső megvalósítását nem vizsgáljuk, hiszen feladatunk „csupán" annak megállapítása, hogy a HDL modell megfelel-e a specifikációban rögzített előírásoknak, tehát elegendő, ha a portlista rendelkezésünkre áll (lásd 5-1. ábra).

VHDL

```
entity counter is
  port (
    clk, rst:     in  std_logic;
    ena, load:    in  std_logic;
    parallel_in:  in  unsigned (3 downto 0);
    cnt:          out unsigned (3 downto 0);
    even:         out std_logic
  );
end entity counter;
```

Verilog

```
module counter (
  input   logic clk, rst,
  input   logic ena, load,
  input   logic [3:0] parallel_in,
  output  logic [3:0] cnt,
  output  logic even
);
```

5-1. ábra. *A szöveges specifikáció alapján készült HDL modell portlistája*

A specifikáció célja végső soron nem más, mint hogy meghatározza, hogy a leendő áramkör adott bemeneti gerjesztés hatására milyen viselkedést kell hogy produkáljon a kimenetein. Értelemszerűen az RTL modellnek ugyanezt a viselkedést kell megvalósítania. Feladatunk tehát csak annyi, hogy az RTL modell bemeneteit meghajtjuk a specifikációban közvetlenül vagy közvetetten meghatározott gerjesztési mintákkal és megfigyeljük, hogy azok milyen viselkedést váltanak ki az áramköri modell kimenetein. Az egymástól függetlenül vizsgálható gerjesztési mintákat és az azokra adott válaszokat teszteseteknek nevezzük.

Szükségünk lesz tehát egy olyan eszközre, ami a HDL modellek időbeli (dinamikus) viselkedését képes szimulálni. Talán nem meglepő módon az ilyen számítógépes szoftvereket HDL szimulátoroknak hívjuk. A gerjesztések előállításához és az RTL modell kimeneteinek megfigyeléséhez létrehozunk egy ún. verifikációs környezetet, vagy röviden tesztkörnyezetet (próbapad, *testbench*), ami valójában maga is egy speciális HDL tervezési egység (VHDL-ben entity, Verilogban module), amely példányosítja a verifikálandó HDL modellt (*Design Under Verification*, DUV[44]), meghajtható és megfigyelhető jeleket kapcsol annak be- es kimeneteire, valamint a bemeneteken előállítja a specifikáció egyes pontjainak való megfelelés igazolását lehetővé tévő gerjesztési mintákat. Az 5-2. ábra a számláló áramá körünk verifikációjához használható tesztkörnyezet vázát mutatja be.

A tesztkörnyezet olyan HDL egyed/modul, amelynek portlistája egyetlen portot sem tartalmaz. Ez elsőre talán furcsának tűnhet, de végül is logikus, a portlista azon jelek halmaza, amely egy egyed/modul belső erőforrásait a külvilággal összekapcsolja. A tesztkörnyezet azonban maga a külvilág, amely esetén külső interfészről nincs értelme beszélni (VHDL: 5. sor, Verilog: 1. sor). A tesztkörnyezetet leíró HDL modellben egy-egy belső jelet deklarálunk a DUV interfészjeleihez (VHDL: 10-14. sor, Verilog: 3-7. sor). Az RTL modell

[44] Gyakran találkozhatunk a *Design Under Test* (DUT) kifejezéssel is, de mivel a tesztelés fogalmilag eltér a verifikációtól, ezért a DUV pontosabb.

VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity testbench is
end entity testbench;

architecture bhv of testbench is

  signal clk, rst:    std_logic             := '0';
  signal ena, load:   std_logic             := '0';
  signal parallel_in: unsigned (3 downto 0) := X"0";
  signal cnt:         unsigned (3 downto 0);
  signal even:        std_logic;

begin

  DUV:  entity work.counter(rtl)
          port map (
            clk => clk,
            rst => rst,
            ena => ena,
            load => load,
            parallel_in => parallel_in,
            cnt => cnt,
            even => even
          );

  process begin
    wait for 10 ns; clk <= not clk;
  end process;

  process begin
    --...
    wait;
  end process;

end architecture bhv;
```

Verilog

```verilog
module testbench ();

  logic clk, rst;
  logic ena, load;
  logic [3:0] parallel_in;
  logic [3:0] cnt;
  logic even;

  counter DUV(
    .clk(clk),
    .rst(rst),
    .ena(ena),
    .load(load),
    .parallel_in(parallel_in),
    .cnt(cnt),
    .even(even)
  );

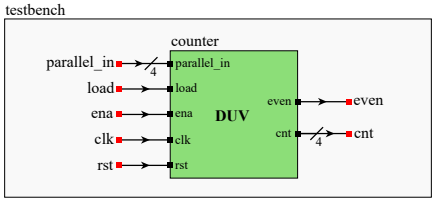
  initial begin
    clk <= 1'b0;
    rst <= 1'b0;
    ena <= 1'b0;
    load <= 1'b0;
    parallel_in <= 4'd0;
  end

  always #10 clk <= ~clk;

  initial begin
    //...
  end

endmodule
```

Áramköri struktúra



5-2. ábra. *A számláló áramkörhöz készített tesztkörnyezet váza*

bemeneteit meghajtó jeleket a szimuláció elején kezdeti értékkel látjuk el. Ennek nyelvi eszköze VHDL-ben a jeldeklaráció részét képező kezdetiérték-adás (*initial value*, lásd a jeldeklarációknál: „:=” szimbólum), Verilogban pedig az ún. initial-blokk, ami egy érzékenységi lista nélküli, a szimuláció elején egyetlen egyszer kiértékelődő, sorrendi utasításblokk. A Verilog modell 19. sorában kezdődő initial-blokk öt, egymástól független, nem blokkoló értékadást tartalmaz, amelyek tehát a szimuláció elején azonnal, és csak egyetlen egyszer kiértékelődnek, ami gyakorlatilag azt jelenti, hogy kezdeti értékkel látjuk el az értékadások bal oldalán lévő jeleket. A tesztkörnyezeten belül létrehozunk egy példányt a verifikálandó RTL modellből, amelynek portjaira az imént deklarált jeleket kapcsoljuk (VHDL: 18-27. sor, Verilog: 9-17. sor). A vizsgált RTL modell bemeneteit meghajtó jeleknek

a specifikációban előírt gerjesztési mintáknak megfelelő dinamikus viselkedését egy vagy több sorrendi utasításblokk (VHDL-ben process, Verilogban always- vagy initial-blokk) segítségével fejezzük ki. Periodikus viselkedés leírására a VHDL process és a Verilog always-blokk érzékenységi lista nélküli változatai kiválóan alkalmasak, hiszen e nyelvi elemek önmaguk is periodikus viselkedésűek; a törzsükben lévő utasítások a szimuláció kezdetétől fogva ciklikusan, újra és újra kiértékelődnek. A VHDL-beli jel/változó értékadásokról, valamint a Verilogbeli blokkoló/nem-blokkoló értékadásokról a 4.4.4. alfejezetben elmondottak itt is érvényesek. Példánkban a `clk` jelet meghajtó órajelforrás periodikus viselkedését a VHDL modellben a 29-31. sorban lévő process, a Verilog modellben a 27. sorban lévő always-blokk írja le. Az órajel élei között eltelt időt a sorrendi utasításblokkok kiértékelésének ideiglenes felfüggesztését végző utasítások (VHDL-ben a `wait for <t>` utasítás, Verilogban a `#<t>` utasítás[45]) segítségével modellezzük. Nem-periodikus gerjesztés leírására olyan sorrendi utasításblokkokat használunk, amelyek kiértékelése a szimuláció kezdetén indul, azonban nem indul újra, amikor a törzsükben található összes utasítás kiértékelésére sor került. Ennek megvalósítására Verilogban a korábban is említett initial-blokk használható (29-31. sor). VHDL-ben nincs külön sorrendi utasításblokk ennek a viselkedésnek a leírására, ezért itt egy olyan process-t használunk (33-36. sor), amelynek utolsó utasítása egy, az utasításblokk kiértékelését véglegesen felfüggesztő, paraméter nélküli `wait` utasítás (35. sor). A gerjesztési mintákat e sorrendi utasításblokkokon belül a verifikálandó RTL modell bemeneteit meghajtó jelekre vonatkozó ütemezett értékadásokkal írjuk le. Az ilyen tesztkörnyezeteket lineáris tesztkörnyezeteknek nevezzük, utalva arra, hogy az egyes gerjesztési minták egyszerűen sorban követik egymást.

Egészítsük ki a tesztkörnyezetet úgy, hogy a REQ-001 és REQ-002 követelménynek való megfelelés igazolható legyen (lásd 5-3. ábra).

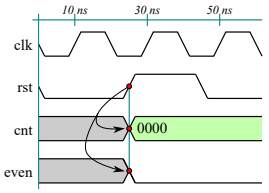
VHDL

```
process begin
  wait for 25 ns; rst <= '1';
  wait for 20 ns; rst <= '0';
  wait;
end process;
```

Verilog

```
initial begin
  #25 rst <= 1'b1;
  #20 rst <= 1'b0;
end
```

A szimuláció eredménye



5-3. ábra. *Az aszinkron reset funkciót ellenőrző teszteset*

A HDL szimuláció eredménye az ún. hullámforma-megjelenítők segítségével vizsgálható. Az 5-3. ábra jobb oldala a számláló áramkör interfészjeleinek viselkedését, vagyis a tesztkörnyezet HDL szimulációjának eredményét mutatja be.

[45] A két megoldás lényegében ugyanaz, azzal az apró különbséggel, hogy míg VHDL-ben a kiértékelés felfüggesztésének időtartamát egy idő dimenziójú kifejezéssel adhatjuk meg, addig a Verilog modellben ez egy dimenzió nélküli szám, ami azt adja meg, hogy a felfüggesztés időtartama hányszorosa a szimuláció időlépésének. A fenti példában feltételezzük, hogy a Verilog szimuláció időlépése 1 ns (RTL szimulációknál ez az általánosan használt időlépés), így a `clk` jel élei között a VHDL tesztkörnyezethez hasonlóan 10 ns telik el, az órajel frekvenciája tehát mindkét tesztkörnyezetben 50 MHz.

Az 5-4. ábra a felfelé számlálás funkció ellenőrzéséhez, vagyis a REQ-004 követelménye nek való megfelelés igazolásához szükséges tesztesettel egészíti ki a tesztkörnyezetet.

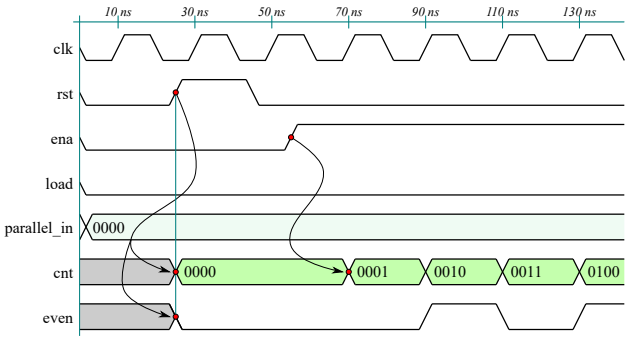
VHDL

```vhdl
process begin
  wait for 25 ns; rst <= '1';
  wait for 20 ns; rst <= '0';
  wait for 10 ns; ena <= '1';
  wait;
end process;
```

Verilog

```verilog
initial begin
  #25 rst <= 1'b1;
  #20 rst <= 1'b0;
  #10 ena <= 1'b1;
end
```

A szimuláció eredménye



5-4. ábra. *A felfelé számlálás funkció ellenőrzése*

Készítsünk olyan tesztesetet is, amely az engedélyezés/tiltás és a párhuzamos betöltés funkciók összefüggéseit (REQ-007) vizsgálja meg (lásd 5-5. ábra).

A logikai alacsony engedélyező bemenet mellett a párhuzamos betöltés funkció működik, ami ellentmond a specifikációban foglaltaknak (REQ-007). Találtunk tehát egy funkcionális hibát, aminek az okát sejthetjük ugyan – a számláló működését leíró `if-else` szerkezetben a bemenetek prioritását az RTL tervező vélhetően rosszul kódolta –, de a verifikáció során feladatunk nem az, hogy feltárjuk a probléma okát és kijavítsuk a modellt.[46] Feladatunk a hibásnak vélt működés pontos dokumentálása, különös tekintettel az ahhoz vezető gerjesztési minta rögzítésére.

Az imént szándékosan nem „hibás", hanem „hibásnak vélt" működésről beszéltünk. Ennek oka, hogy a fenti, egyszerű példával ellentétben egy összetettebb DUV esetén egy ilyen helyzetben korántsem biztos, hogy valóban az RTL modell hibájáról van szó. Vizsgáljuk meg az 5-6. ábrán látható gerjesztést és az RTL modellnek az erre a gerjesztésre adott válaszát.

Ebben a tesztesetben ugyancsak a párhuzamos betöltés funkciót használjuk, ezúttal azonban nem magát a funkciót verifikáljuk, hiszen ezt már megtettük korábban, hanem

[46] Az RTL tervező a kódjába való belekontárkodásra vérmérsékletétől függően a szemöldökráncolástól a nyolc napon túl gyógyuló testi sértésig szinte bármivel reagálhat, amire jó oka van; egy bonyolult áramkör esetén a teljes HDL modellt, ezáltal a hibák kijavítására tett kísérletek hatását is a tervező látja át legjobban. A funkcionális verifikáció során felmerülő problémákat ezért mindig a modellt készítő RTL tervező(k), a verifikációs mérnök(ök) és – később részletezett okok miatt – a specifikáció készítője együttműködve oldják meg.

VHDL

```
process begin
  wait for 25 ns; rst <= '1';
  wait for 20 ns; rst <= '0';
  wait for 10 ns; ena <= '1';

  wait for 50 ns;
  parallel_in <= "1100";
  load <= '1';
  wait for 20 ns; load <= '0';

  wait for 20 ns; ena <= '0';

  wait for 20 ns;
  parallel_in <= "0110";
  load <= '1';
  wait for 20 ns; load <= '0';
  wait;
end process;
```

Verilog

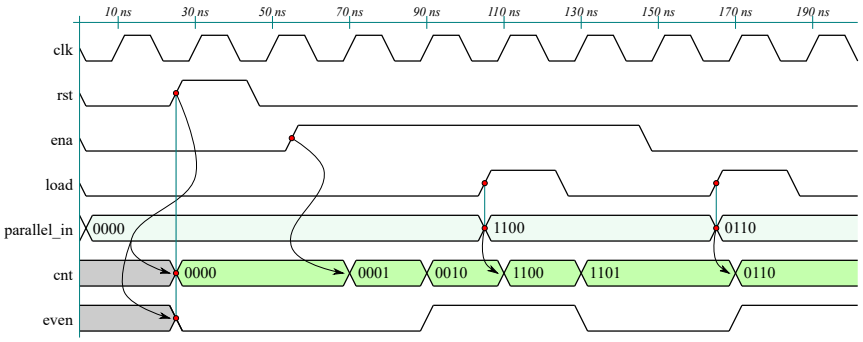
```
initial begin
  #25 rst <= 1'b1;
  #20 rst <= 1'b0;
  #10 ena <= 1'b1;

  #50
  parallel_in <= 4'b1100;
  load <= 1'b1;
  #20 load <= 1'b0;

  #20 ena <= 1'b0;

  #20
  parallel_in <= 4'b0110;
  load <= 1'b1;
  #20 load <= 1'b0;
end
```

A szimuláció eredménye



5-5. ábra. *Az engedélyezés/tiltás és a párhuzamos betöltés funkciók összefüggéseinek ellenőrzése*

segítségével az átforduláshoz közeli állapotba szeretnénk hozni a számlálót, hogy magát az átfordulást ellenőrizhessük. Ebben az esetben azt várjuk, hogy a betöltés sikerül, de a szimuláció nem támasztja alá várakozásunkat. A gerjesztést jobban szemügyre véve azt látjuk, hogy a `load` jelen előállított pozitív impulzus olyan rövid, hogy felfutó és lefutó éle között az órajelnek nincs felfutó éle. A specifikáció egyértelműen rögzíti, hogy a számláló teljesen szinkron működésű kell, hogy legyen, tehát bármely bemenetének változására csak az órajel felfutó éleinél tud reagálni. Ez tulajdonképpen nem jelent mást, mint hogy ebben az esetben a hiba nem az RTL modellben van, hanem a tesztesethez tartozó gerjesztésben. Korrigáljuk a tesztesetet, majd vizsgáljuk meg a számláló átfordulását (lásd 5-7. ábra).

A hullámformára tekintve a verifikációs mérnök már írja is a hibajegyet, amely szerint a számláló átfordulásakor, amikor a kimenet értéke nullát vesz fel, akkor az `even` flag nulla, ami nyilván hibás, hiszen a nulla az páros szám. Az RTL tervező erre azt válaszolja, hogy a specifikáció azt is rögzíti, hogy a flag értékét a `cnt` kimenet mindenkori értéke alapján kell meghatározni, továbbá azt is, hogy reset hatására MINDEN kimenetnek nullának kell lennie, tehát kénytelen volt olyan nyakatekert logikát tervezni az `even` flag előállításához, ami minden, nullától eltérő számról helyesen meg tudja állapítani, hogy páros vagy páratlan,

VHDL

```
process begin
  wait for 25 ns; rst <= '1';
  wait for 20 ns; rst <= '0';
  wait for 10 ns; ena <= '1';

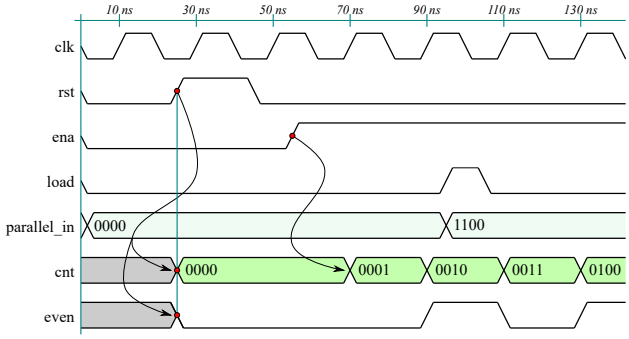
  wait for 40 ns;
  parallel_in <= "1100";
  load <= '1';
  wait for 10 ns; load <= '0';
  wait;
end process;
```

Verilog

```
initial begin
  #25 rst <= 1'b1;
  #20 rst <= 1'b0;
  #10 ena <= 1'b1;

  #40
  parallel_in <= 4'b1100;
  load <= 1'b1;
  #10 load <= 1'b0;
end
```

A szimuláció eredménye



5-6. ábra. *A párhuzamos betöltés funkció látszólag nem működik*

VHDL

```
process begin
  wait for 25 ns; rst <= '1';
  wait for 20 ns; rst <= '0';
  wait for 10 ns; ena <= '1';

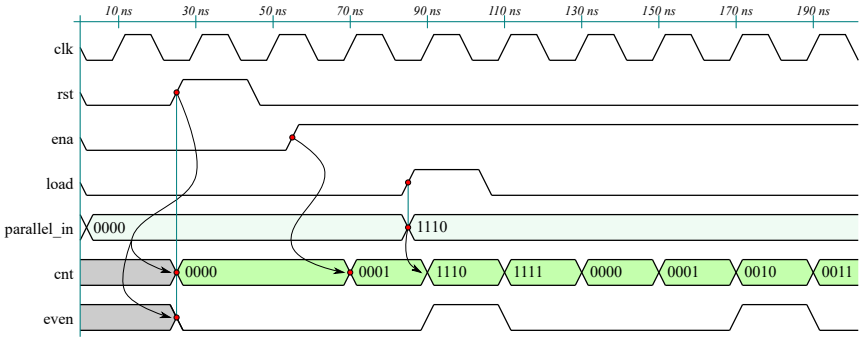
  wait for 30 ns;
  parallel_in <= "1100";
  load <= '1';
  wait for 20 ns; load <= '0';
  wait;
end process;
```

Verilog

```
initial begin
  #25 rst <= 1'b1;
  #20 rst <= 1'b0;
  #10 ena <= 1'b1;

  #30
  parallel_in <= 4'b1100;
  load <= 1'b1;
  #20 load <= 1'b0;
end
```

A szimuláció eredménye



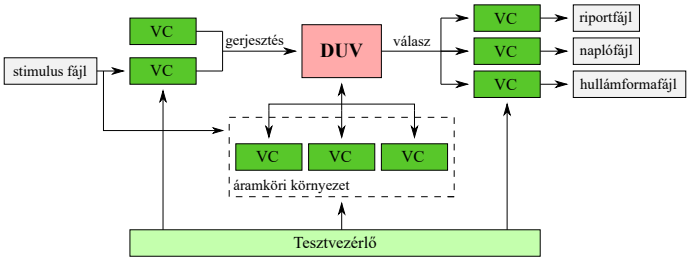
5-7. ábra. *A számláló átfordulásának ellenőrzése*

kivéve a nullát.[47] Ebben az esetben tehát nem az RTL modellt és nem is a gerjesztést, hanem magát a specifikációt kell hibásnak tekintenünk, hiszen önellentmondást tartalmaz (hacsak nem tényleg erre a fura működésű even flagre van szükség, amit már furcsasága miatt is érdemes egyértelműen rögzíteni a specifikációban).

A funkcionális verifikáció során tehát az RTL modell kimenetein tapasztalt, hibásnak vélt működés nemcsak magának az RTL modellnek a hibájából következhet. Az egész verifikációs folyamat célja, hogy az itt bemutatottakhoz hasonló problémákra fény derüljön, azokat a specifikáció készítője, az RTL tervező és a verifikációs mérnök tisztázza,[48] így közelítve egymáshoz a leendő áramkörrel kapcsolatos elképzeléseket.

5.1 A verifikációs környezet

A verifikációs környezet feladata a rendszer áramköri környezetének modellezése olyan módon, hogy a külvilág és a DUV kölcsönhatásának megfigyelése alapján lehetővé váljon annak megállapítása, hogy az RTL modell által leírt viselkedés megfelel-e a specifikációnak. Ebből a szempontból a fejezet bevezetőjében bemutatott verifikációs környezet nagyon egyszerű, hiszen ott az áramköri környezet tulajdonképpen csak néhány jelből áll. A gyakorlatban a verifikációs környezetek lényegesen bonyolultabbak lehetnek. A DUV bemeneteit meghajtó jelek gerjesztési mintáinak korábban bemutatott közvetlen megfogalmazása csak viszonylag egyszerű esetekben hatékony, és ugyanez vonatkozik a DUV válaszainak hullámforma-megjelenítő segítségével való közvetlen, manuális vizsgálatára is. A bemeneti gerjesztések előállítását és a kimeneti jelek megfigyelését összetett verifikációs környezetekben ún. verifikációs komponensek (*Verification Component*, VC) végzik. Egy verifikációs környezet általános blokkvázlatát mutatja be az 5-8. ábra.



5-8. ábra. *Összetett verifikációs környezetek általános modellje*

A verifikációs komponensek legegyszerűbb esetben a DUV-hez hasonlóan olyan funkcionális elemek, amelyeket hardverleíró nyelvek segítségével írunk le. E hardverleíró nyelvű modellek azonban a felhasználható nyelvi elemek szempontjából lényegesen eltérnek

[47] Hozzáteszi továbbá, hogy ennek ő sem örült, amikor látta a specifikációt, mert hasonló esetekben mindig az LSB-re szokott akasztani egy invertert, és a probléma meg van oldva, itt pedig extra logikát kellett kitalálnia, hogy a specifikációnak megfeleljen... elvégre a vevőnek mindig igaza van.

[48] Összetett funkció esetén itt nem egy-egy személyt, inkább egész tervezői csoportokat kell az egyes szerepek mögé képzelni.

magától a DUV-től. Ahogy a 4. fejezetben láthattuk, egy RTL modell viselkedésének és szerkezetének leírásához nem túl sokféle nyelvi elemre van szükségünk, egész egyszerűen azért, mert a digitális áramkörök nem túl sokféle alapvető funkcionális elemből épülnek fel. A hardverleíró nyelvek által kínált nyelvi eszközöknek az RTL modellezésnél használt részhalmazát a nyelv szintetizálható részhalmazának nevezzük, mert ezek azok a nyelvi elemek, amelyeket nemcsak a szimulációs, hanem a szintézis szoftverek is képesek feldolgozni, belőlük kapuszintű modellt képesek előállítani. A verifikációs komponensek esetén nincs ilyen korlátozás, hiszen ezeket a modulokat csak szimulációs célra használjuk, nem vetjük őket alá automatizált áramkörszintézisnek. Ennek megfelelően a verifikációs komponensek működésének leírásához minden rendelkezésre álló nyelvi elemet felhasználhatunk, és érdemes is felhasználnunk, mert így sokkal összetettebb viselkedésmintákat sokkal egyszerűbben és gyorsabban tudunk kifejezni.

Feladatkörüket tekintve a verifikációs komponensek sokfélék lehetnek. Ha a DUV bemeneteinek gerjesztési mintáit nem közvetlenül a verifikációs környezet HDL modelljébe kódolva rögzítjük, hanem az előírt tesztesetek definícióit valamilyen külső fájlban tároljuk, akkor egy verifikációs komponenst használhatunk arra, hogy ezt a fájt a szimuláció során beolvassa, értelmezze, és a benne tárolt adatok alapján előállítsa a DUV bemeneteit közvetlenül meghajtó jelek gerjesztési mintáit. A DUV kimeneteinek megfigyelésére készíthetünk olyan verifikációs komponenseket, amelyek a szimuláció során kinyerhető összes információt nem túl strukturáltan rögzítő hullámforma helyett csak egy adott vizsgálati szempont szerint lényeges, megszűrt információt rögzítenek naplófájlok és riportfájlok formájában.

Ugyancsak verifikációs komponenseket használhatunk a DUV-t körülvevő egyéb funkcionális egységeknek, vagyis a leendő áramköri környezet elemeinek viselkedési modellezésére. Ezeknek a verifikációs komponenseknek elsősorban annak igazolásában van szerepük, hogy a DUV megfelelően valósítja meg az interfészeire előírt követelményeket, kommunikációs protokollokat, ezért maguknak a verifikációs komponenseknek nem kell feltétlenül az általuk képviselt funkcionális egység teljes viselkedését részletesen leírniuk, elegendő, ha az interfészjeleken megfelelően „utánozzák" annak viselkedését. Az ilyen, csak egy adott interfészspecifikációnak (pl. PCIe, WISHBONE, AXI stb.) megfelelő tranzakciók modellezésére alkalmas, de funkcionálisan nem teljes verifikációs komponenseket BFM *(Bus Functional Model)* modelleknek nevezzük.

A verifikációs komponenseknek egy fontos típusát képviselik az ún. referenciamodellek, amelyek a korábban említett verifikációs komponensekkel ellentétben nem a DUV áramköri környezetének elemeit, hanem magát a DUV-t modellezik, jellemzően viselkedési szinten. Egy referenciamodellre épülő verifikációs környezetbe a DUV-t és a referenciamodellt egyaránt beépítjük, majd egyazon gerjesztéssel meghajtva őket azt ellenőrizzük, hogy ugyanazt az eredményt produkálják-e a kimeneteiken. A referenciamodellt – ahogy arra elnevezése is utal – funkcionálisan helyesnek feltételezzük, ha tehát a DUV ugyanarra a gerjesztésre ugyanazt a választ állítja elő, mint amit a referenciamodell, akkor a DUV RTL megvalósítását helyesnek tekintjük. Általában az „egyazon gerjesztés" alatt nem azt értjük, hogy a két modell bemeneteire ugyanazokat a jeleket kapcsoljuk, majd a kimeneteket minden időpillanatban egyesével összehasonlítjuk. A referenciamodell jellemzően az

RTL modellnél elvontabb, így interfésze is egészen más jellegű, mint egy RTL modellé (adott esetben lehet akár egy egyszerű függvényhívás is), így a gerjesztés fogalmát is magasabb elvonatkoztatási szinten kell értelmeznünk. Egy szabványos interfésszel ellátott aritmetikai áramkör verifikációjához például készíthetünk olyan referenciamodellt, amely egyáltalán nem valósítja meg az interfész-protokollt, csupán az aritmetikai alapfunkciót. Ilyenkor a DUV-t és a referenciamodellt a saját, egyedi interfészüknek megfelelően el kell látnunk ugyanazokkal a bemeneti adatokkal, majd a számítás végeztével az eredményeiket ugyancsak a saját interfészük segítségével ki kell olvasnunk belőlük, hogy összehasonlíthassuk azokat. Egy ilyen verifikációs környezetben tehát a DUV alapfunkcióját – vagyis azt, hogy az adott számítási műveletet helyesen végzi el – egy referenciamodell segítségével igazoljuk, az interfészspecifikációnak való megfelelés ellenőrzésére pedig használhatjuk a korábban említett BFM modelleket.

A verifikációs környezetnek ugyancsak fontos eleme a tesztvezérlő *(test sequencer)*, amely a különböző tesztesetek végrehajtásának ütemezését, a verifikációs komponensek működésének összehangolását végzi. Legegyszerűbb esetben a tesztvezérlőt egy, a bevezető példában is látható, a DUV interfészjeleit közvetlenül vagy verifikációs komponenseken keresztül, közvetetten meghajtó folyamatként képzelhetjük el.

5.2 Verifikációs komponensek

Funkcionálisan teljes viselkedési modellek • Tegyük fel, hogy feladatunk a MiniCore mikroprocesszor RTL modelljének funkcionális verifikációja. A processzormag áramköri környezetéről a rendszerterv alapján tudjuk, hogy biztosan tartalmaz egy utasításmemóriát és egy adatmemóriát. A mag verifikációja során ezeket az elemeket a verifikációs környezetbe egy-egy verifikációs komponensként építhetjük be, amelyek a processzor és a memóriák közötti kommunikációs interfész 4-92. ábrán szemléltetett időzítési modelljének megfelelően működnek. Az 5-9. ábra az adatmemóriát képviselő verifikációs komponens interfészdeklarációját mutatja be.

Mivel a memóriainterfész definíciója nem rögzíti a memóriamodul ún. feléledési idejét és beírási/kiolvasási késleltetését, ezért a viselkedési modellekben ezeket paraméterként

VHDL

```
entity dmem_vc is
  generic (
    wakeup_latency: time;
    write_latency:  time;
    read_latency:   time
  );

  port (
    read, write:  in  std_logic;
    ready:        out std_logic;
    address:  in  std_logic_vector (9 downto 0);
    data_in:  in  std_logic_vector (31 downto 0);
    data_out: out std_logic_vector (31 downto 0)
  );
end entity dmem_vc;
```

Verilog

```
module dmem_vc
#(
  parameter wakeup_latency,
  parameter write_latency,
  parameter read_latency
)

(
  input   logic         read, write,
  output  logic         ready,
  input   logic [9:0]   address,
  input   logic [31:0]  data_in,
  output  logic [31:0]  data_out
);
```

5-9. ábra. *Az adatmemóriát leíró verifikációs komponens interfészdeklarációja*

VHDL

```vhdl
process
  begin

    wait on write, read;

    if ( rising_edge(write) ) then
      wait for wakeup_latency;
      ready <= '0';
      wait for write_latency;
      content(to_integer(unsigned(address))) <= data_in;
      ready <= '1';
    elsif ( rising_edge(read) ) then
      wait for wakeup_latency;
      ready <= '0';
      wait for read_latency;
      data_out <= content(to_integer(unsigned(address)));
      ready <= '1';
    end if;
  end process;
```

Verilog

```verilog
always begin

  @(write, read);

  if ( write == 1'b1 ) begin
    #wakeup_latency;
    ready <= 1'b0;
    #write_latency;
    content[address] <= data_in;
    ready <= 1'b1;
  end else if ( read == 1'b1 ) begin
    #wakeup_latency;
    ready <= 1'b0;
    #read_latency;
    data_out <= content[address];
    ready <= 1'b1;
  end

end
```

5-10. ábra. *Az adatmemóriát leíró verifikációs komponens viselkedési modellje*

valósítottuk meg, amelyeket csak egy konkrét verifikációs környezetbe való beültetéskor kell meghatározni. A memóriamodul viselkedésének leírásakor bármilyen nyelvi elemet felhasználhatunk, ami jócskán leegyszerűsítheti modelljeinket (lásd 5-10. ábra).

Figyeljük meg, hogy míg a memóriainterfésznek a processzor-oldali megvalósítása a MiniCore mikroprocesszor RTL modelljében egy állapotgépet, több vezérlési állapotot és két különálló regisztert (`mar`, `mdr`) igényelt, addig a verifikációs komponens HDL leírásában ugyanezt a kommunikációs protokollt néhány utasítás segítségével le tudtuk írni.

Az utasításmemória interfészének időzítési modellje az adatmemóriáéval megegyezik, a hozzá tartozó verifikációs komponens tehát hasonló lesz. Két dologban van eltérés; az utasításmemória egyrészt nem valósítja meg a memóriába való beírás funkciót, hiszen a processzor csak olvasni fogja az utasításmemóriát, továbbá tartalmazza a processzor által futtatni kívánt utasításokat. Ezek az utasítások a beíró interfész hiányában ebben a modellben egyszerűen a belső memóriatartalmat leíró tömbnek a szimuláció legelején történő inicializálásával kerülnek bele az utasításmemóriába. Az 5-11. ábra az utasításmemóriát leíró verifikációs komponens HDL modelljét mutatja be. A szimuláció elején ezt a memóriát három `loadimm` utasítással inicializáljuk, amelyek a processzor belső regisztertömbjének 0., 1. és 2. rekeszébe rendre a 0, 1 és 2 értékeket töltik be.

BFM-ek • Az adat- és utasításmemóriát leíró verifikációs komponensek viselkedési szintű, funkcionálisan teljes modellek, hiszen valóban megvalósítják azt, amit egy memóriának meg kell valósítania; adatokat tárolnak el és hozzáférhetővé teszik azokat a külvilág számára egy meghatározott interfészen keresztül. A verifikációs komponensek az általuk képviselt egység funkcióját nem feltétlenül modellezik teljes egészében, vagy azért, mert az túlságosan összetett, vagy azért, mert a DUV ellenőrzése szempontjából felesleges. Tegyük fel, hogy a MiniCore mikroprocesszor maghoz az adatmemórián kívül egy UART perifériavezérlő áramkört is illesztünk a processzor címtartományára leképezett perifériaként. Ez azt jelenti, hogy a processzor által a `memread` és `memwrite` utasítások végrehajtása során az adatmemória-interfészen kezdeményezett olvasási és írási ciklusokra a megszólított memóriacímtől függően vagy az adatmemória, vagy az UART perifériavezérlő válaszol.

VHDL

```vhdl
entity imem_vc is
  generic (
    wakeup_latency:  time;
    read_latency:    time
  );

  port (
    read:        in  std_logic;
    ready:       out std_logic := '1';
    address:     in  std_logic_vector (9 downto 0);
    data_out:    out std_logic_vector (17 downto 0)
                     := (others => '0')
  );
end entity imem_vc;

architecture bhv of imem_vc is

  type content_t is array (0 to 1023) of
    std_logic_vector (17 downto 0);

  signal content: content_t := (
    0 => B"0000_000_00000000000", -- loadimm $0, 0
    1 => B"0000_001_00000000001", -- loadimm $1, 1
    2 => B"0000_010_00000000010"  -- loadimm $2, 2
  );

begin

  process begin
    wait on read;
    if ( rising_edge(read) ) then
      wait for wakeup_latency;
      ready <= '0';
      wait for read_latency;
      data_out <=
        content(to_integer(unsigned(address)));
      ready <= '1';
    end if;
  end process;

end architecture bhv;
```

Verilog

```verilog
module imem_vc
#(
  parameter wakeup_latency,
  parameter read_latency
)(
  input   logic         read,
  output  logic         ready,
  input   logic [9:0]   address,
  output  logic [17:0]  data_out
);

  logic [17:0]  content [0:1023];

  initial begin
    ready <= 1'b1;
    data_out <= 32'd0;
    content[0] <= 18'b0000_000_00000000000;
    content[1] <= 18'b0000_001_00000000001;
    content[2] <= 18'b0000_010_00000000010;
  end

  always begin

    @(read);

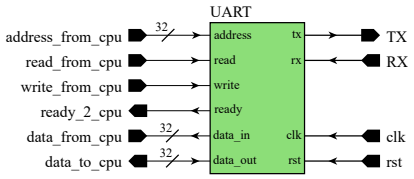
    if ( read == 1'b1 ) begin
      #wakeup_latency;
      ready <= 1'b0;
      #read_latency;
      data_out <= content[address];
      ready <= 1'b1;
    end

  end

endmodule
```

5-11. ábra. *Az utasításmemóriát leíró verifikációs komponens viselkedési modellje*

Ehhez az UART perifériavezérlőt úgy kell elkészíteni, hogy megvalósítsa a 4-92. ábrán bemutatott interfész-specifikációt. Az 5-12. ábra egy ilyen UART perifériavezérlő áramkörnek az interfészét szemlélteti.



5-12. ábra. *A MiniCore processzorhoz illeszthető UART perifériavezérlő interfészjelei*

Az UART perifériavezérlő RTL modelljének funkcionális verifikációja során ellenőriznünk kell a processzor-oldali interfészspecifikációnak való megfelelést. Kézenfekvő lenne a mikroprocesszor teljes RTL modelljét verifikációs komponensként példányosítani az UART modul verifikációs környezetében, majd olyan `memread` és `memwrite` utasítássorozatokat végrehajtatni vele, amelyek az UART vezérlőt megszólítják. Ez a megoldás

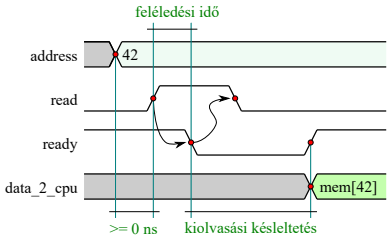
VHDL

```
procedure minicore_read (
  target_address:       in  std_logic_vector (31 downto 0);
  data_read:            out std_logic_vector (31 downto 0);
  signal clk:           in  std_logic;
  signal read:          out std_logic;
  signal ready:         in  std_logic;
  signal address:       out std_logic_vector (31 downto 0);
  signal data_2_cpu:    in  std_logic_vector (31 downto 0)
) is begin
  wait until rising_edge(clk);
  address <= target_address;
  read <= '1';
  wait until falling_edge(ready);
  wait until rising_edge(clk);
  read <= '0';
  wait until rising_edge(ready);
  data_read := data_2_cpu;
end procedure;
```

Verilog

```
task automatic minicore_read (
  input   logic [31:0] target_address,
  output  logic [31:0] data_read,
  ref     logic clk,
  ref     logic read,
  ref     logic ready,
  ref     logic [31:0] address,
  ref     logic [31:0] data_2_cpu
);
  @(posedge clk);
  address = target_address;
  read = 1'b1;
  @(negedge ready);
  @(posedge clk);
  read = 1'b0;
  @(posedge ready);
  data_read = data_2_cpu;
endtask
```

Az olvasás művelet időzítési modellje



5-13. ábra. *A MiniCore mikroprocesszor adatmemória-interfészének olvasási ciklusát modellező BFM*

működőképes ugyan, de elég körülményes, ráadásul ha egy bonyolultabb, szabványos interfésztípusról (AMBA, AXI stb.) és egy, a beágyazott rendszerekben elterjedten használt mikroprocesszorról van szó, egyáltalán nem biztos, hogy annak teljes RTL vagy akár csak viselkedési szintű modellje rendelkezésre áll. Ilyen esetekben használjuk a BFM modelleket az interfészspecifikációnak való megfelelés ellenőrzésére. A BFM-ek olyan, jellemzően viselkedési szinten megfogalmazott verifikációs komponensek, amelyek egy adott modulnak nem a teljes funkcióját, csupán egy megadott interfészén tapasztalható viselkedését modellezik. A BFM-ek hardverleíró nyelvű megvalósítására használhatunk különálló modulokat (VHDL entity, Verilog module) vagy ún. eljárásokat. Ezeket a BFM típusokat rendre modul-alapú és eljárás-alapú BFM-eknek nevezzük. Az alábbiakban az elterjedtebb megoldásra, az eljárás-alapú BFM-ekre mutatunk példát.

Az eljárások a függvényekhez hasonló nyelvi eszközök, amelyeket a VHDL `procedure`-nek, a Verilog `task`-nak nevez. A 4.4.4. pontban a Gray-enkóder/dekóder példáján láthattuk, hogy függvények segítségével miként tudunk összetett kombinációs hálózatokat hatékonyan leírni; törzsük sorrendi utasításblokkot képez, így a bemenetek és kimenetek között akár igen bonyolult összefüggéseket is röviden írhatunk le, de mindig szem előtt kell tartanunk, hogy bármilyen bonyolult is a függvény törzse, a benne foglalt utasítások kiértékelése mindig egyetlen szimulációs ciklus alatt történik meg. Ez RTL tervezési szempontból azt jelentette, hogy a leírt funkciót annak összetettségétől függetlenül egy tároló elemeket nem tartalmazó kombinációs hálózatként kell elképzelnünk. A függvények és eljárások közötti

legfőbb különbség, hogy ez utóbbiak törzse olyan utasításokat is tartalmazhat, amelyek ideiglenesen felfüggesztik az adott eljárás végrehajtását, így a paraméterlistájukon átadott jelek segítségével egészen összetett kommunikációs protokolloknak megfelelő gerjesztési mintákat is elő tudnak állítani.[49] Az 5-13. ábra a MiniCore mikroprocesszor adatmemória-interfészének olvasási, az 5-14. ábra pedig az írási ciklusát modellező BFM-et mutatja be.

A BFM-eket megvalósító eljárások paraméterei két kategóriába sorolhatók; az egyik kategóriába tartozó paraméterek a felhasználó és a BFM, a másikba tartozók pedig a BFM és a DUV között teremtenek kapcsolatot. Ennek megfelelően a továbbiakban e két, eltérő funkciójú paraméterhalmazra „felhasználói interfészként" és „DUV interfészként" hivatkozunk. Mindkét csoportban értelmezhetők ki- és bemeneti paraméterek. A felhasználói interfész esetén a bemeneti paraméterek azok az adatok, amelyekre a BFM-nek szüksége van a tranzakció végrehajtásához (az írási és olvasási ciklusokat megvalósító BFM-ek esetén ez a megszólítandó memóriacím, `target_address`), a kimeneti paraméterek pedig azok az adatok, amelyeket a tranzakció során a felhasználó visszakap a DUV-tól a BFM közvetítésével (az olvasási ciklus esetén a kiolvasott memóriarekesz tartalma, `data_read`, az írási ciklus esetén ilyen paraméter nincs). A DUV interfészen a bemeneti paraméterek azok a jelek, amelyeket a BFM a tranzakció végrehajtása során olvas (`clk`, `ready`, `data_2_cpu`), a kimeneti paraméterek pedig azok, amelyeket ír (`read`, `write`, `address, data_from_cpu`).

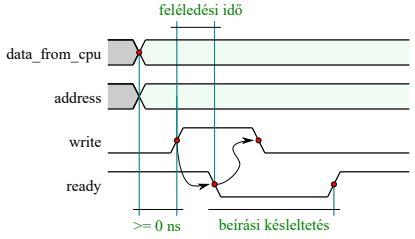
VHDL

```
procedure minicore_write (
  target_address:       in  std_logic_vector (31 downto 0);
  data_2_write:         in  std_logic_vector (31 downto 0);
  signal clk:           in  std_logic;
  signal write:         out std_logic;
  signal ready:         in  std_logic;
  signal address:       out std_logic_vector (31 downto 0);
  signal data_from_cpu: out std_logic_vector (31 downto 0)
) is begin
  wait until rising_edge(clk);
  data_from_cpu <= data_2_write;
  address <= target_address;
  write <= '1';
  wait until falling_edge(ready);
  wait until rising_edge(clk);
  write <= '0';
  wait until rising_edge(ready);
end procedure;
```

Verilog

```
task automatic minicore_write (
  input   logic [31:0] target_address,
  input   logic [31:0] data_2_write,
  ref     logic clk,
  ref     logic write,
  ref     logic ready,
  ref     logic [31:0] address,
  ref     logic [31:0] data_from_cpu
);
  @(posedge clk);
  data_from_cpu = data_2_write;
  address = target_address;
  write = 1'b1;
  @(negedge ready)
  @(posedge clk)
  write = 1'b0;
  @(posedge ready);
endtask
```

Az írás művelet időzítési modellje



5-14. ábra. *A MiniCore mikroprocesszor adatmemória-interfészének írási ciklusát modellező BFM*

[49] A sorrendi utasításblokkok végrehajtását ideiglenesen felfüggesztő utasítások a HDL-ek nem szintetizálható nyelvi szerkezetei közé tartoznak.

Minden teszteset egy-egy teljes busztranzakciót jelent, végrehajtása pedig a következő lépésekből áll:

1. A BFM felhasználói interfészén keresztül átadjuk a végrehajtandó tranzakció bemeneti paramétereit és azokat a jeleket, amelyekben a tranzakció végén előálló kimeneti paramétereket várjuk. A BFM-híváskor átadjuk a BFM DUV interfészének jeleit is, vagyis a verifikációs környezet azon belső jeleit, amelyeken a BFM-nek a busztranzakciót végre kell hajtania. A Verilog modellben a `ref` kulcsszó (referencia) arra utal, hogy az itt átadott jelekről az eljárás meghívásakor nem készül másolat, tehát az eljárás a verifikációs környezet belső jeleit fogja közvetlenül meghajtani.[50]
2. A BFM az adott busz-tranzakcióra vonatkozó szabályok, vagyis a buszprotokoll alapján a DUV interfész jelein végrehajtja a tranzakciót.
3. Ha a tranzakcióban a felhasználó visszakap valamilyen adatot a DUV-től, akkor ezeket az adatokat a BFM elhelyezi a felhasználói interfész kimeneti paramétereiben, majd visszatér.

VHDL

```
L_DUV: entity work.uart(rtl)
         port map (
           clk       => clk,
           rst       => rst,
           read      => read,
           write     => write,
           address   => address,
           data_in   => data_from_cpu,
           data_out  => data_2_cpu,
           ready     => ready,
           tx        => tx,
           rx        => rx
         );

process begin
  wait for 10 ns; clk <= not clk;
end process;

process
  variable data_from_uart: std_logic_vector (31 downto 0);
begin
  wait for 230 ns; rst <= '1';
  wait for 123 ns; rst <= '0';

  wait for 453 ns;
  minicore_write(
    X"00000400", X"0000000E",
    clk, write, ready, address, data_from_cpu
  );

  wait for 450 ns;
  minicore_read(
    X"00000400", data_from_uart,
    clk, read, ready, address, data_2_cpu
  );

  wait;
end process;
```

Verilog

```
uart DUV(
  .clk(clk),
  .rst(rst),
  .read(read),
  .write(write),
  .address(address),
  .data_in(data_from_cpu),
  .data_out(data_2_cpu),
  .ready(ready),
  .tx(tx),
  .rx(rx)
);

initial begin
  clk <= 1'b0; rst <= 1'b0;
  write <= 1'b0; read <= 1'b0;
  address <= 32'd0;
  data_from_cpu <= 32'd0;
  rx <= 1'b1;
end

always #10 clk <= ~clk;

initial begin
  #230 rst <= 1'b1;
  #123 rst <= 1'b0;

  #453 minicore_write(
    32'h00000400, 32'h0000000E,
    clk, write, ready, address, data_from_cpu
  );

  #450 minicore_read(
    32'h00000400, data_from_uart,
    clk, read, ready, address, data_2_cpu
  );

end
```

5-15. ábra. *A MiniCore mikroprocesszorhoz illeszthető UART modul verifikációs környezete*

[50] Ez ismerős lehet a C++ nyelvből, ugyanezt a megoldást ott is referencia szerinti paraméterátadásnak nevezzük.

Az 5-13. és 5-14. ábrán látható BFM-ek rendelkezésre állása esetén az UART perifériavezérlő modul verifikációs környezete tulajdonképpen ugyanolyan egyszerű lehet, mint amit a bevezetőben, a számláló esetén is láthattunk; nincs másra szükség, mint hogy példányosítsuk a DUV-t, vagyis az egyes ki- és bemeneteihez egy-egy vezérelhető és megfigyelhető jelet kapcsoljunk, majd azokon a szükséges gerjesztési mintákat előállítsuk. E gerjesztési mintákat azonban ezúttal nem közvetlenül, hanem a BFM-eken keresztül, eljáráshívásokkal valósítjuk meg (lásd 5-15. ábra).

Referenciamodell • A referenciamodell típusú verifikációs komponensek használatának szemléltetéséhez tekintsük példaként a 4.8. pontban részletesen megtervezett, iteratív osztó áramkört. Az áramkör RTL modelljének funkcionális verifikációjához a megszokott módon létrehozunk egy verifikációs környezetet, amelyben példányosítjuk az osztó áramkör RTL modelljét (lásd 5-16. ábra).

VHDL

```
L_DUV:  entity work.divisor(rtl) generic map (8)
          port map (
            clk, rst, rqst, rdy, dividend, divisor,
            zero_div, quotient, remainder
          );

process begin
  wait for 10 ns; clk <= not clk;
end process;

L_REF_MODEL: process ( divisor, dividend )
begin
  quotient_ref <= dividend / divisor;
  remainder_ref <= dividend rem divisor;
end process;

L_TEST: process begin
  wait for 230 ns; rst <= '1';
  wait for 123 ns; rst <= '0';
  wait for 145 ns;

  dividend <= to_signed(123, 8);
  divisor <= to_signed(-9, 8);
  rqst <= '1';
  wait until rising_edge(rdy);
  rqst <= '0';

  wait;
end process;
```

Verilog

```
divisor #(8) DUV(
  clk, rst, rqst, rdy, dividend, divisor,
  zero_div, quotient, remainder
);

initial begin
  clk <= 1'b0; rst <= 1'b0; rqst <= 1'b0;
  dividend <= 8'd0; divisor <= 8'd0;
end

always #10 clk <= ~clk;

// REF_MODEL
always @ ( dividend, divisor ) begin
  quotient_ref <= dividend / divisor;
  remainder_ref <= dividend % divisor;
end

initial begin
  #230 rst <= 1'b1;
  #123 rst <= 1'b0;
  #145

  dividend <= 123;
  divisor <= -9;
  rqst <= 1'b1;
  @(posedge rdy);
  rqst <= 1'b0;
end
```

5-16. ábra. *Az előjeles osztó áramkör referenciamodell-alapú verifikációs környezete*

Az RTL modell mellett a verifikációs környezetbe beépítünk egy olyan folyamatot is, amely a `dividend` és `divisor` jelek minden változásakor azonnal kiszámítja azok hányadosát (`quotient_ref`) és az osztás maradékát (`remainder_ref`). Ebben a verifikációs környezetben ez a folyamat fogja betölteni a referenciamodell szerepét. Ezek után nincs más dolgunk, mint különböző operandusokkal elindítani az osztó áramkör működését, majd a `rdy` jel felfutó éleinél ellenőrizni, hogy az RTL modell által előállított `quotient` és `remainder` jeleken ugyanazt az értéket látjuk-e, mint a referenciamodell által előállított `quotient_ref` és `remainder_ref` jeleken.

5.3 Önellenőrző verifikációs környezetek

Az előzőekben bemutatott tesztkörnyezetekben a verifikációs komponensek mindegyikének a gerjesztési minták előállításában volt szerepe. A DUV által ezekre a gerjesztésekre adott válaszok egy ilyen tesztkörnyezetben csak a szimuláció során előálló hullámforma vizsgálatával ellenőrizhetők. A példák egyszerűsége miatt elsőre ez talán nem tűnik komoly problémának, azonban a DUV és a verifikációs komponensek összetettségének növekedésével ez a feladat igen hamar túlságosan is bonyolulttá és időigényessé válik. A verifikációs komponenseknek egyre összetettebb viselkedésmintákat kell megvalósítaniuk, a hullámforma-megjelenítőben pedig egyre több, jellemzően párhuzamosan végbemenő folyamatot kell nyomon követnünk, folyamatosan változó jelek és buszok tömkelegét egyszerre megfigyelve. Ha visszatekintünk az összetett digitális rendszerek tervezésének 1-4. ábrán szemléltetett folyamatára, akkor azt is észre kell vennünk, hogy a verifikáció is iteratív, mivel a DUV több okból is megváltozhat a folyamat során. Egyrészt magának a verifikációnak a célja, hogy felderítse a DUV esetleges funkcionális hibáit, amelyek értelemszerűen csak az RTL modell módosításával orvosolhatók, másrészt a specifikáció is megváltozhat időközben, ami ugyancsak befolyásolhatja a DUV által megvalósított funkciót. A DUV vagy a specifikáció változásai az addig elvégzett verifikációs folyamat eredményeit, a lefuttatott teszteket teljesen érvénytelenítik, ami a hullámformák manuális vizsgálatára épülő, egyébként is fáradságos és időigényes verifikációt teljesen reménytelenné teszi. Minden, a DUV-t érintő változás esetén az összes szimulációt újra kell futtatni, vagyis ún. regressziós[51] tesztelést kell végezni annak eldöntésére, hogy a DUV-n elvégzett változtatások nem rontották-e el a korábban már helyesnek ítélt működést.

A verifikációs környezetek hatékonyságának szempontjából minőségi ugrást jelent, ha olyan verifikációs komponenseket is alkalmazunk, amelyek a DUV által előállított eredményeket is automatikusan megvizsgálják, és egy adott követelmény teljesülésére vonatkozóan egy „helyes/nem helyes" eredményt képesek előállítani. Ha rendelkezésünkre állnak ezek az automatikus ellenőrzést végző komponensek, akkor a regressziós tesztelés ugyancsak automatizálható, ami a DUV-n végzett változtatások újraellenőrzését nagyságrendekkel felgyorsítja. A legalapvetőbb, automatizált regressziós tesztelést lehetővé tévő verifikációs komponensek olyan kisebb-nagyobb HDL kódrészletek, amelyek a verifikációs környezet jeleit olvasva megállapítják egy adott teszteset eredményéről, hogy helyes vagy sem. Ezeket a kódrészleteket check-eknek, az ezeket tartalmazó verifikációs környezeteket pedig „önellenőrző" *(self-checking)* verifikációs környezeteknek nevezzük. Az 5-17. ábra azt mutatja be, hogy az előjeles osztó áramkörhöz készített verifikációs környezetet miként lehet kiegészíteni a referenciamodell és a DUV eredményeit automatikusan összehasonlító check-kel.

Miután az osztó áramkört és ezáltal a referenciamodellt elláttuk ugyanazokkal a bemeneti adatokkal, elindítjuk az osztó áramkör működését. Amikor a `rdy` jel felfutásával az osztó áramkör jelzi, hogy a számítást elvégezte, akkor egy egyszerű feltételvizsgálattal összehasonlítjuk a DUV és a referenciamodell eredményeit. Ennek a rövid kódrészletnek a beépítésével elértük azt, hogy a DUV megváltoztatása és a teszteset újrafuttatása esetén

[51] A „regresszió" latin eredetű szó, jelentése visszalépés, visszafelé mozgás.

VHDL

```
L_REF_MODEL: process ( divisor, dividend )
begin
  quotient_ref <= dividend / divisor;
  remainder_ref <= dividend rem divisor;
end process;

L_TEST: process begin
  wait for 230 ns; rst <= '1';
  wait for 123 ns; rst <= '0';
  wait for 145 ns;

  dividend <= to_signed(123, 8);
  divisor <= to_signed(-9, 8);
  rqst <= '1';
  wait until rising_edge(rdy); rqst <= '0';

  L_CHECK: if ( quotient = quotient_ref and
                remainder = remainder_ref ) then
    report "CHECK OK";
  else report "CHECK FAIL";
  end if;

  wait;
end process;
```

Verilog

```
// REF_MODEL
always @ ( dividend, divisor ) begin
  quotient_ref <= dividend / divisor;
  remainder_ref <= dividend % divisor;
end

initial begin
  #230 rst <= 1'b1;
  #123 rst <= 1'b0;
  #145

  dividend <= 123;
  divisor <= -9;
  rqst <= 1'b1;
  @(posedge rdy); rqst <= 1'b0;

  // CHECK
  if ( quotient == quotient_ref &&
       remainder == remainder_ref )
    $display("CHECK OK");
  else $display("CHECK FAIL");

end
```

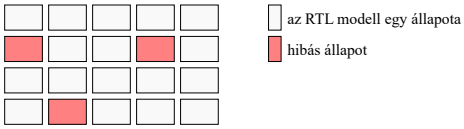
5-17. ábra. *A referenciamodell és a DUV eredményeit automatikusan összehasonlító check*

immár felesleges a hullámforma-megjelenítőben részletesen nyomon követnünk az eseményeket, egész egyszerűen a szimulátor szoftver szabványos kimenetét kell ellenőriznünk, ami sokkal gyorsabban elvégezhető. A check-ek eredményének közlése természetesen más formában, például a megfelelő bejegyzések naplófájlokba való elhelyezésével is elképzelhető, ez már a konkrét verifikációs környezettel szemben támasztott követelmények alapján eldöntendő megvalósítási részlet.

5.4 A gerjesztési minták előállítása

A bevezetőben bemutatott számláló áramkör példáján is jól látszik, hogy már egy viszonylag egyszerű DUV esetén is igen sokféle gerjesztési minta előállítására lehet szükség ahhoz, hogy az összes előírt funkció helyes megvalósításáról meggyőződhessünk. Az egyes tesztesetekhez tartozó gerjesztési minták előállításának többféle módszere létezik. Az e módszerek közötti elvi különbségek szemléltetéséhez definiáljuk a DUV állapotterének fogalmát. A DUV állapottere nem más, mint az RTL modell összes lehetséges, a normál működés során elképzelhető állapotainak halmaza, ahol egy-egy állapot alatt az interfészjelek és a belső tároló elemek egy-egy konkrét értékegyüttesét értjük. Teljesen szinkron működésű RTL modellek esetén elmondható, hogy a DUV az órajel minden felfutó élénél az állapotterének egyik állapotából egy másikba léphet át, vagy megtarthatja az előző felfutó élnél beállt állapotát. Ha a DUV-ban funkcionális hiba van, akkor az állapottérben biztosan lesz olyan állapot, amelyben ennek a hibának a következménye az RTL modell interfészén megfigyelhető,[52] ezeket az állapotokat hibás állapotnak nevezzük (lásd 5-18. ábra).

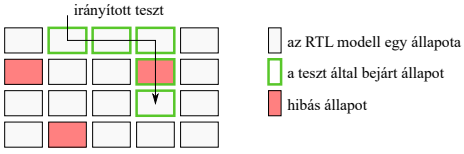
[52] Ez logikailag könnyen belátható; ha egy funkcionális hiba következménye sohasem figyelhető meg az RTL modell kimenetein, akkor az valójában nem is hiba. Ilyen értelemben tehát a hiba létezésének szükséges feltétele annak megfigyelhetősége.



5-18. ábra. *Egy RTL modell állapotterének szemléltetése*

A funkcionális verifikáció során tehát tulajdonképpen nem teszünk mást, mint hogy az egyes tesztesetekhez tartozó gerjesztési minták segítségével igyekszünk a DUV állapotterének minden állapotát bejárni megfigyelhető hibákat keresve. Fontos hangsúlyozni, hogy ha a verifikáció során egy hibás állapotot fedezünk fel, akkor nem a hiba okát, hanem annak következményét figyeltük meg. A hiba oka és következménye között egészen bonyolult összefüggések is lehetnek, térben és időben is igen messze eshetnek egymástól. Ez az oka annak, hogy a verifikációs mérnök feladata nem a hiba okának megállapítása, csupán a megfigyelhető hibajelenséghez vezető gerjesztési minta rögzítése, a hibaok felderítése már ismét az RTL tervező feladata.

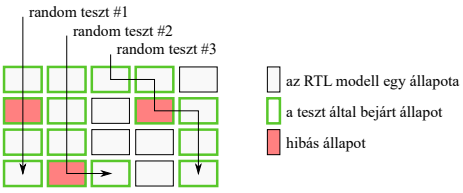
Az állapottér fogalmának tisztázása után térjünk vissza tehát a gerjesztési minták előállításának különböző módszereire. Akkor, amikor egy konkrét funkció ellenőrzését célzó, specifikus gerjesztést állítunk elő – ahogy tettük azt a számláló esetén is –, irányított tesztekről beszélünk (lásd 5-19. ábra).



5-19. ábra. *Irányított teszt*

Az irányított tesztek előnye, hogy – megfelelő minőségű specifikáció rendelkezésre állása esetén – a hozzájuk tartozó gerjesztési minták könnyen előállíthatók. Hátrányuk azonban, hogy nem képesek a váratlan hibák felfedésére, amelyekre a teszt írója nem gondol előre. Egy irányított teszt megtervezése során a teszt írója egy ún. hibamodellt alkot, vagyis előre feltételez egy hibajelenséget, például hogy az osztó áramkör nem megfelelően működik, ha mindkét operandusa negatív. E feltételezést aztán egy teszteset megfogalmazásával igazolja vagy cáfolja, úgy, hogy a DUV állapotterét szándékoltan a kiszemelt hibás állapot felé irányítja. Ezt a módszert követve azok a hibás állapotok, amelyek egy előre nem is feltételezett hibaokra vezethetők vissza, nagyon nehezen vagy egyáltalán nem deríthetők fel. Ezt a problémát az ún. véletlenszerű tesztekkel *(random test)* orvosolhatjuk.

A véletlenszerű tesztelés során automatizáltan állítunk elő nagyszámú, hasonló tesztesetet úgy, hogy az egyes tesztesetekhez tartozó gerjesztési minták véletlenszerű adatokat tartalmaznak. Így tulajdonképpen egyszerre tudunk nagy mennyiségben előállítani különböző teszteseteket, amelyek a DUV állapotterének más-más állapotait fedik le, tehát hamar ráakadhatunk az esetleg hibás állapotokra (lásd 5-20. ábra).



5-20. ábra. *Véletlenszerű teszt*

Fontos kiemelni, hogy a véletlenszerűséget csak a referenciamodell esetén látottakhoz hasonlóan magas elvonatkoztatási szinten tudjuk értelmezni. Egy BFM viselkedése például nem lehet véletlenszerű, mert egy konkrét interfészjel-halmaz időbeli viselkedésének az adott protokollnak pontosan megfelelőnek kell lennie, különben az egész teszt értelmetlen lenne. Véletlenszerűnek tekinthetjük azonban például azt, hogy az osztó áramkör operandus-bemeneteire milyen számértékeket kapcsolunk. Az 5-21. ábrán látható példában ciklus segítségével állítunk elő száz darab különböző operandus-párost az osztó áramkör és annak referenciamodellje számára. Figyeljük meg azt is, hogy az operandusok előállításakor külön figyelmet kellett fordítanunk arra, hogy az osztó semmiképp se legyen nulla, hiszen erre a szituációra a referenciamodellünk nincs felkészítve. Ez azonban csak az egyik ok.

VHDL

```
L_REF_MODEL: process ( divisor, dividend )
begin
  quotient_ref <= dividend / divisor;
  remainder_ref <= dividend rem divisor;
end process;

L_TEST: process
  variable seed1, seed2:        integer := 145;
  variable seed3, seed4:        integer := 89;
  variable rand1, rand2:        real;
begin
  wait for 230 ns; rst <= '1';
  wait for 123 ns; rst <= '0';

  for i in 0 to 99 loop
    wait for 145 ns;
    uniform(seed1, seed2, rand1);
    uniform(seed3, seed4, rand2);
    if ( rand2 = 0.0 ) then rand2 := 1.0; end if;
    dividend <= to_signed(integer(rand1*255.0), 8);
    divisor <= to_signed(integer(rand2*255.0), 8);
    rqst <= '1';
    wait until rising_edge(rdy); rqst <= '0';

    L_CHECK: if ( quotient = quotient_ref and
                  remainder = remainder_ref ) then
          report "CHECK OK";
    else
      report "CHECK FAIL";
    end if;
  end loop;

  wait;
end process;
```

Verilog

```
// REF_MODEL
always @ ( dividend, divisor ) begin
  quotient_ref <= dividend / divisor;
  remainder_ref <= dividend % divisor;
end

initial begin
  #230 rst <= 1'b1;
  #123 rst <= 1'b0;

  for ( int i = 0; i <= 99; i = i+1 )
  begin
    #145
    dividend <= $random;
    divisor <= $random;
    if ( divisor == 0 ) divisor = 8'd255;
    rqst <= 1'b1;
    @(posedge rdy); rqst <= 1'b0;

    // CHECK
    if ( quotient == quotient_ref &&
         remainder == remainder_ref )
      $display("CHECK OK");
    else
      $display("CHECK FAIL");
  end

end
```

5-21. ábra. *Véletlenszerű tesztek generálása ciklus segítségével*[53]

[53] A VHDL modellben a véletlenszám-generálásra az `ieee` könyvtár `math_real` csomagjának `uniform` nevű függvényét használtuk fel.

A nullával való osztás problémája egyúttal a véletlenszerű tesztek fő hátrányára is rávilágít, nevezetesen hogy az ún. kulcsállapotok *(corner case)* elérése sokáig tarthat. Ha a példabeli referenciamodellünket fel is készítettük volna a nullával való osztásra, és gondoskodtunk volna arról, hogy ilyen esetekben is modellezzük a DUV-tól elvárt viselkedést, akkor is nagyon sok véletlenszerű tesztesetet kellene lefuttatnunk ahhoz, hogy ez a speciális eset legalább egyszer „véletlenül" előforduljon.

A véletlenszerű teszteknek ezt a hiányosságát rendszerint a lefedendő állapottér méretének csökkentésével oldjuk meg; a szűkített állapottérben a véletlenszerű tesztek hatékonyabban tudnak működni, hamarabb „rátalálnak" az összes lehetséges hibás állapotra. Ezt a módszert szűkített véletlen tesztelésnek nevezzük *(constrained random test)*.

A gyakorlatban egy RTL modell funkcionális verifikációja során nem egyetlen verifikációs környezetet készítünk, és a gerjesztési minták előállítására sem csak egyetlen módszert, hanem jellemzően az itt bemutatott megoldásokat együttesen alkalmazzuk.

5.5 A verifikációs környezetek minősítése

Egy összetett rendszer funkcionális verifikációjának legfőbb nehézsége abban rejlik, hogy a DUV állapottere hihetetlenül nagy lehet, amelyben a váratlan, tehát hibamodellhez nem társított, így irányított tesztekkel le nem fedett hibák felderítése rendkívül sok időt igényelhet. A funkcionális verifikáció elméletileg csak akkor igazolhatná a specifikációnak való megfelelést, ha a teljes állapottér minden egyes állapotáról egyértelműen ki tudnánk jelenteni, hogy nem hibás, ez azonban rendszereink összetettsége miatt gyakorlatilag lehetetlen. Ez egyúttal azt is jelenti, hogy a funkcionális verifikáció valójában csak azt tudja egyértelműen bizonyítani, hogy egy RTL modell hibás (ehhez elég egyetlen hibás állapotot felfedezni), de azt gyakorlatilag képtelenség egyértelműen bizonyítani, hogy az RTL modell helyes. A verifikáció folyamatának előrehaladtával egyre kevesebb új, addig fel nem fedezett hibára derül fény, amelyek „jelentősége" ráadásul egyre csökken, hiszen bekövetkezésük valószínűsége a DUV normál működése során csekély, máskülönben már sokkal korábban felfedeztük volna őket. Felmerül tehát a kérdés, hogy mikor jelenthetjük ki, hogy a funkcionális verifikáció véget ért. Ennek megállapítása inkább gazdasági, mint műszaki kérdés. Egy bizonyos pont után egy új hiba felfedezése a ráfordítandó idő miatt egyszerűen túl költséges, nem kifizetődő. Hogy ez pontosan mikor következik be, az rendszerint az adott rendszer verifikációjáért felelős szakértő szubjektív döntésén múlik.

Vannak azonban módszerek e döntés meghozásának megkönnyítésére. Ezek a módszerek a DUV-n elvégzett tesztek számszerű minősítésén alapulnak. Olyan mérőszámokat definiálunk, amelyek valamilyen szempont alapján kifejezik azt, hogy egy DUV-t milyen mértékben verifikáltunk. E mérőszámoknak kétféle típusát különböztetjük meg:

- **Implicit** mérőszámok azok, amelyeket a funkcionális verifikáció során használt szoftver eszközök, legegyszerűbb esetben a HDL szimulátorok automatikusan állítanak elő.

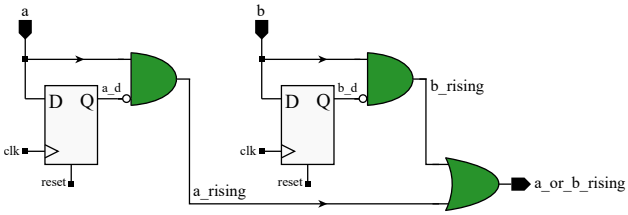
- **Explicitnek** nevezzük azokat a mérőszámokat, amelyeket a verifikáció folyamatának megtervezése során mi magunk definiálunk, értékük pontos meghatározását mi magunk végezzük, vagy legalábbis az azokat kiszámító szoftver-infrastruktúrát (parancsfájlokat, verifikációs komponenseket stb.) manuálisan állítjuk elő.

Kódlefedés ⋆ A legegyszerűbb, implicit mérőszámhalmaz az ún. kódlefedés *(code coverage)*. A kódlefedési mérőszámokat egy konkrét verifikációs környezetben lefuttatott tesztesethalmazra értelmezzük. E mérőszámok azt fejezik ki, hogy az adott tesztesethalmaz a DUV HDL modelljét „mennyire mozgatja meg", a HDL kódbázis hány százaléka értékelődik ki a szimuláció során. A leggyakrabban használt mérőszámok a következők:

- **Utasítás-lefedés** *(statement coverage)*: Azt vizsgálja, hogy a HDL modellt felépítő utasítások (folytonos értékadások, példányosítások és folyamatok) közül melyek értékelődtek ki és melyek nem.
- **Elágázás-lefedés** *(branch coverage, decision coverage)*: Azt mutatja meg, hogy a feltételes utasítások (`if-else` és `case` szerkezetek) mely ágai értékelődtek ki és melyek nem.
- **Feltétel-lefedés** *(condition coverage, expression coverage)*: Azt vizsgálja, hogy a feltételes kifejezések és logikai kifejezések az adott szimuláció lefuttatása során kiértékelődtek-e igazra és hamisra egyaránt.
- **Élváltás-lefedés** *(toggle coverage)*: Azt mutatja meg, hogy a HDL modell egyes jelein a szimuláció során előfordult-e fel- és lefutó él egyaránt.
- **Állapotgép-lefedés** *(FSM coverage)*
  - **Állapot-lefedés** *(state coverage):* Kimutatja, hogy a szimuláció során egy adott állapotgép mely állapotait jártuk be és melyeket nem.
  - **Állapotátmenet-lefedés** *(transition coverage):* Egy állapotgépnek alapvető jellegzetessége, hogy egy állapotból több másik állapotba is átléphet. Ez a lefedési mérőszám megmutatja, hogy a lehetséges állapotátmenetek közül melyek valósultak meg egy szimuláció lefuttatása során és melyek nem.

Fontos hangsúlyozni, hogy a kódlefedés nem tökéletes megoldás annak megállapítására, hogy egy verifikációs környezet vagy több különálló verifikációs környezet együttes tesztjei mennyire járják be a DUV állapotterét, vagyis hogy lehetnek-e még a DUV állapotterében lefedetlen hibás állapotok. Ennek magyarázata abban rejlik, hogy a szoftvertechnológiából származó kódlefedés koncepciója nem veszi figyelembe azt, hogy a HDL modellek alapvetően párhuzamos jellegűek. A kódlefedés segítségével megállapítható, hogy két különálló utasítás lefutott vagy sem, de nem tekintjük külön eseteknek a két különálló kódrészlet egyszerre és egymás után történő kiértékelését. Márpedig a DUV állapottere szempontjából e két eset két külön állapotot jelent. Tegyük fel, hogy feladatunk az 5-22. ábrán látható, speciális éldetektor HDL modelljének elkészítése.

A kettős éldetektor feladata, hogy kimenetén egy pozitív impulzust állítson elő, ha bármely bemenetén felfutó él keletkezik. A kapcsolási sémát specifikációnak tekintve elkészítjük az áramkör HDL modelljét (lásd 5-23. ábra).



5-22. ábra. *Éldetektor két független jel megfigyelésére*

VHDL

```
architecture rtl of dual_edge_detector is
  signal a_d, b_d, a_rising, b_rising: std_logic;
begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      a_d <= '0'; b_d <= '0';
    elsif ( rising_edge(clk) ) then
      a_d <= a; b_d <= b;
    end if;
  end process;
  a_rising <= a and not a_d;
  b_rising <= b and not b_d;
  a_or_b_rising <= a_rising xor b_rising;

end architecture rtl;
```

Verilog

```
module dual_edge_detector (...);

  logic a_d, b_d, a_rising, b_rising;

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) begin
      a_d <= 1'b0; b_d <= 1'b0;
    end else begin
      a_d <= a; b_d <= b;
    end
  end
  assign a_rising = a & ~a_d;
  assign b_rising = b & ~b_d;
  assign a_or_b_rising = a_rising ^ b_rising;

endmodule
```

5-23. ábra. *A kettős éldetektor HDL modellje*

A HDL modellhez természetesen készítünk egy verifikációs környezetet is. A funkció egyszerűsége miatt most nem lesz szükségünk verifikációs komponensekre, mindössze egy egyszerű, lineáris tesztkörnyezetet hozunk létre, amelyben az éldetektor bemeneteit közvetlenül a tesztvezérlő folyamat segítségével hajtjuk meg. Az 5-24. ábra ezt a tesztkörnyezetet mutatja be.

VHDL

```
L_DUV:  entity work.dual_edge_detector(rtl)
          port map (clk, rst, a, b, a_or_b_rising);

process begin
  wait for 10 ns; clk <= not clk;
end process;

process begin
  wait for 23 ns; rst <= '1';
  wait for 34 ns; rst <= '0';

  wait for 45 ns; wait until rising_edge(clk);
  a <= '1';
  wait for 90 ns; wait until rising_edge(clk);
  b <= '1';
  wait for 111 ns; wait until rising_edge(clk);
  a <= '0';
  wait for 89 ns; wait until rising_edge(clk);
  b <= '0';

  wait;
end process;
```

Verilog

```
dual_edge_detector DUV
  (clk, rst, a, b, a_or_b_rising);

always #10 clk <= ~clk;

initial begin
  clk <= 1'b0; rst <= 1'b0; a <= 1'b0; b <= 1'b0;
end

initial begin
  #23 rst <= 1'b0;
  #34 rst <= 1'b0;

  #45 @(posedge clk);
  a <= 1'b1;
  #90 @(posedge clk);
  b <= 1'b1;
  #111 @(posedge clk);
  a <= 1'b0;
  #111 @(posedge clk);
  b <= 1'b0;
end
```

5-24. ábra. *A kettős éldetektorhoz készített verifikációs környezet és tesztesetek*

A kettős éldetektor specifikációja igen egyszerű, bármelyik bemenetén is érkezik egy felfutó él, annak a kimeneten egy pozitív impulzust kell előidéznie. Ennek megfelelően a tesztvezérlőben négy tesztesetet hozunk létre:

1. felfutó él érkezik az `a` bemeneten
2. felfutó él érkezik a `b` bemeneten
3. lefutó él érkezik az `a` bemeneten
4. lefutó él érkezik a `b` bemeneten

A szimuláció során meggyőződünk róla, hogy az első két teszteset végrehajtása során valóban keletkezik egy-egy pozitív impulzus a kimeneten, míg a második két esetben – várakozásainknak megfelelően – nem történik a kimeneten semmi. A fent említett, erre a HDL modellre értelmezhető összes lehetséges kódlefedési mérőszámot megvizsgálva azt látjuk, hogy mindegyik százszázalékos, ebből azonban sajnos nem következtethetünk arra, hogy a teszteseteink a DUV teljes állapotterét lefedve sem találtak hibát. Ez akkor válik nyilvánvalóvá, ha a tesztkörnyezethez hozzáadunk még egy tesztesetet, amelyben a két bemenet pontosan ugyanabban a pillanatban vált alacsony szintről magasra (lásd 5-25. ábra).

VHDL

```
L_DUV:  entity work.dual_edge_detector(rtl)
          port map (clk, rst, a, b, a_or_b_rising);

process begin
  wait for 10 ns; clk <= not clk;
end process;

process begin
  wait for 15 ns; rst <= '1';
  wait for 10 ns; rst <= '0';

  wait for 20 ns; wait until rising_edge(clk);
  a <= '1';
  wait for 35 ns; wait until rising_edge(clk);
  b <= '1';
  wait for 35 ns; wait until rising_edge(clk);
  a <= '0';
  wait for 35 ns; wait until rising_edge(clk);
  b <= '0';
  wait for 35 ns; wait until rising_edge(clk);
  a <= '1'; b <= '1';

  wait;
end process;
```

Verilog

```
dual_edge_detector DUV
  (clk, rst, a, b, a_or_b_rising);

always #10 clk <= ~clk;

initial begin
  clk <= 1'b0; rst <= 1'b0; a <= 1'b0; b <= 1'b0;
end

initial begin
  #15 rst <= 1'b0;
  #10 rst <= 1'b0;

  #20 @(posedge clk);
  a <= 1'b1;
  #35 @(posedge clk);
  b <= 1'b1;
  #35 @(posedge clk);
  a <= 1'b0;
  #35 @(posedge clk);
  b <= 1'b0;
  #35 @(posedge clk);
  a <= 1'b1; b <= 1'b1;
end
```

5-25. ábra. *A kettős éldetektor verifikációs környezete egy újabb teszteset hozzáadása után*

A százszázalékos kódlefedés alapján azt mondhatnánk, hogy ez a hozzáadott teszteset redundáns, felesleges, nem tesztel semmi olyasmit, amit a korábbi tesztesetek ne próbáltak volna ki, a szimuláció eredményéből azonban azt látjuk, hogy a két bemenet egyszerre történő megváltozása esetén az éldetektor kimenete nem változik, ami egyértelmű funkcionális hiba. Találtunk tehát egy hibás állapotot, amelyet a korábbi tesztesetek a százszázalékos kódlefedés ellenére nyilvánvalóan nem fedtek le. A hiba okát egyébként az éldetektor HDL modelljét tüzetesebben szemügyre véve hamar kideríthetjük; a kimeneti OR kapu helyett véletlenül XOR kaput használtunk. Ebből az egyszerű példából is jól látszik, hogy

a kódlefedés miért nem tökéletes megoldás annak megállapítására, hogy a teszteseteink a DUV teljes állapotterét bejárják-e, lehetővé téve az összes hibás állapot felderítését.

Funkcionális lefedés *(functional coverage)* • Az explicit lefedési mérőszámok legelterjedtebben alkalmazott típusa az ún. funkcionális lefedés. Ebben a megközelítésben a DUV-tól elvárt funkciókat valamilyen formában rögzítjük, majd a verifikáció során e rögzített funkciók helyes működésének alátámasztására vagy cáfolására alkalmas teszteseteket fogalmazunk meg. Létrehozunk egy adatbázist az elvárt funkciókról és az azokat lefedő tesztesetekről, így a verifikáció előrehaladtával folyamatosan nyomon követhetjük, hogy melyek a még le nem fedett funkciók. Az Olvasóban joggal merülhet fel a kérdés, hogy ez tulajdonképpen miben is különbözik az irányított teszteléstől, a válasz pedig az, hogy tulajdonképpen semmiben. A funkcionális lefedési mérőszámok alkalmazásának célja, hogy a specifikációnak vagy magának az RTL modellnek a részletes ismeretében módszeresen tervezzük meg a verifikáció folyamatát azáltal, hogy az elvárt funkciókat informálisan vagy formálisan rögzítjük, azok tesztesetekkel való lefedettségének mérésére pedig egyedi megoldásokat dolgozzunk ki magának a verifikáció folyamatának megkezdése előtt. A bevezetőben bemutatott számlálós példa esetén – a specifikáció jellegéből adódóan – könnyen tudunk funkcionális lefedettségi mérőszámot definiálni például úgy, hogy a követelménylista minden egyes elemét egy-egy elvárt funkciónak tekintjük, és a verifikáció során számszerűen nyomon követjük, hogy e követelmények hány százalékához készült teszteset, amely bizonyítottan képes az adott funkció helyes megvalósítását igazolni vagy cáfolni. Ebben az esetben tehát egyetlen funkcionális mérőszámunk van, amely a verifikáció folyamatának készültségi fokára utal, így segíthet eldönteni, hogy a verifikáció folyamatát befejezettnek, az RTL modellt pedig helyesnek tekinthetjük-e.

A kódlefedéshez hasonlóan a funkcionális lefedettségi mérőszámokkal kapcsolatban is fontos megjegyezni, hogy nem garantálják az RTL modellben esetlegesen előforduló hibák maradéktalan felderítését, legfőképpen azért, mert a verifikálandó funkciók halmazának meghatározása nem automatikus, a specifikáció vagy az RTL modell ismeretében azt a verifikációs mérnöknek magának kell elvégeznie. Ez a megoldás is csupán a funkcionális verifikáció módszerességét támogató eszköz, amely csökkenti a felderítetlen hibák előfordulásának valószínűségét.

5.6 A funkcionális verifikáció szintjei

A DUV belső megvalósításának láthatósága, hozzáférhetősége alapján a funkcionális verifikációnak három típusát különböztetjük meg:

- **Black-box** verifikáció alatt azt értjük, amikor a DUV belső felépítése egyáltalán nem ismert, a verifikációs környezet és a DUV közötti kölcsönhatás kizárólag annak interfészén keresztül valósulhat meg. Ez a korlátozás mind beállíthatóság *(controllability)*, mind megfigyelhetőség *(observability)* szempontjából megnehezíti a verifikáció folyamatát. A beállíthatóság nehézsége azt jelenti, hogy lehetnek a DUV

állapotterének olyan állapotai, amelyek pusztán a külső interfészek használatával nehezen, bonyolult és időigényes gerjesztési mintákkal érhetők el. A megfigyelhetőség problémája pedig abban nyilvánul meg, hogy ha a DUV-ban lévő hiba és az interfészen is megfigyelhető hibajelenség térben és/vagy időben távol vannak egymástól, akkor a hibaok felderítése nagyon nehéz feladat. Ugyanakkor a black-box verifikációra az is jellemző, hogy az ennek során kifejlesztett verifikációs komponensek és tesztesetek teljesen függetlenek lehetnek a DUV megvalósításától, ilyen értelemben tehát újrafelhasználhatók. Ha a DUV belső megvalósítása változik is a tervezési folyamat során, a pusztán a külső interfészekkel kölcsönható verifikációs komponensek és tesztesetek továbbra is használhatók, amivel időt takaríthatunk meg.

- **White-box** verifikáció: Ahogy elnevezése is utal rá, a white-box verifikáció során a DUV belső felépítésének minden részlete ismert, a verifikálandó HDL modell teljes egészében rendelkezésre áll. Előnyök és hátrányok szempontjából nem meglepő módon ez az eset épp az ellenkezője a black-box verifikációnak; a kritikusnak ítélt állapotok könnyen előidézhetők (akár kifejezetten e célra bevezetett ún. *injection* interfészek segítségével, akár úgy, hogy a verifikációs környezet közvetlenül a DUV belső jeleire kényszeríti a beállítani kívánt értékeket), a működés pedig közvetlenül megfigyelhető. Az ilyen verifikációs környezetek és az azt alkotó verifikációs komponensek és tesztesetek azonban csak a DUV-nak azzal a konkrét megvalósításával együtt használhatók, amelyre kifejlesztették őket, a DUV változása esetén tehát egyúttal a verifikációs környezetet is módosítanunk kell.
- **Grey-box** verifikáció: Kompromisszumos megoldás a black-box és white-box verifikáció között. Lényegében olyan, black-box jellegű, tehát csak a DUV interfészjeleivel kölcsönható tesztesetekre kell itt gondolnunk, amelyeket a DUV belső felépítésének részleges ismeretében, célirányosan hozunk létre. A DUV részleges ismerete azt jelenti, hogy lehet némi fogalmunk a DUV állapotteréről, tudhatjuk azt, hogy melyek a tesztesetekkel feltétlenül lefedendő, kritikusnak ítélt állapotok és azt is, hogy ezek milyen gerjesztési mintákkal érhetők el hatékonyan.

Almodulszintű tervezői verifikáció • A funkcionális verifikációnak egy gyakran emlegetett alapszabálya, hogy egy adott funkciót megvalósító IP mag verifikációját nem végezheti maga az RTL tervező, mivel ő a verifikáció során csak olyan hibalehetőségekre írná meg a megfelelő teszteseteket, amelyekre valószínűleg már a HDL kódolás során is gondolt. Hatékonyabb, ha a funkcionális verifikációt egy másik ember hajtja végre a specifikáció alapján. Ennek az alapelvnek a bölcsessége kétségbevonhatatlan, ezt mindenki hamar megtapasztalhatja, aki RTL tervezésre és HDL modellek készítésére adja a fejét. Gyakorlati szempontból azonban elég életszerűtlen, hogy a HDL modell úgy álljon össze, hogy a fejlesztés során a tervező semmiféle verifikációt ne végezzen. A HDL modell elkészítése önmagában is egy iteratív folyamat, amely során a tervező a modell helyességét HDL szimulációkkal folyamatosan ellenőrzi. Ez tulajdonképpen egy, a dolog természetéből adódóan white-box jellegű funkcionális verifikáció, a végső HDL modell helyességének igazolását célzó erőfeszítések első lépcsőfoka. A HDL modell összetettségétől függően az ebben

a fázisban kifejlesztett verifikációs környezetek, komponensek és tesztesetek kiforrottsága nagyon eltérő lehet. A verifikációnak ezen a szintjén jellemzőek az egyszerű, lineáris, verifikációs komponenseket nélkülöző tesztkörnyezetek, de a HDL modell összetettségének növekedésével egyre inkább kifizetődővé válik az intelligens, önellenőrző, automatizált regressziós tesztelésre alkalmas verifikációs környezetek létrehozása.

Almodulszintű szisztematikus verifikáció • A HDL modell elkészülte után szisztematikus verifikációt végzünk az almodul specifikációja alapján. Ezt a feladatot a korábban említett okok miatt jellemzően nem a HDL modell készítője végzi. Szisztematikusság alatt itt azt értjük, hogy az ebben a fázisban elkészített verifikációs környezetekkel szemben alapvető követelmény, hogy alkalmasak legyenek automatikus regressziós tesztelésre, és a bennük foglalt tesztesetek a specifikáció alapján összeállított verifikációs terv szerint készüljenek el, ami a funkcionális lefedettség nyomon követését hivatott lehetővé tenni. Nem magukkal a verifikációs környezetekkel, hanem inkább a verifikáció folyamatával szemben támasztott, ugyancsak alapvető követelmény a kódlefedési mérőszámok nyomon követése.

Szisztematikus toplevel verifikáció • Amikor egy makrocella összes almoduljának RTL terve elkészült és átesett az almodulszintű szisztematikus verifikáción, akkor megkezdődhet a toplevel integráció, ami lényegében az almodulokat tömörítő HDL modell létrehozása, valamint az ezek megfelelő együttműködését igazoló funkcionális verifikáció elvégzése. Azt gondolhatnánk, hogy ha az egyes almodulok RTL modelljeinek helyességét már igazoltuk, akkor az ezek összekapcsolásával kapott toplevel modell verifikációja során nagy meglepetés már nem érhet minket; több különálló, önmagában helyes modell összekapcsolva is helyes működést fog produkálni. Sajnos a gyakorlatban egyáltalán nem ez a helyzet. Ha feltételezzük, hogy az almodulok RTL modelljei valóban helyesek, tehát a specifikációjukban foglaltakat maradéktalanul teljesítik, az egyes almodulok együttműködése és a teljes makrocellát leíró RTL modell funkcionális helyessége akkor is két további tényezőn múlik:

- Az almodulokkal szemben támasztott funkcionális követelményeket és azok interfészeinek működését helyesen dokumentáltuk a rendszertervezés során.
- A teljes rendszer funkcionálisan teljes egészként való működését és az egyes almodulok együttműködését helyesen modelleztük a rendszertervezés során.

Ezekre a kulcsfontosságú kérdésekre az almodulok önálló verifikációja nem ad választ, ezért a toplevel RTL modellt ugyancsak funkcionális verifikációnak kell alávetnünk. Ezt a folyamatot integrációs tesztelésnek is nevezzük. Az ennek véghezviteléhez szükséges eszközöket már ismerjük, a különbség csupán az, hogy a toplevel verifikáció során a különböző tesztesetekhez tartozó ellenőrzések, check-ek most magasabb szintű, rendszerszintű követelmények teljesülését fogják ellenőrizni. Ennek mélyebb megértéséhez térjünk vissza ismét a mikrokontrolleres példához. Egy UART perifériavezérlő verifikációja során felhasználjuk a processzormag BFM modelljét a busztranzakciók ellenőrzéséhez, és valószínűleg olyan verifikációs komponenst is használunk, ami UART kommunikációt képes

imitálni. Ezekkel ellenőrizzük, hogy a processzor által az UART-vezérlőnek elküldött adatbájt megjelenik-e a TX vonalon az UART protokollnak megfelelő jelalak formájában. Ellenőrizzük azt is, hogy az UART-vezérlő RX bemenetét megfelelően meghajtva a perifériavezérlő generálja-e a processzormag BFM modellje felé a megszakításokat. A toplevel verifikáció során ezeknek a részleteknek a működését már helyesnek feltételezzük, és magasabb szintű elvárásokat fogalmazunk meg. Készíthetünk olyan verifikációs komponenst, ami egy UART-on vezérelt külső eszközt, például egy vezetéknélküli adóvevőt vagy egy szenzort modellez. A mikrokontroller toplevel modelljének helyes működéséről ez esetben úgy győződhetünk meg, hogy egy konkrét program, egy kész alkalmazás futását szimuláljuk, amely az UART interfésszel ellátott külső eszközt használja valamilyen feladat végrehajtására. Tegyük fel, hogy a rendszertervezés fázisban rosszul, félreérthetően dokumentáltuk az elképzelt buszinterfész működését, és ennek következtében a processzormag RTL modelljének viselkedése eltér a BFM modellétől. Az almodulok önálló verifikációja nem képes felfedni ezt a hibát, nem derül ki, hogy a processzormag és a perifériavezérlő másképp valósítja meg a közös kommunikációs protokollt, hiszen önmaga specifikációjának mindkét RTL modell megfelel. A toplevel verifikáció során azonban a két almodul közvetlenül össze van kapcsolva, így a hibára hamar fény derül.

5.7 Verifikációs keretrendszerek

Az ebben a fejezetben elmondottak alapján megérthető, hogy miért jelent óriási kihívást a bonyolult digitális rendszerek RTL modelljeinek funkcionális verifikációja, és miért kell az RTL tervezésnél lényegesen több időt és energiát fordítani rá. Az is belátható, hogy az RTL tervezés és a funkcionális verifikáció igen szorosan összefüggő, de nagyon eltérő készségeket, szemléletet és eszközöket igénylő feladatkörök. A hardverleíró nyelvek kifejlesztése és elterjedése óta, de különösen az utóbbi egy évtizedben a funkcionális verifikáció módszertana – a digitális rendszereink összetettségének megállíthatatlannak tűnő növekedésétől hajtva – rengeteget fejlődött. Ebben a fejezetben azt láthattuk, hogy a verifikáció alapelveinek (önellenőrző tesztkörnyezetek, verifikációs komponensek, tesztesetek, check-ek) megvalósítására a hardverleíró nyelvek önmagukban is alkalmasak, azonban nagyméretű rendszerek esetén hatékonyságuk nem kielégítő. Már a kilencvenes évek második felében elkezdődött az ún. hardver-verifikációs nyelvek (*Hardware Verification Langage*, HVL, pl. e, OpenVera) kifejlesztése és alkalmazása. E nyelvek célja, hogy hatékony eszközöket nyújtsanak a HDL-alapú RTL tervek funkcionális verifikációjához. Az új, specifikus nyelvek kifejlesztése mellett egy másik fejlesztési irány magának a SystemVerilog nyelvnek a megfelelő továbbfejlesztése volt, aminek eredményeként a SystemVerilogot legalább annyira tartjuk számon hardver-verifikációs, mint klasszikus hardverleíró nyelvként.

A funkcionális verifikáció módszertanának fejlődése nemcsak maguknak az alkalmazott formális nyelveknek a fejlődésében nyilvánul meg. Ahogy az RTL tervezésben az IP-alapú megközelítés alappillérévé a HDL modellek újrafelhasználhatósága vált, ugyanez az elvárás a verifikációs környezetek és különösen a verifikációs komponensek kapcsán is hamar felmerült. Ez az igény vezetett a funkcionális verifikáció módszerét valamelyest formalizáló,

egységesítő keretrendszerek, mint például az eRM *(e Reuse Methodology)*, majd azután az OVM *(Open Verification Methodology)* és végül az UVM *(Unified Verification Methodology)* kifejlesztéséhez. Ezek a keretrendszerek az adhoc, vagy teljesen szabványos, az iparban elterjedten alkalmazott megoldásokat (pl. interfészek) tömörítő, összetett HDL modellek verifikációjához szükséges eszközöket, verifikációs komponenseket a HDL modellekhez hasonlóan újrafelhasználható IP formájában bocsátják a felhasználó rendelkezésére. A konkrét verifikációs komponenseken kívül definiálják a verifikáció készültségi szintjét leíró lefedettségi mérőszámokat és azok összegyűjtésének és dokumentálásának módját is. Ezeket a modern verifikációs módszereket CAD szoftvereink is széles körben támogatják, elmondható tehát, hogy szerves részét képezik az összetett digitális rendszerek tervezési folyamatának.



6

A digitális áramkörök időzítési kérdései

A digitális áramkörökben az órajelnek kulcsfontosságú szerepe van. Ez az az ütemjel, amelynek szabályos időközönként bekövetkező változásai regiszter-transzfer szinten az idő múlásának illúzióját keltik. Ez azonban – regiszter-transzfer szinten – még valóban csupán illúzió, hiszen ezek a modellek nem tartalmazzák az idő fogalmát, az ezen az elvonatkoztatási szinten végzett szimulációk ún. *zero-delay* szimulációk, amelyek csak az események egymásutániságát képesek visszaadni, de azt nem, hogy az egyes események között valójában mennyi idő telik el. Egy RTL modellen futtatott *zero-delay* szimulációnál az órajel felfutó élei között eltelt konkrét időtartam az elvégzett műveletek szempontjából teljesen irreleváns; mindegy, hogy modellünk órajelbemenetére 10 MHz vagy 1 GHz frekvenciájú órajelforrást kapcsolunk, adott számú órajel-felfutóél után modellünk állapota pontosan ugyanaz lesz. Az órajel felfutó éle hatására a jeleken végzett transzformációk pillanatszerűen mennek végbe, garantáltan a következő órajel-felfutóél előtt.

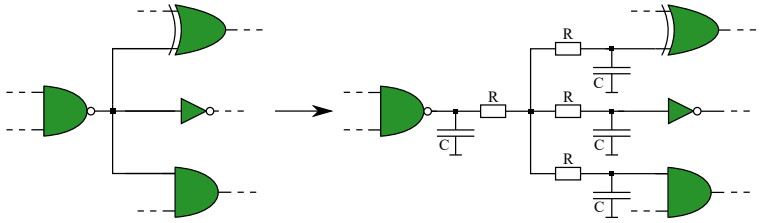
Ez egy hasznos absztrakció, egy modellezési módszer, ami a modellezett rendszerben végbemenő események ok-okozati viszonyait jól leírja, elfedi azonban azt a tényt, hogy a fizikailag megvalósított áramkörökben semmilyen változás nem következhet be pillanatszerűen. Ez könnyen belátható, ha belegondolunk, hogy az RTL modellekben minden változás, minden logikai alacsony szintről magasba és magas szintről alacsonyra történő átváltás tulajdonképpen a modellezett áramkör csomópontjain mérhető villamos potenciál megváltozását jelenti. Márpedig egy csomópont potenciálját csak úgy növelhetjük vagy csökkenthetjük, ha a csomópontra többlet töltést viszünk fel, vagy vezetünk el onnan, ami csak véges idő alatt kivitelezhető, legyen szó akármilyen kis mennyiségű töltésről is.

Ebben a fejezetben áttekintjük a digitális áramkörök késleltetéseinek fizikai hátterét, megvizsgáljuk, hogy milyen funkcionális hibák merülhetnek fel a véges jelváltozási sebességekből adódóan, majd bemutatjuk, hogy a tervezés során mi mindent kell megtennünk ezeknek az időzítési hibáknak az elkerülése érdekében.

6.1 A késleltetések fizikai háttere, metastabilitás

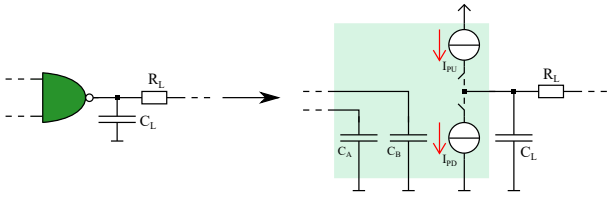
Bármely logikai kapu számára az általa meghajtott többi logikai kapu és a kimenetére kapcsolt összes vezeték RC terhelést jelent. A vezetékszakaszok esetén a rezisztív összetevő annak köszönhető, hogy bár a huzalozást tipikusan jól vezető anyagokból alakítjuk ki, ezek ellenállása nyilván nem nulla, ráadásul keresztmetszetük igen kicsi. A kapacitív összetevő

azért van jelen, mert az egyes vezetékszakaszok a közelükben futó más vezetékszakaszokkal, sőt a hordozóban megvalósított, táp- vagy földpotenciálra kapcsolt zsebekkel együtt is kapacitív struktúrákat képeznek. A meghajtott kapuk bemenetei – CMOS kapcsolástechnikát feltételezve – valójában mind egy-egy MOS tranzisztor gate-struktúrájához kapcsolódnak, melyek ugyancsak kapacitív szerkezetűek. A logikai kapuk hálózatának e parazita elemeit szemlélteti a 6-1. ábra.



6-1. ábra. *Parazita kapacitások és ellenállások egy kombinációs hálózatban*

A CMOS kapcsolástechnikával megvalósított logikai kapuk ki- és bemeneteinek véges jelváltozási sebessége a 6-2. ábrán látható modell alapján érthető meg.



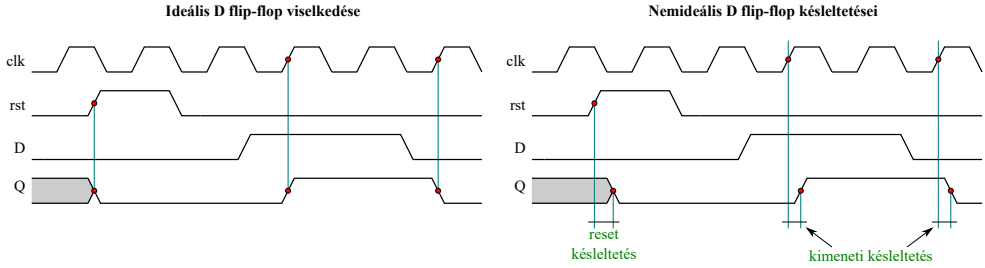
6-2. ábra. *CMOS logikai kapuk egyszerűsített modellje*

Amikor a kapu bemenetein lévő jelszintekből adódóan a kimenetnek alacsony szintről magas szintre kell váltania, akkor a kapu felhúzó hálózatát reprezentáló áramgenerátor[54] a kimenetre kapcsolódva elkezdi feltölteni a C_L kapacitást, miközben a lehúzó hálózat áramgenerátora és a kimenet közötti kapcsolat megszakad. Ha a kimenetnek magas szintről alacsonyra kell váltania, akkor a felhúzó hálózatot leválasztjuk a kimenetről, a lehúzó hálózat áramgenerátora pedig kihúzza a töltést a kimenetet terhelő kapacitásból. Ez az egyszerűsített modell is jól mutatja, hogy végtelen kapcsolási sebesség, vagyis nulla késleltetés nem érhető el, hiszen ez azt jelentené, hogy a felhúzó és lehúzó hálózatokat képviselő áramgenerátoroknak végtelen árammal kellene tölteni és kisütni a C_L kimeneti kapacitást, ami fizikailag persze nem megvalósítható. A töltő/kisütő áramokat elsősorban a felhúzó és

[54] A CMOS kapuk felhúzó és lehúzó hálózatát itt egyszerű áramgenerátorokkal modelleztük. Ha ezt a modellt használva számszerűen is levezetnénk egy konkrét kapu kimenetének jelváltozási sebességét, akkor a valóságosnál nagyobb értéket kapnánk. Valójában a felhúzásért és lehúzásért MOS tranzisztorok felelnek, amelyek kimeneti karakterisztikája nem teljesen egyezik meg egy áramgenerátor karakterisztikájával. Ahogy csökken a drain és a source kivezetések közötti feszültség, úgy kerülünk egyre közelebb a tranzisztor karakterisztájának trióda-szakaszához. Ha a drain–source feszültség már kisebb, mint a tranzisztor küszöbfeszültsége (a tranzisztor kimeneti karakterisztikájának trióda-szakaszán), akkor a töltő/kihúzó áram semmiképp sem tekinthető már állandónak, a feszültség csökkenésével közel arányosan az áram is csökken.

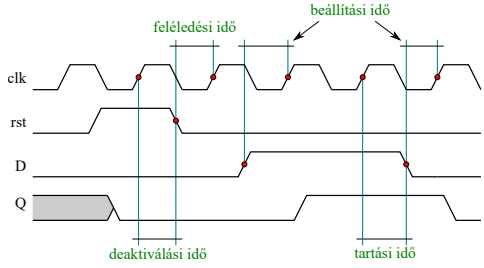
lehúzó hálózatokat alkotó tranzisztorok telítési árama határozza meg, ami a kimeneti fokozat tranzisztorainak geometriai méretétől, küszöbfeszültségétől és technológiai paramétereitől (pl. a gate-oxid felületegységre eső kapacitásától) függ. Ezek alapján belátható tehát, hogy egy logikai kapu bemenetének megváltozása csak késleltetve jelenhet meg a kimeneten. A terjedési késleltetés *(propagation delay)* függ a logikai kapu meghajtóképességétől, a bemenet jelváltozási sebességétől és a kimenet terhelésétől is.

Hasonló áramköri szintű felépítésük miatt késleltetésük természetesen nemcsak a logikai kapuknak, hanem a flip-flopoknak is van. A 6-3. ábra egy aszinkron resettel ellátott D flip-flop reset késleltetésének *(reset delay)* és kimeneti késleltetésének *(clock-to-output delay*[55]*)* definícióját szemlélteti.



6-3. ábra. *A D flip-flopok késleltetései*

A D flip-flopok kimenetein tapasztalható késleltetésekkel nem ér véget az időzítéshez kapcsolódó nemidealitások sora; a flip-flopok bemeneteire kapcsolt jelek időzítésének meg kell felelniük bizonyos feltételeknek ahhoz, hogy a flip-flopok helyesen működjenek. Ha az áramkörön belül akár csak egy olyan flip-flop is van, amelyik esetén ezek a feltételek a logikai kapuk, vezetékszakaszok és a többi flip-flop korábban bemutatott késleltetései miatt sérülnek, akkor az egész áramkört hibásnak kell tekintenünk. A 6-4. ábra ezeket az időzítési feltételeket szemlélteti.



6-4. ábra. *A nemideális D flip-flop bemeneteire vonatkozó időzítési feltételek*

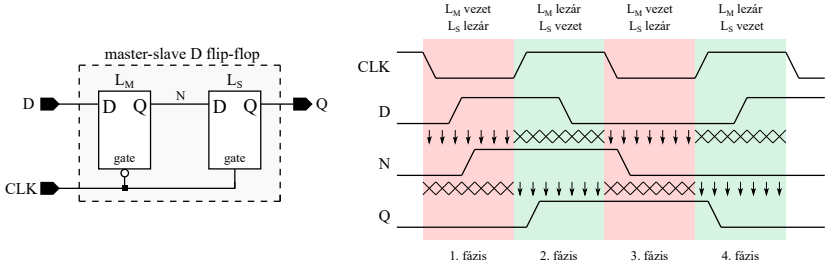
A nemideális D flip-flopok használatakor a következő feltételeknek kell eleget tennünk:

[55] A D flip-flopok kimeneti késleltetésére bizonyos források a logikai kapuknál megismert *„propagation delay”* kifejezést használják.

- **Beállítási idő** *(setup-time)*: Az órajel felfutó éle előtt legalább ennyi idővel a D bemenetnek már stabilnak kell lennie.
- **Tartási idő** *(hold-time)*: Az órajel felfutó éle után legalább ennyi időnek kell eltelnie, mielőtt újabb változás jelenik meg a D bemeneten.
- **Feléledési idő** *(reset recovery time)*: Az órajel felfutó éle előtt legalább ennyi idővel az aszinkron reset jelet deaktiválnunk kell ahhoz, hogy a flip-flop hibátlanul feléledjen reset állapotból. Ez tulajdonképpen a reset bemenet deaktiválására értelmezett beállítási idő.
- **Deaktiválási idő** *(reset removal time)*: Az órajel felfutó éle után legalább ennyi időnek kell eltelnie, mielőtt az aszinkron reset jelet deaktiváljuk. Ez tulajdonképpen a reset bemenet deaktiválására vonatkozó tartási idő.

A D flip-flopok beállítási és tartási időre vonatkozó időzítési feltételeinek hátterét egy CMOS kapcsolástechnikán megvalósított, egyszerűsített, reset/preset és órajel-engedélyező bemenetekkel nem rendelkező master-slave D flip-flop példáján mutatjuk be.

A master-slave D flip-flop funkcionálisan két, ellenütemben engedélyezett D latch-ből épül fel. A 6-5. ábra azt mutatja be, hogy e két latch együttműködése hogyan valósítja meg az élvezérelt D flip-flopra jellemző viselkedést.

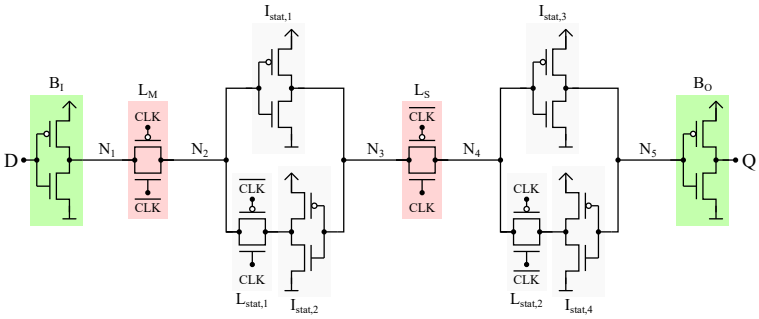


6-5. ábra. *Két latch-ből felépülő, master-slave D flip-flop működése*

Amikor az órajelbemenet alacsony szintű, akkor az L_M master latch ún. transzparens állapotban van, tehát D bemenete és Q kimenete között vezet, míg az L_S slave latch lezár, vagyis nem vezet. Az N jelű belső csomópont tehát követi a D bemenet változásait. Az órajel felfutó élénél a master latch lezár, a slave latch pedig transzparens állapotba vált át, tehát a belső csomópont értéke – ami tulajdonképpen nem más, mint a D bemenet értéke az órajel felfutó élének pillanatában – a Q kimenetre kerül.

A beállítási és tartási idők magyarázatához a master-slave D flip-flop áramköri szintű megvalósítását kell megvizsgálnunk, amit a 6-6. ábra mutat be.

Az ábrán a D flip-flopra jellemző működést megvalósító master és slave latch-funkciókat az L_M és L_S jelzésű transmission-gate kapcsolók valósítják meg. Az $I_{stat,1}$, $I_{stat,2}$, $I_{stat,3}$, és $I_{stat,4}$ jelzésű, keresztbe csatolt inverterek biztosítják a statikus működést, vagyis azt, hogy a flip-flop tápfeszültség jelenléte esetén bármennyi ideig képes fenntartani belső



6-6. ábra. *Egy CMOS master-slave D flip-flop áramköri szintű modellje*

csomópontjainak és kimenetének potenciálját.[56] Az általuk létrehozott, erős pozitív visszacsatolás elősegíti a belső csomópontok jelszintjeinek gyors stabilizálását az órajel felfutó és lefutó éleinél bekövetkező átkapcsolások pillanatában. E pozitív visszacsatolásokat az $L_{stat,1}$ és $L_{stat,2}$ jelzésű transmission-gate kapcsolók aktiválják. E kapcsolók lezárt állapota esetén ezek a kiegészítő fokozatok egy egyszerű invertálást valósítanak meg. A B_I és B_O jelű inverter fokozatok zajszűrő funkciót látnak el, amely a működés robusztusságát hivatott növelni.

Tekintsük át először azt az esetet, amikor a D bemeneten az időzítési kritériumok nem sérülnek.

Ha a CLK bemenet alacsony szintű, akkor

- az L_M latch transzparens állapotú, $L_{stat,1}$ leválaszt, L_S leválaszt, $L_{stat,2}$ transzparens;
- az N_1 és N_2 csomópontokon a D bemenet negált, az N_3 csomóponton pedig a D bemenet ponált értéke jelenik meg;
- az N_4 és N_5 csomópontok potenciálját az $L_{stat,2}$ latch által megvalósított pozitív visszacsatolás az előző ciklusban beállt stabil állapotban stabilizálja, a Q kimenet pedig ennek a stabil állapotnak a negáltja;
- az N_3 csomópont potenciálja követi a D bemenet változásait két inverter és az L_M kapcsoló késleltetésével.

Amikor a CLK bemenet magas szintre vált, akkor

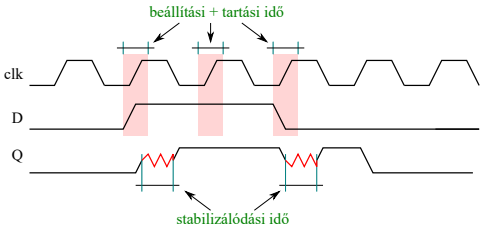
- az L_M latch leválaszt, $L_{stat,1}$ transzparens, az L_S latch transzparens, $L_{stat,2}$ leválaszt;
- az N_2 és N_3 csomópontok potenciálját az $L_{stat,2}$ latch által létrehozott pozitív visszacsatolás stabilizálja. Az N_2 csomópont a D bemenet negált, az N_3 pedig annak ponált értékét tartja;

[56] A flip-flop alapfunkciójának megvalósításához a két transmission-gate elegendő lenne, de ekkor a két kapcsolóelem és a kimenet potenciálját tulajdonképpen csak a vezetékek és egyéb parazita kapacitásokban tárolt töltésmennyiség tartaná fenn. A félvezető struktúrák szivárgási árama miatt ez a töltésmennyiség hamar lecsökken, tehát az eszköz csak a bemenetnek egy bizonyos kapcsolási aktivitása felett lenne képes működni. Ha a flip-flop bemenete sokáig nem változna meg, akkor az eszköz „elfelejtené" korábbi állapotát, tehát nem működne mint tároló elem. A szivárgási áram persze a visszacsatoló inverterek jelenléte esetén is csökkenti a belső csomópontok potenciálját, de így azt a tápforrásból folyamatosan visszapótoljuk. Ezek az inverterfokozatok tehát a statikus működésen felül a statikus fogyasztást is „biztosítják".

- az N_4 csomópont felveszi az N_3 csomópont, vagyis a D bemenet értékét, az N_5 csomópont az N_4 csomópont negáltja lesz;
- a Q kimenet az N_3 csomópont, tehát a D bemenet értékét veszi fel két inverter és az L_S kapcsoló késleltetése után.

A D flip-flopban lévő, transmission-gate kapcsolókkal megvalósított latch-ek tehát az órajel élei hatására transzparens üzemmódból lezárt üzemmódba kapcsolnak át. Az órajel felfutó élénél tehát van egy rövid időtartam, ami alatt az összes transmission-gate kapcsoló a transzparens és a lezárt üzemmód között van. Ez azt jelenti, hogy az órajel felfutó élénél egy rövid pillanatig, amíg az összes transmission-gate kapcsoló teljesen át nem kapcsol transzparens vagy lezárt állapotba, addig a D bemenetnek minden változása éreztetheti a hatását, és a belső csomópontok között töltésátrendeződést idézhet elő. Ebben a pillanatban bármely csomópontra igaz, hogy vagy tápfeszültségen vagy földpotenciálon van, a D bemenet által előidézett töltésátrendeződés tehát minden csomópontra nézve azt jelenti, hogy a potenciál a komparálási szint felé mozdul el. A probléma abból adódik, hogy a gyors stabilizálást segítő, keresztbecsatolt invertereknek létezik egy instabil munkapontja, amelyben a bemenetük és a kimenetük egyaránt a komparálási szinten van. Ez a munkapont instabil, a környezet zaja hatására a keresztbe csatolt inverterek által megvalósított pozitív visszacsatolás hamar visszaáll az egyik stabil munkapontba (stabilizálódási idő, *resolution time*), ahol a Q kimenet értéke egyértelműen tápfeszültség vagy földpotenciál, amíg azonban ez nem következik be, addig a kimenet értéke is a komparálási szint körül ingadozik. Ezt az ideiglenes állapotot a D flip-flop metastabil állapotának nevezzük.

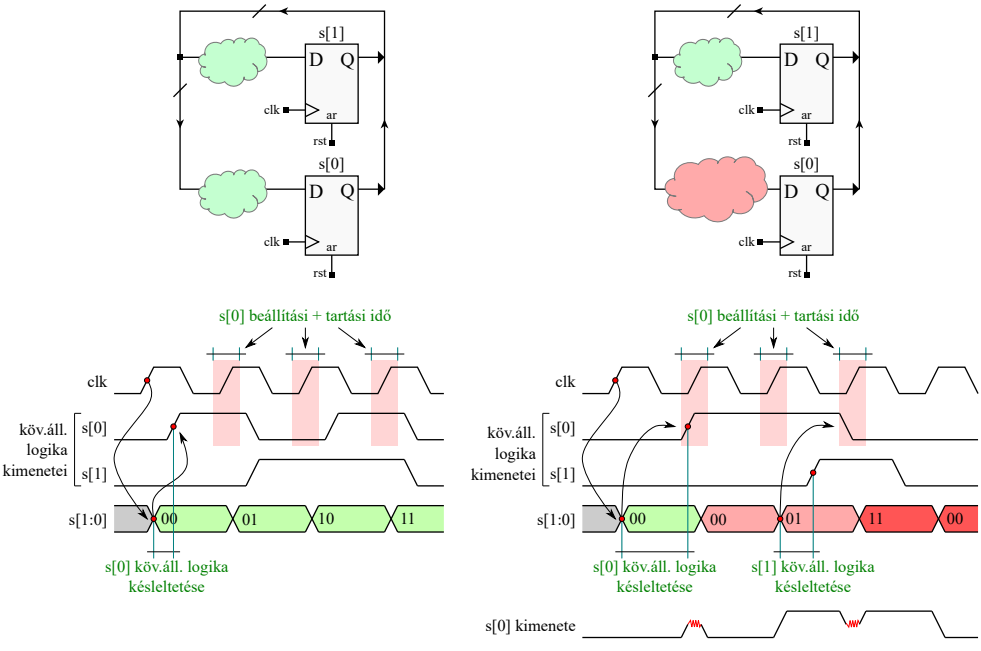
Ha tehát a D flip-flop bemenetére vonatkozó időzítési feltételek nem teljesülnek, vagyis az órajel felfutó élének környezetében megváltozik a D bemenet potenciálja, akkor a flip-flop bizonyos valószínűséggel metastabil állapotba kerül (lásd 6-7. ábra).



6-7. ábra. *A D flip-flop kimenetének viselkedése metastabil állapot esetén*

Metastabil állapotban a flip-flop Q kimenetének potenciálja sem logikai alacsony sem magas szintnek nem tekinthető. A metastabil állapotban lévő flip-flop által meghajtott logikai kapuk alacsony és magas értékként egyaránt értelmezhetik a komparálási szint körül ingadozó kimenetet, ráadásul egymástól teljesen függetlenül, tehát a metastabil kimenet az áramkör egyes részei számára logikai magas, más részei számára pedig logikai alacsony szintként jelenik meg, ami nyilván funkcionális hibát okozhat. Ráadásul a komparálási szint körül mozgó bemenet hatására a CMOS logikai kapukban felléphet az ún. „egymásba vezetés” jelensége is, amikor a felhúzó és lehúzó hálózat egyidejűleg vezet, így az adott

kapu pillanatnyi áramfelvétele drasztikusan megnövekedik. Ha ez a helyzet huzamosabb ideig fennáll, az akár az eszköz tönkremeneteléhez is vezethet. A D flip-flop a metastabil állapotból a környezetének zaja hatására egy idő után visszatér stabil állapotba, azonban a metastabil állapotban töltött időt nem ismerhetjük, csak azt tudhatjuk, hogy a metastabil állapotban maradás valószínűsége exponenciálisan csökken az idővel. Ha a metastabilitás az órajel periódusidejénél tovább áll fenn, akkor a következő flip-flop is könnyen metastabil állapotba kerülhet és így tovább. A másik nagy gond a metastabil állapotba kerülő flip-flopokkal az, hogy a bemenet értékétől teljesen független, hogy a metastabil állapotból kilépve melyik logikai szinten stabilizálódik a kimenetük. Ezt láthatjuk a 6-7. ábra hullámformáján is; az első esetben a D bemenet megváltozása hatására a flip-flop metastabil állapotba kerül, de a kimenet még az órajel következő felfutó éle előtt stabilizálódik, ráadásul úgy, hogy a beálló érték a D bemenet új értéke, tehát az időzítési feltételek megsértése ellenére a flip-flopnak valójában sikerült bemintavételeznie a bemenetet. A második esetben a kimenet ugyancsak stabilizálódik az órajel következő felfutó éle előtt, azonban a beálló stabil kimeneti jelszint ezúttal nem a bemenet új értékével azonos, a flip-flop Q kimenete tehát csak egy órajelciklus múlva fogja sikeresen bemintavételezni az immár stabil D bementet. Ez az egy órajelciklusnyi késés elsőre talán nem tűnik olyan nagy problémának, de ha belegondolunk, hogy egy digitális áramkörben egyáltalán nem mindegy, hogy egy adat melyik órajelciklusban jelenik meg egy adott regiszterben (valójában erről szól az RTL tervező munkájának nagy része), belátható, hogy ez funkcionális hibának tekintendő. Ez akkor válik leginkább szembetűnővé, ha a kérdéses flip-flop egy állapotgép állapotregiszterének



6-8. ábra. *Metastabil állapot egy állapotregiszterben*

egy bitjét képviseli. Ha a következő állapotot meghatározó logika késleltetéséből adódóan egy állapotregiszter valamelyik flip-flopja metastabil állapotba kerül és a 6-7. ábra második feléhez hasonlóan csak egy órajelciklus késéssel képes lekövetni a következő állapotot meghatározó logika kimenetét, az valójában azt jelenti, hogy az állapotgép egy teljesen más állapotsorozatot járhat be, mint ha nem állna fenn időzítési hiba. Erre a jelenségre mutat példát a 6-8. ábra.

Tegyük fel, hogy az állapotregiszter normális esetben a 00 → 01 → 10 →11 → 00 állapotsorozatot kell, hogy bejárja. A 6-8. ábrán a bal oldali esetben a következő állapotot meghatározó logikák késleltetése kicsi, tehát az állapotregiszter megváltozása után kimenetüket gyorsan előállítják, így az órajel következő felfutó élénél az s[0] és s[1] flip-flopok hibátlanul tudják azt mintavételezni. A jobb oldali esetben azt látjuk, hogy az s[0] állapotregiszter következő állapotát meghatározó logika olyan nagy, hogy az állapot beállása után kimenetének értékét csak jelentős késleltetéssel tudja előállítani. A késleltetése olyan nagy, hogy az s[0] flip-flop D bemenetén sérül a beállítási időre vonatkozó kritérium. Az órajel második felfutó élénél az s[0] flip-flop először metastabil állapotba kerül, majd alacsony szinten stabilizálódik, az állapotgép tehát marad a 00 állapotban. Az órajel harmadik felfutó élénél az s[0] regiszter következő állapotát meghatározó logika bemenete és ezáltal kimenete is már stabil, így az s[0] flip-flop gond nélkül mintavételezi azt, az állapotregiszter tehát felveszi a 01 értéket. Ennek hatására az s[1] regiszter következő állapotát meghatározó logika rövid idő alatt előállítja kimenetét, de az s[0]-ra vonatkozó érték ismét csak jelentős késleltetéssel jelenik meg. Az órajel negyedik felfutó élénél az s[1] regiszter helyesen felveszi a magas értéket, az s[0] regiszter viszont megint metastabil állapotba kerül, majd – mivel ezúttal sem sikerül a mintavételezés – értéke logikai magas szinten stabilizálódik. Az állapotgép tehát az 11 állapotba kerül, amelynek hatására a következő állapotot meghatározó logika a 00 értéket állítja elő a kimenetein, ami az s[0] regiszter szempontjából nem jelent semmi változást, tehát a következő állapot, amit az állapotgép felvesz, valóban a 00 lesz. Látható tehát, hogy az s[0] flip-flop beállítási időre vonatkozó időzítési kritériuma a következő állapotot meghatározó logika túlságosan nagy késleltetése miatt sérült, ami teljesen hibás működéshez vezetett. Ebben a példában azt feltételeztük, hogy az időzítési kritérium megsértése mindenképpen metastabilitáshoz vezet, továbbá hogy a mintavételezés semmiképp sem sikerül. A valóságban a helyzet még kellemetlenebb, mivel sem a metastabil állapot bekövetkezése, sem a stabilizálódó kimenet logikai szintje nem determinisztikus, az áramkör viselkedése tehát teljesen megjósolhatatlan lesz, a hiba pedig gyakorlatilag reprodukálhatatlan.

RTL tervezőként sem hagyhatjuk teljesen figyelmen kívül a későbbi fizikai megvalósításból eredő problémákat, bár az RTL modellezési módszer önmagában semmilyen eszközt nem ad a kezünkbe, amivel számszerűen vizsgálhatnánk az időzítési kérdéseket. A 6-8. ábrán látható példánál az okozza a problémát, hogy egy bizonyos kombinációs hálózat késleltetése túl nagy, de tulajdonképpen úgy is fogalmazhatnánk, hogy az órajel élei túl sűrűn követik egymást. Ha az órajel periódusidejét megnöveljük, időzítési problémáink egycsapásra megszűnnek, és pontosan ez az, amiből már az RTL tervezés során is kiindulhatunk. Ha két regiszter között a jelnek sok-sok logikai kapun kell keresztülverekednie magát, akkor nagy lesz a két regiszter közötti jelterjedési idő, ami azt jelenti, hogy a leendő áramkörünkre kapcsolható maximális órajel-frekvencia alacsony lesz. A 4. fejezet bevezetőjében

bemutatott, az RTL tervezés lényegét összefoglaló példaáramkörök, a különböző akkumulátor struktúrák esetén is ezt az elvet alkalmaztuk; ha sok műveletet akarunk elvégezni egyetlen órajelciklus alatt (pl. össze akarunk adni négy számot), akkor a kiadódó kombinációs hálózat nagy késleltetésű lesz, ami csökkenti a maximális órajel-frekvenciát. A megoldás a nagyméretű kombinációs hálózat pipeline regiszterekkel való szétbontása volt.

A regiszterek közötti kombinációs hálózatok méretének becslése regiszter-transzfer szinten nem egyszerű feladat, hiszen itt még nem látjuk pontosan a HDL kódunkból szintetizált logikai kapukat. A részletes, kapuszintű struktúra csak a szintézis során alakul ki, a késleltetéseket ugyancsak igen nagymértékben befolyásoló vezetékezés pedig csak az elhelyezés és huzalozás során. Ekkor az időzítések kvantitatív vizsgálatára már használhatunk statikus időzítésanalízis szoftvereket, de ha már az RTL tervezés fázisában tudni szeretnénk, hogy egy adott HDL modellrészlet hozzávetőlegesen mekkora kombinációs hálózatokat fog eredményezni, akkor kénytelenek vagyunk az 1.2. pontban is említett előzetes szintézis segítségével kicsit „előredolgozni". Az előzetes szintézis lényegében azt jelenti, hogy egy kis részfunkcióra, egy kisebb tervezési egységre vonatkoztatva a teljes szintézis folyamatot végrehajtjuk annak érdekében, hogy megvizsgálhassuk a HDL modellünkből szintetizált áramköri szerkezetet kapuszinten. Az előzetes szintézis eredményeként kapott kapuszintű modell nem feltétlenül lesz teljesen azonos azzal a struktúrával, amit a toplevel integráció utáni, végleges szintézis során kapunk. A kisebb részfunkciót megvalósító pontos áramköri szerkezetet ti. befolyásolhatja az azt körülvevő többi áramkör a logikai optimalizáció és – még inkább – az elhelyezés és huzalozás fázisában. Mindenesetre az előzetes szintézis eredményesen használható az RTL modellünk belső, potenciálisan kritikus méretű/késleltetésű kombinációs hálózatainak felderítéséhez.

Az előzetes szintézis elsődleges bemenete értelemszerűen a HDL modell, ezenkívül azonban az RTL tervezőnek a szintézis előtt meg kell fogalmaznia azt is, hogy a HDL modellből előállított áramkörnek milyen időzítési peremfeltételeknek *(timing constraints)* kell megfelelnie. A szintézis szoftverek figyelembe veszik ezeket az időzítési előírásokat a kapuszintű modell előállítása és optimalizációja során.

6.2 Időzítési kényszerek

A 6-8. ábra példájában azt láttuk, hogy a szintetizált áramkörben egy kombinációs hálós zat késleltetése adott órajel-frekvencia esetén túl nagy volt, ami funkcionálisan teljesen rossz működéshez vezetett. Az RTL modell funkcionális helyessége tehát csak az első nagy mérföldkő a működő áramkörhöz vezető úton. A másik, hasonlóan fontos mérföldkő az, amikor az RTL modellből az adott céltechnológiára végzett szintézis az időzítési kényszereknek, peremfeltételeknek *(timing constraint)* is minden szempontból megfelel *(timing closure)*. Az időzítési peremfeltételek olyan elvárások, amelyeket az RTL tervező fogalmaz meg, és amelyek – a HDL modell mellett – az automatizált áramkörszintézis bemenetéül szolgálnak. Az időzítési kényszereket külső tényezők rögzítésével fejezzük ki, mint például az órajel előírt periódusideje, de ezek közvetetten az áramkör belső késleltetéseire vonatkozóan fogalmaznak meg elvárásokat.

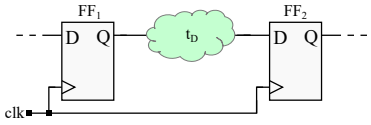
Az adott szintézis-szoftverrendszerre jellemző parancsfájlok segítségével formalizált peremfeltételeket a logikai és fizikai szintézis szoftverek egyaránt figyelembe veszik, majd – mivel a feltételek teljesülése ennek ellenére nem biztosított – a fizikai terv elkészülte után, a statikus időzítésvizsgálat során azok teljesülését újra ellenőrizzük. Ebben az alfejezetben az áramkörök fizikai szerkezetéből levezetjük, hogy RTL tervezőként pontosan milyen peremfeltételeket kell rögzítenünk, majd bemutatjuk azt is, hogy ezt hogyan tehetjük meg formálisan, a tervezőrendszereink számára is feldolgozható formában.

Az időzítési viszonyok vizsgálatához az áramköreink belső szerkezetét leíró modelleket használunk. Mint minden modellre, ezekre is igaz, hogy pontosságuk, valósághűségük korlátozott. Minél pontosabb modellt használunk, minél több lehetséges késleltetés-összetevőt veszünk figyelembe, annál kisebb lesz az esélye, hogy a megvalósított áramkörben időzítési problémák merülnek fel. Az alábbi fejezetben először a lehető legegyszerűbb modellből indulunk ki, amit azután több lépésben finomítunk. Célunk, hogy a bevezetett fogalmak és számítási módszer segítségével az Olvasó maga is létre tudjon hozni az itt bemutatottaknál lényegesen részletesebb és összetettebb modelleket az időzítési viszonyok vizsgálatára.

6.2.1 Az időzítési kényszerek specifikálása

6.2.1.1 Órajelfeltétel *(clock constraint)*

A digitális áramkörökre érdemes egyszerűen csak flip-flopoknak és logikai kapuknak a 4-1. ábrán láthatóhoz hasonló hálózataként gondolni, ha nem a funkciót, hanem pusztán csak az időzítési viszonyokat vizsgáljuk. Az időzítési peremfeltételek fogalmának megértéséhez a 6-9. ábra alapján vizsgáljuk meg, hogy mi is történik egy digitális áramkörben akkor, amikor az órajelen megjelenik egy felfutó él.



6-9. ábra. *Egy lokális adatút felépítése*

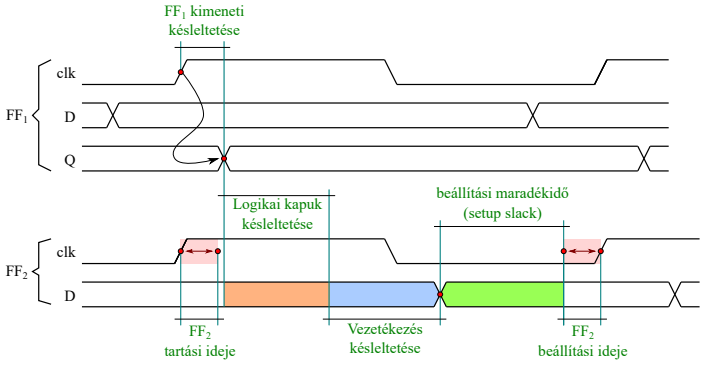
Ez a struktúra egy ún. lokális adatút *(local datapath)*, ami nem más, mint két, egyazon órajelről járó flip-flop, amelyek valamilyen kombinációs hálózaton keresztül kapcsolódnak egymáshoz. A kibocsátó *(launch)* flip-flop (FF_1) Q kimenete az órajel n-edik felfutó éle hatására megváltozik, és ez a változás a kombinációs hálózaton végighaladva megjelenik a fogadó *(capture)* flip-flop (FF_2) D bemenetén. Ezt a megváltozott bemenetet a fogadó flip-flop az órajel n+1-edik felfutó éle hatására mintavételezi és eltárolja. Ezek alapján már megfogalmazhatjuk a valódi, késleltetésekkel bíró logikai kapukból és flip-flopokból felépülő áramkörre vonatkozó legalapvetőbb időzítési peremfeltételt:

„Az áramkörben található bármely lokális adatútban a kibocsátó és fogadó flip-flop közötti kombinációs hálózat késleltetése nem lehet nagyobb az órajelperiódusnál.”

$$t_D < T_{clk}$$

ahol t_D a két flip-flop közötti jelterjedési idő, T_{clk} pedig az órajel periódusideje. Ez az általánosan megfogalmazott időzítési peremfeltétel a gyakorlatban úgy nyer számszerű formát, hogy egy makrocellával kapcsolatban az alkalmazandó órajel frekvenciája általában a specifikáció része. Ha például a fenti adatút órajelére 100 MHz-es frekvencia van előírva, az azt jelenti, hogy az adatút késleltetése 10 ns-nál kisebb kell, hogy legyen. Ha újrafelhasználható makrocellát tervezünk, akkor előfordulhat, hogy nincs konkrét specifikáció az órajel frekvenciájára vonatkozóan, ekkor arra törekszünk, hogy a leendő áramkörben a legnagyobb késleltetésű lokális adatút, más néven a kritikus út minél kisebb késleltetésű legyen.

Ezzel azonban még csak az időzítési kérdések alapproblémájára világítottunk rá, a helyzet a valóságban kissé árnyaltabb. A 6-10. ábra a két flip-flop közötti adatátvitel folyamatát szemlélteti a legalapvetőbb késleltetési összetevők figyelembevételével.



6-10. ábra. *Egy lokális adatút késleltetésének legfontosabb összetevői*

Az időzítési diagram a következő késleltetés-komponenseket szemlélteti:

- FF_1 kimeneti késleltetése: Az órajel felfutó éle és az FF_1 flip-flop Q kimenetének megváltozása között eltelt idő. Ez a késleltetés függ a flip-flop fizikai tulajdonságaitól és a terheléstől.
- FF_1 – FF_2 lokális adatút logikai kapuinak együttes késleltetése (*logic delay* vagy *cell delay*).
- FF_1 – FF_2 lokális adatút vezetékeinek együttes késleltetése (*interconnect delay* vagy *net delay*).

Beállítási maradékidő *(setup slack)* • Egy lokális adatút akkor működik megfelelően, ha az órajel n-edik felfutó éle hatására a kibocsátó flip-flop Q kimenetén megjelenő változás kellően gyorsan jut el a fogadó flip-flop D bemenetére ahhoz, hogy az órajel n+1-edik

felfutó éle hatására a mintavételezés hibátlanul megtörténhessen. A helyes mintavételezés feltétele, hogy a fogadó flip-flop D bemenete az n+1-edik órajel-felfutóél előtt már stabil legyen. Hogy pontosan mennyivel kell az órajel felfutó éle előtt már stabilnak lennie a jelnek, azt a fogadó flip-flop beállítási ideje határozza meg. Ezek alapján tehát a lokális adatút késleltetésére vonatkozó peremfeltételt a következőképpen kell pontosítanunk:

$$t_D < T_{clk} - T_{C2O,FF1} - T_{setup,FF2}$$

ahol $T_{C2O,FF1}$ a kibocsátó flip-flop kimeneti késleltetése, $T_{setup,FF2}$ pedig a fogadó flip-flop beállítási ideje. A 6-10. ábrán azt látjuk, hogy a jel a fogadó flip-flop bemenetén jóval előbb stabilizálódik, mint az szükséges lenne, tehát az adatútban időzítési szempontból van még némi tartalék. Ezt a tartalék időtartamot a lokális adatút beállítási maradékidejének *(setup slack)* nevezzük. A teljes áramkör időzítési szempontból akkor tekinthető hibátlannak, ha ez a beállítási maradékidő minden egyes lokális adatútra vonatkoztatva nullánál nagyobb. A beállítási maradékidő a különböző késleltetési összetevőkkel a következőképpen fejezhető ki:

$$setup\ slack = T_{clk} - t_D - T_{C2O,FF1} - T_{setup,FF2}$$

Ha a beállítási maradékidő negatív értékű, az azt jelenti, hogy a fogadó flip-flop D bemenete a beállítási idő alatt változik, a flip-flop tehát metastabil állapotba kerülhet *(setup-time violation)*. Tegyük fel, hogy áramkörünket egy Microsemi RTAX2000S típusú FPGA-ban akarjuk megvalósítani, amelynek belső flip-flopjainak kimeneti késleltetése 1,1 ns, beállítási ideje pedig 0,25 ns. Ez azt jelenti, hogy ha az órajel frekvenciájára továbbra is a 100 MHz-es előírás érvényes, akkor a lokális adatút késleltetése legfeljebb 8,65 ns lehet. Ha feltételezzük, hogy a lokális adatút vezetékeinek és logikai kapuinak együttes késleltetése 6,7 ns, akkor a beállítási maradékidő 1,95 ns.

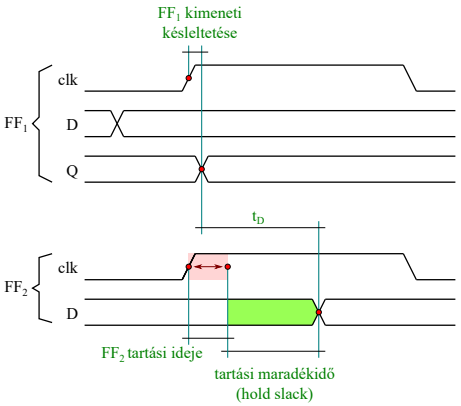
Tartási maradékidő *(hold slack)* • A kibocsátó és fogadó flip-flop közötti adatút késleltetése tehát egy adott órajel-frekvencia-előírás mellett nem lehet egy bizonyos értéknél nagyobb, ez azonban csak az érem egyik oldala. A 6-11. ábra egy olyan helyzetet mutat be, amelynél a kibocsátó flip-flop kimeneti késleltetése és a lokális adatút vezetékeinek és logikai kapuinak együttes késleltetése is viszonylag kicsi.

Ebben az esetben a probléma az órajelnek nem az n+1-edik, hanem már az n-edik felfutó élénél jelentkezik. Ha az adatút késleltetése csekély, akkor fennáll a veszélye annak, hogy az n-edik felfutó élnél a fogadó flip-flop tartási időre vonatkozó időzítési kritériuma sérül. Megfogalmazható tehát egy minimális késleltetési feltétel is a lokális adatútra vonatkozóan:

$$t_D > T_{hold,FF2} - T_{C2O,FF1}$$

ahol $T_{hold,FF2}$ a fogadó flip-flop tartási ideje. Definiálható továbbá az ún. tartási maradékidő *(hold slack)* a következőképpen:

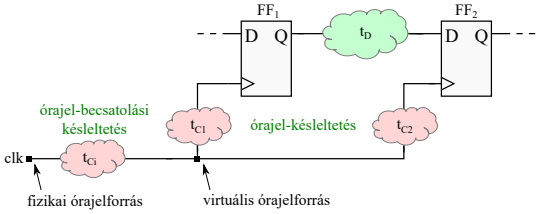
$$hold\ slack = (T_{C2O,FF1} + t_D) - T_{hold,FF2}$$



6-11. ábra. *A tartási maradékidő*

A tartási maradékidőnek a beállítási maradékidőhöz hasonlóan minden esetben nullánál nagyobbnak kell lennie, máskülönben a fogadó flip-flop a tartási időre vonatkozó kritérium sérülése miatt metastabil állapotba kerülhet *(hold-time violation)*.

Órajelelcsúszás *(clock skew)* • A lokális adatutak 6-9. ábrán látható modellje bizonyos szempontból idealizált, sok tényezőt nem vesz figyelembe. A 6-12. ábra egy némileg valósághűbb képet fest egy lokális adatút felépítéséről, és ebből adódóan az adatút maximális késleltetésére vonatkozó peremfeltételt is finomítanunk kell.

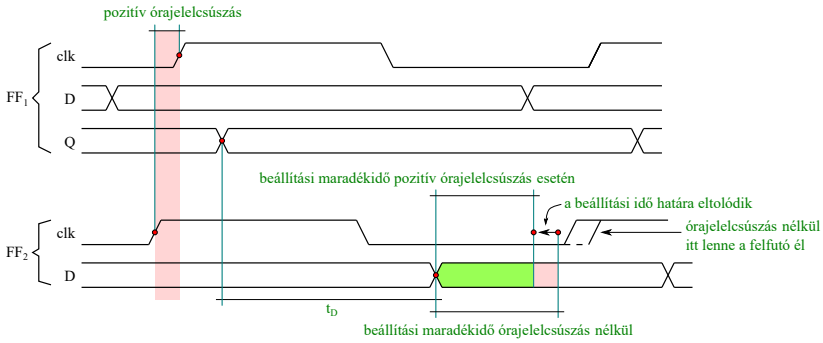


6-12. ábra. *A lokális adatutaknak az órajelhálózat késleltetéseit is figyelembe vevő modellje*

Ez a modell figyelembe veszi azt, hogy az órajel az adatjelekhez hasonlóan vezetékeken és bufferekeren keresztül jut el az egyes flip-flopokhoz, amelyeknek szintén van valamekkora késleltetésük. A fizikai órajelforrás és a virtuális órajelforrás, vagyis a két flip-flophoz vezető órajel-ág közös pontja közötti, ún. órajel-becsatolási késleltetés (t_{Ci}, *clock insertion delay*) a lokális adatút szempontjából nem számít, a virtuális órajelforrás és az egyes flip-flopok órajelbemenete közötti késleltetések (t_{C1}, t_{C2}) viszont annál inkább. Az e két összetevő közötti különbséget órajelelcsúszásnak *(clock skew)* nevezzük. Az órajelelcsúszás értéke pozitív és negatív egyaránt lehet, definíció szerint a következőképpen határozható meg:

$$skew = t_{C1} - t_{C2}$$

ahol t_{C1} a virtuális órajelforrás és a kibocsátó flip-flop órajelbemenete közötti késleltetés, t_{C2} pedig a virtuális órajelforrás és a fogadó flip-flop órajelbemenete közötti késleltetés. A pozitív órajelelcsúszás tehát azt jelenti, hogy a fogadó flip-flop „előbb érzékeli" az órajel éleit, mint a kibocsátó flip-flop, negatív órajelelcsúszás esetén pedig éppen fordítva. Akár pozitív, akár negatív az órajelelcsúszás, időzítési szempontból mindegyik problémát okozhat. A 6-13. ábra azt mutatja be, hogy mit jelent a pozitív órajelelcsúszás az ideális, késleltetések nélküli órajelhálózat esetéhez képest.



6-13. ábra. *A pozitív órajelelcsúszás hatása*

Ha tehát az órajelelcsúszást is figyelembe akarjuk venni az időzítési peremfeltételek megfogalmazásánál, akkor az adatút maximálisan megengedhető késleltetésére vonatkozó kifejezést a következőképpen kell ismét finomítanunk:

$$t_D < T_{clk} - T_{C2O,FF1} - T_{setup,FF2} - skew$$

Ebben az esetben a beállítási maradékidő kiszámításának módja a következő:

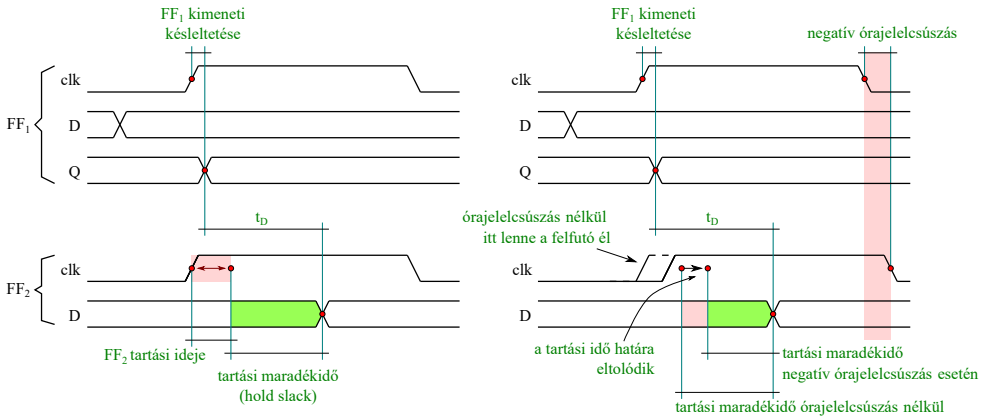
$$setup\ slack = T_{clk} - t_D - T_{C2O,FF1} - T_{setup,FF2} - skew$$

Az előző, számszerűsített példánál maradva adjunk hozzá a lokális adatút-modellünkhöz 1,2 ns órajelelcsúszást. Ekkor az adatút maximálisan megengedhető késleltetése 7,45 ns-ra, a beállítási maradékidő pedig 0,75 ns-ra csökken.

Ezek alapján jogosan vonhatjuk le azt a következtetést, hogy a negatív órajelelcsúszás jótékony hatással van az időzítési viszonyokra, hiszen ha a fogadó flip-flop órajele késik, akkor a jelnek is több ideje van elérni a flip-flop D bemenetét. Ez tulajdonképpen így is van, azonban a negatív órajelelcsúszás kérdését más szemszögből vizsgálva mégiscsak nemkívánatos jelenségnek bizonyul.

Ha nem számolunk órajelelcsúszással, akkor ahhoz, hogy a fogadó flip-flop tartási idejére vonatkozó kritérium sérüljön, a kibocsátó flip-flop kimeneti késleltetésének és az adatút teljes késleltetésének az összege kisebb kell, hogy legyen, mint a fogadó flip-flop tartási ideje –, ami igen valószínűtlen. A mai modern technológiákon megvalósított flip-flopok kimeneti késleltetése már önmagában nagyobb, mint a tartási idők, így tulajdonképpen

a tartási idők kritériumának sérülése még akkor sem fordulhatna elő, ha az adatút késleltetése nulla lenne. A 6-14. ábra jobb oldala azt szemlélteti, hogy miként hat a negatív órajel lelcsúszás a tartási maradékidőre.



6-14. ábra. *A negatív órajelelcsúszás hatása az időzítési viszonyokra*

Ha ezt a hatást is figyelembe akarjuk venni az időzítésre vonatkozó számítások során, akkor az adatút minimális késleltetésére és a tartási maradékidőre vonatkozó összefüggéseket az alábbiak szerint kell pontosítanunk:

$$t_D > T_{hold,FF2} - T_{C2O,FF1} + |skew|$$

$$hold\ slack = (T_{C2O,FF1} + t_D) - T_{hold,FF2} - |skew|$$

Látható tehát, hogy a negatív órajelelcsúszás lehet ugyan jótékony hatással a beállítási maradékidőkre, a tartási maradékidőket ugyanakkor csökkenti, ami szintén időzítési hibákhoz vezethet.

Az eddig elmondottakat összegezve tehát megállapíthatjuk, hogy kétféle időzítési hibatípus léphet fel egy digitális áramkörben:

- Egy bizonyos flip-flop beállítási idejére vonatkozó időzítési kritérium sérül *(setup-time violation)*: Ennek a hibának a gyakorlatban az elsődleges oka jellemzően az, hogy egy lokális adatút két flip-flopja között túl nagy méretű kombinációs hálózat van. Ha tehát az előzetes szintézis részeként elvégzett időzítésvizsgálat során ilyen típusú hibát fedezünk fel, akkor az RTL modellt úgy kell módosítani, hogy ennek a bizonyos kombinációs hálózatnak a mérete kisebb legyen. Alkalmazhatunk például pipeline regisztereket, ahogy azt a 4-4. ábra akkumulátor áramköre esetén tettük, vagy kisebb részfeladatokra bonthatjuk a funkciót, amelyeket már kisebb méretű kombinációs hálózatokkal is kezelni tudunk, erre a megoldásra jó példa a 4.7. pontban bemutatott kaszkád órajel-előosztó. A lényeg, hogy bár az időzítési viszonyok finomhangolása a fizikai tervezés problémaköre, az RTL terv előzetes szintézis alapján

véghezvitt optimalizációjával az ilyen típusú időzítési hibák lehetőségét viszonylag korán kiszűrhetjük. Ha pedig sehogy sem sikerül a hibát kiküszöbölni, akkor – elfogadván, hogy áramkörünk az előírásoknak nem tud maradéktalanul megfelelni – az órajel frekvenciájára vonatkozó követelmény lazításával a probléma egycsapásra megszűnik, tehát egy kicsit lassabb, de funkcionálisan jól működő megoldást kaptunk.

- Egy bizonyos flip-flop tartási idejére vonatkozó időzítési kritérium sérül *(hold-time violation)*: A gyakorlatban az ilyen típusú hibákhoz tipikusan két dolognak egyszerre kell jellemeznie a lokális adatutat; nagyon kis késleltetésű logika a két flip-flop között (jellemzően léptetőregisztereknél fordul elő ilyesmi, ahol a flip-flopok között csak vezeték van, logikai kapuk nincsenek) és negatív órajelelcsúszás. Ha az előzetes szintézis során ilyen hibát tapasztalunk, akkor sajnos az RTL terv módosításával semmit nem érünk el. A túl csekély késleltetésű lokális adatút problémáját legfeljebb az adatút kombinációs hálózatának indokolatlan, szándékos megnövelésével próbálhatjuk megoldani, ami az RTL modell implementációfüggővé tételét jelentené, ráadásul a szintézis során figyelnünk kellene, hogy a felesleges logikát a szoftver ne optimalizálja ki. A negatív órajelelcsúszással regiszter-transzfer szinten ugyancsak nem tudunk kezdeni semmit, hiszen – ahogy a 4.4.4. pontban is állítottuk – az RTL tervezésnek egy nagyon fontos alapvetése, sőt, tulajdonképpen a HDL-ek szimulációs szemantikájának alapvető sajátossága, hogy ideális órajelhálózatot feltételez. Egy másik érdekes tulajdonsága ennek az időzítési problémának, hogy még az órajel előírt frekvenciájának csökkentése sem oldja meg; a lokális adatút késleltetésének minimumára vonatkozó kifejezésben nem szerepel az órajel periódusideje, csak olyan változók, amelyeket vagy a megvalósítás alapjául szolgáló technológia ($T_{hold,FF2}$, $T_{C2O,FF1}$) vagy az órajelhálózat sajátosságai *(skew)* határoznak meg. A tartási időkre vonatkozó kritériumokól adódó időzítési hibák kezelését tehát kénytelenek vagyunk a fizikai tervezés fázisában megoldani, annál is inkább, mert ez a hiba jellemzően akkor fordul elő, amikor egy léptetőregiszter vagy ahhoz hasonló struktúra elhelyezése olyan szerencsétlen módon sikerül, hogy a problémás lokális adatút kibocsátó flip-flopja lényegesen messzebb kerül a virtuális órajelforráshoz, mint a fogadó flip-flop. Az ilyen időzítési hibák tehát tipikusan a fizikai terv hiányosságaiból adódnak, tehát ott is kell megoldani őket, az elhelyezés és huzalozás módosításával. Egyes elhelyező és huzalozó algoritmusok ezeket az időzítési hibákat képesek automatikusan, az adatútba illesztett bufferek segítségével orvosolni, ha a tartási maradékidő nem sokkal kisebb nullánál.[57]

Összefoglalva tehát, egy digitális áramkör minden lokális adatútjára teljesülnie kell a következő időzítési peremfeltételeknek:[58]

[57] A Microsemi Designer nevű fizikai szintézis szoftvere például az Axcelerator FPGA család RT *(Rad-Tolerant)* verziói esetén a -3 ns-ot meg nem haladó tartási maradékidőt automatikusan képes javítani.

[58] A lokális adatutak itt bemutatott időzítési modellje már elég részletesnek tűnik, de az időzítés-analízis szoftverek segítségével további tényezők is figyelembe vehetők, mint például az órajelforrás frekvenciastabilitása és remegése *(jitter)*. Ezek a tényezők az órajelelcsúszásnál bemutatotthoz hasonló módon hatnak az időzítési viszonyokra, ezért ezeket itt részletesen nem tárgyaljuk.

$$t_D < T_{clk} - T_{C2O,FF1} - T_{setup,FF2} - skew$$

$$t_D > T_{hold,FF2} - T_{C2O,FF1} + |skew|$$

ahol

- t_D a lokális adatút kibocsátó flip-flopjának Q kimenete és a fogadó flip-flop D bemenete közötti vezetékek és logikai kapuk együttes késleltetése,
- T_{clk} az előírt órajel-frekvencia,
- $T_{C2O,FF1}$ a kibocsátó flip-flop kimeneti késleltetése *(clock-2-output delay),*
- $T_{setup,FF2}$ a fogadó flip-flop tartási ideje *(setup time),*
- *skew*: az órajelelcsúszás mértéke,
- $T_{hold,FF2}$: a fogadó flip-flop tartási ideje.

Az első feltétel teljesülése esetén biztosított az, hogy a lokális adatút fogadó flip-flopjának beállítási idejére vonatkozó kritérium nem fog sérülni, a második feltétel teljesülése pedig a fogadó flip-flop tartási idejére vonatkozó kritérium teljesülését biztosítja. Figyeljük meg, hogy e paraméterek közül az órajel periódusidejét leszámítva minden más a szintetizált áramkörre jellemző állandó. Ez azt jelenti, hogy az áramkörünk összes lokális adatútjára megfogalmazhatjuk e két időzítési peremfeltételt pusztán az órajel periódusidejének rögzítésével *(clock constraint)*.

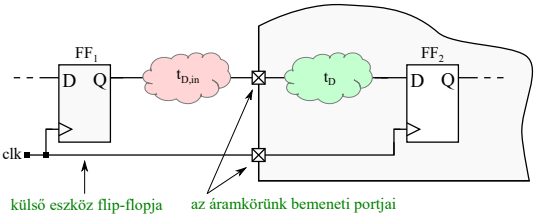
Az órajel periódusidejére vonatkozóan megfogalmazott peremfeltétel segítségével az áramkör összes olyan adatútjára megfogalmazhatjuk a szükséges időzítési követelményeket, amelyek egy belső flip-flop Q kimenetén kezdődnek és egy másik belső flip-flop D bemenetén végződnek. Ez azonban még nem minden. Meg kell vizsgálnunk áramkörünknek azokat az adatútjait is, amelyek egy bemeneti porton kezdődnek és egy belső flip-flop D bemenetén végződnek, valamint azokat, amelyek egy belső flip-flop Q kimenetén kezdődnek és egy kimeneti porton végződnek. Az ezekre az adatutakra vonatkozó időzítési peremfeltételeket rendre bemeneti és kimeneti késleltetésnek nevezzük *(input delay, output delay)*.

6.2.1.2 Bemeneti késleltetés *(input delay constraint)*

A 6-15. ábra a bemeneti késleltetés számszerű vizsgálatához használt adatút-modellt mutatja be[59].

Ez az adatút-modell nagyon hasonló ahhoz, mint amit a belső adatutak vizsgálatához használtunk. A különbség az, hogy ezúttal a kibocsátó flip-flop szerepét egy képzeletbeli, külső eszköz flip-flopja tölti be, és a két flip-flop közötti késleltetésnek kicsit máshogy alakulnak az összetevői. A fogadó flip-flop beállítási és tartási időire vonatkozó kritériumok teljesülését ezúttal a következő feltételek biztosítják:

[59] A bemeneti és kimeneti késleltetések bemutatásánál az áttekinthetőség kedvéért a korábbinál egyszerűbb, ideális órajel-terjesztőhálózatot feltételező modellt használunk. Az órajelhálózat nemideális jellege a korábban látottakhoz hasonlóan vehető figyelembe.



6-15. ábra. *A bemeneti késleltetések számításához használt adatút-modell*

$$t_{D,in} + t_D < T_{clk} - T_{C2O,FF1} - T_{setup,FF2}$$

$$t_{D,in} + t_D > T_{hold,FF2} - T_{C2O,FF1}$$

A belső, lokális adatutak esetéhez hasonlóan most is a t_D késleltetésre vonatkozóan szeretnénk valamilyen feltételt megfogalmazni, hiszen ez az az adatút, amit az órajel periódusidejére vonatkozó követelmény megfogalmazásával még nem fedtünk le. Rendezzük át a fenti összefüggéseket úgy, hogy ez legyen az egyetlen elem a bal oldalon:

$$t_D < T_{clk} - (T_{C2O,FF1} + t_{D,in}) - T_{setup,FF2}$$

$$t_D > T_{hold,FF2} - (T_{C2O,FF1} + t_{D,in})$$

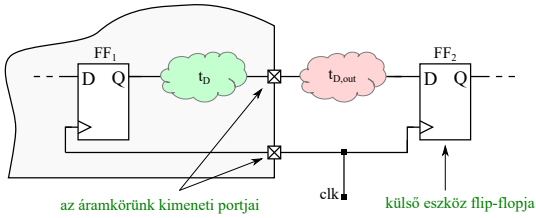
E kifejezések jobb oldala alapján már látható, hogy az órajel periódusidején kívül mit kell még követelményként megadnunk ahhoz, hogy a t_D késleltetés peremfeltételeknek való megfelelése megállapítható legyen; szükségünk van a kibocsátó flip-flop kimeneti késleltetésére, valamint az e flip-flop és az eszközünk bemeneti portja közötti vezeték késleltetésére. E két késleltetés-összetevő összegét nevezzük bemeneti késleltetésnek. Szigorúan véve a bemeneti késleltetésnek két különböző értékére, a maximumára és a minimumára is szükségünk van, amivel ki tudjuk fejezni, hogy ezek a paraméterek függhetnek bizonyos külső körülményektől, például a hőmérséklettől. Ha ezt is figyelembe akarjuk venni, akkor a t_D késleltetésre vonatkozó maximumfeltételben a maximális, a minimumfeltételben pedig a minimális bemeneti késleltetéssel kell számolnunk:

$$input\ delay_{max} = T_{C2O,FF1,max} + t_{D,in,max}$$

$$input\ delay_{min} = T_{C2O,FF1,min} + t_{D,in,min}$$

6.2.1.3 Kimeneti késleltetés *(output delay constraint)*

A kimeneti késleltetés vizsgálatakor hasonlóan járunk el. A 6-16. ábra a kimeneti késleltetés számszerű vizsgálatához használt adatút-modellt mutatja be.



6-16. ábra. *A kimeneti késleltetések számításához használt adatút-modell*

A fogadó flip-flop beállítási és tartási időire vonatkozó kritériumok teljesülését ezúttal a következő feltételek biztosítják:

$$t_D + t_{D,out} < T_{clk} - T_{C2O,FF1} - T_{setup,FF2}$$

$$t_D + t_{D,out} > T_{hold,FF2} - T_{C2O,FF1}$$

Rendezzük át ezúttal is a fenti összefüggéseket úgy, hogy t_D legyen az egyetlen elem a bal oldalon:

$$t_D < T_{clk} - T_{C2O,FF1} - (T_{setup,FF2} + t_{D,out})$$

$$t_D > (T_{hold,FF2} - t_{D,out}) - T_{C2O,FF1}$$

Ezúttal tehát az órajel periódusidején kívül a fogadó flip-flop beállítási és tartási idejét, valamint az eszközünk kimeneti portja és a fogadó flip-flop D bemenete közötti vezeték késleltetését is meg kell határoznunk ahhoz, hogy a t_D adatút késleltetésének peremfeltételeknek való megfeleléséről meggyőződhessünk. Korábban, a bemeneti késleltetés esetén csak azért beszéltünk két különböző, maximális és minimális értékről, mert ki akartuk fejezni azt, hogy az e késleltetést alkotó összetevők függhetnek a külső körülményektől, például a hőmérséklettől. A kimeneti késleltetés esetén kicsit más a helyzet. A fenti, zárójeles kifejezések a kimeneti késleltetés maximális (fent) és minimális (lent) értékének definíciói, amelyek ebben az esetben eltérnek:

$$output\ delay_{max} = T_{setup,FF2} + t_{D,out}$$

$$output\ delay_{min} = T_{hold,FF2} - t_{D,out}$$

Figyeljük meg, hogy a kimeneti késleltetés definíció szerinti minimuma akár negatív is lehet, sőt, mivel a flip-flopok tartási ideje a mai modern technológiákon elég csekély,[60] a gyakorlatban többnyire negatív. Annak érdekében, hogy a kimeneti késleltetés definíciója szemléletesebb legyen, a kifejezést érdemes az alábbi formára hozni:

[60] A korábban is említett Microsemi RTAX eszközcsalád flip-flopjai esetén a tartási idő 0 ns.

$$output\ delay_{min} = -(t_{D,out} - T_{hold,FF2})$$

A fenti kifejezésből látható, hogy nulla tartási időt feltételezve a kimeneti késleltetés minimuma definíció szerint egyenlő az áramkörünk kimenete által meghajtott vezetékezés késleltetésének a mínusz egyszeresével.

Összefoglalva tehát az alábbi időzítési peremfeltételeket fogalmaztuk meg eddig:

- **Órajel-feltétel** *(clock constraint)*: Közvetetten az áramkörünk összes belső lokális adatútjára vonatkozóan meghatároz egy maximális megengedhető terjedési késleltetést a kibocsátó és a fogadó flip-flop között. Ezzel egy adott áramköri struktúra esetén ellenőrizhetővé válik a fogadó flip-flopok beállítási idejére vonatkozó kritérium teljesülése. Az ugyanezen flip-flopok tartási idejére vonatkozó kritérium teljesülésének ellenőrizhetőségéhez nem kell semmilyen időzítési peremfeltételt megadni, ehhez elegendő magának az áramkörnek a belső késleltetéseit ismerni.
- **Bemeneti késleltetés** *(input delay constraint)*: Közvetetten az áramkörünk bemeneti portjai és az azok által meghajtott flip-flopok közötti adatutakra határoz meg maximális és minimális terjedési késleltetést. Maximális értéke a fogadó flip-flopoknak a beállítási, minimális értéke pedig az ezek tartási idejére vonatkozó kritériumok teljesülését teszi ellenőrizhetővé egy adott áramköri struktúra esetén.
- **Kimeneti késleltetés** *(output delay)*: Közvetetten az áramkörünk kimeneti portjai és az azokat meghajtó flip-flopok közötti adatutakra határoz meg maximális és minimális terjedési késleltetést. A maximális érték az eszközünk által meghajtott, fiktív külső flip-flopoknak a beállítási, minimális értéke pedig az ezek tartási idejére vonatkozó kritériumok teljesülését teszi ellenőrizhetővé egy adott áramköri struktúra esetén.

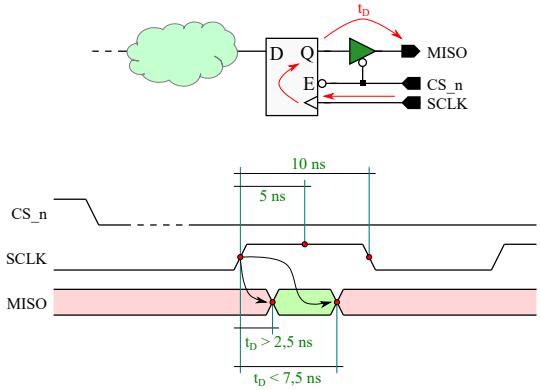
A bemeneti és kimeneti késleltetésekkel kapcsolatban megjegyzendő, hogy bár a vonatkozó adatutak vizsgálatához olyan modelleket használtunk, amelyek egy másik, egyazon órajelről járó külső eszköz és flip-flopjainak jelenlétét feltételezik, a gyakorlatban egyáltalán nem biztos, hogy az áramköri környezet pontosan így néz ki. Ezekre az időzítési peremfeltételekre inkább úgy érdemes gondolni, mint amelyekkel meghatározhatjuk, hogy az eszközünk fizikai interfészén, vagyis a portjain milyen időzítési viszonyokat szeretnénk elérni. A bemeneti késleltetés fogalmát tehát annak leírására használjuk, hogy a bemeneti portok változásai várhatóan milyen viszonyban lesznek az órajelünk változásaival, a kimeneti késleltetéssel pedig azt fejezzük ki, hogy kimeneti portjainktól milyen viselkedést várunk el ugyancsak az órajelhez viszonyítva. Az itt bemutatott, külső eszközöket feltételező modell az az absztrakció, amit tervezőrendszereink ezeknek az elvárt időzítési viszonyoknak a kifejezésére használnak.

6.2.1.4 Időzítési kivételek *(timing exceptions)*

Az órajelfeltétel, valamint a bemeneti és kimeneti késleltetésekre vonatkozó feltételek definiálásával áramkörünk minden adatútját „lefedtük", tehát elvileg hátradőlhetünk és várhatjuk, hogy a szintézis algoritmusok elvégezzék a piszkos munkát. Ha igazán jó eredményt

akarunk elérni, akkor viszont gondoskodhatunk arról is, hogy ezeknek az algoritmusoknak a dolgát ne nehezítsük meg feleslegesen. A legtöbb áramkörben vannak olyan adatutak, amelyekre a funkció pontos ismeretében egyedi időzítési előírásokat, ún. időzítési kivételeket *(timing exception)* fogalmazhatunk meg. Ezeket az előírásokat azért nevezzük kivételeknek, mert általában először a fenti három peremfeltételt rögzítjük, majd – miután így már az áramkör minden adatútjára van valamilyen megkötésünk – ezután adjuk hozzá a speciálisan egy-egy adatútra vonatkozó egyedi feltételt, amely felülírja vagy kiegészíti az előzőleg megfogalmazottakat. Ezek a kivételek általában (bár nem minden esetben) lazítják az órajelfeltétel által az adott lokális adatút késleltetésével szemben támasztott követelményt, ami azt jelenti, hogy a logikai és fizikai szintézis algoritmusok nem próbálják feleslegesen optimalizálni az adott adatutat, tehát az áramkörünk mérete és fogyasztása is kisebb lehet, a szintézis folyamata pedig hamarabb befejeződhet.

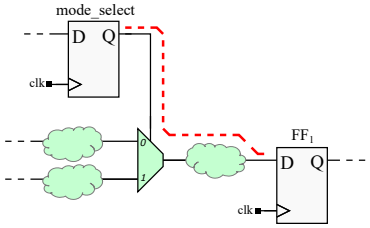
Minimális/maximális késleltetés *(min delay / max delay)* • Az időzítési kivételek legközvetlenebbül megfogalmazott formája, ha egy konkrét adatútra vonatkozóan számszerűen definiáljuk a megengedhető legnagyobb *(max delay)* és/vagy legkisebb *(min delay)* késleltetésértéket. Ennek a megoldásnak az előnye pontosan a közvetlenségében rejlik, hiszen ekkor nem az órajel frekvenciáján vagy a külső, általában csak feltételezett vezetékkésleltetéseken keresztül, közvetett módon definiáljuk az adatút késleltetéseinek határértékeit, hanem egyszerűen egy-egy számértékként adjuk meg azokat. Hátránya, hogy hatékony alkalmazásához az áramkör működését igen jól kell ismernünk. A 6-17. ábra egy 50 MHz-es SPI slave interfésszel rendelkező áramkör egy részletét és a MISO kimenet SCLK bemenethez viszonyított, elvárt időzítési viszonyait szemlélteti.



6-17. ábra. *SPI slave interfészáramkör részlete és elvárt időzítési viszonyai*

Az SPI slave áramkör az SCLK órajel felfutó éleinél kapuzza ki a MISO kimenetre az adatot. Ahhoz, hogy a master eszköz az SCLK órajel következő lefutó élénél megfelelően mintavételezni tudja a MISO vonalat, biztosítani kell, hogy az ekkor már stabil. A slave oldalon megfogalmazott időzítési peremfeltételekkel azt szeretnénk elérni, hogy a MISO kimenet az SCLK órajel pozitív félperiódusa közepének a 2,5 ns-os környezetében stabilizálódjon. Az SCLK bemeneti port és a MISO kimeneti port közötti adatútra tehát egy 2,5 ns-os minimális és egy 7,5 ns-os maximális késleltetést írunk elő.

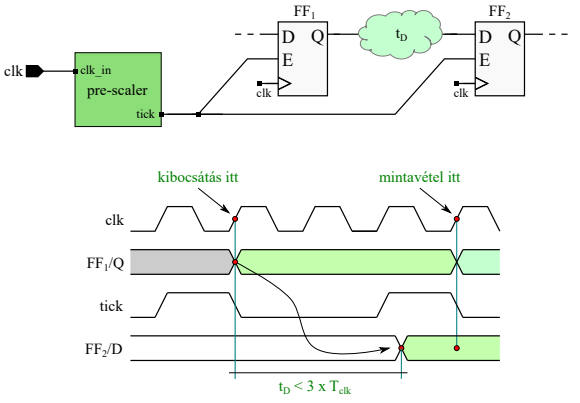
Időzítési szempontból érdektelen adatút *(false path)* • A legegyszerűbb időzítési kivétel, ha egy konkrét adatútra vonatkozóan egész egyszerűen nem kívánunk semmilyen időzítési kritériumot megfogalmazni. Tegyük fel, hogy áramkörünknek két fő működési módja van, és az aktív működési módot a *mode_select* flip-flop értéke határozza meg. Ez a flip-flop minden bizonnyal a 6-18. ábrának megfelelő kapcsolásban több helyen is megjelenik a netlistában.



6-18. ábra. *Időzítés szempontjából irreleváns lokális adatút (false path)*

Ha a két működési mód közötti váltás nincs szigorú időzítési feltételekhez kötve, vagyis nem szükséges, hogy az üzemmódváltás két órajel-felfutóél között megtörténjen, akkor a *mode_select* flip-flop értékét statikusnak tekinthetjük. Ekkor a *mode_select* és az FF_1 flip-flop közötti lokális adatút késleltetésének nincs jelentősége. Az ilyen adatutakat időzítés szempontjából érdektelen adatutaknak *(false path)* nevezzük.

Többciklusú adatút *(multicycle path)* • Az órajelfeltétel megadásával áramkörünk minden belső lokális adatútjára vonatkozóan megfogalmaztunk egy megengedhető maximális késleltetést. E feltétel azt mondja ki, hogy az adatnak egyetlen órajelcikluson belül meg kell érkeznie a fogadó flip-flop bemenetére. Bizonyos esetekben – az áramkör architekturális sajátosságai miatt – egy lokális adatút megengedett maximális késleltetése az órajelperiódus többszöröse lehet. A 6-19. ábrán látható példában a kibocsátó flip-flop és a fogadó flip-flop órajel-engedélyező bemenetét egy órajel-előosztóval hajtjuk meg, amely az órajelnek csak minden harmadik felfutó élénél engedélyezi a két flip-flopot.



6-19. ábra. *Többciklusú adatút (multicycle path)*

Ez azt jelenti, hogy az FF_1 és FF_2 flip-flop közötti lokális adatút esetén a jelnek három teljes órajelperiódus áll rendelkezésére, hogy megérkezzen a fogadó flip-flop bemenetére. Az ilyen lokális adatutakat többciklusú adatutaknak *(multicycle path)* nevezzük. Ezeknél az adatutaknál természetesen ugyanúgy jelen vannak a korábban tárgyalt késleltetés-összetevők, de ahogy a belső lokális adatutak esetén elegendő volt magát az órajel-periódust követelményként rögzíteni, úgy a többciklusú adatutaknál, mintegy másodlagos kritériumot elegendő magát a szorzótényezőt rögzíteni. Így valójában már két peremfeltételt adtunk meg ugyanarra az adatútra; egy órajelfeltételt és egy szorzótényezőt.

6.2.2 Az időzítési kényszerek formális rögzítése

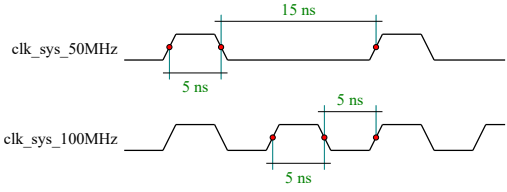
A bemutatott időzítési peremfeltételek rögzítése, majd annak ellenőrzése, hogy a szintetizált áramkör megfelel-e ezeknek a peremfeltételeknek, az RTL tervező feladata. Az időzítési feltételek formális megfogalmazása az adott technológiára jellemző parancsfájlok segítségével történik. Ebben az alfejezetben részletesen bemutatjuk ennek módját a legelterjedtebben használt parancsfájl-típus, az SDC *(Synopsys Design Constraint)* esetén.

Órajelfeltétel *(clock constraint)* ⬩ Az áramkörünk órajeleit a `create_clock` SDC-paranccsal definiálhatjuk a 6-20. ábrának megfelelő módon.

SDC

```
# freq.: 50 MHz, duty-cycle: 25%
create_clock -name system_clock_slow -period 20 -waveform {0 5} [get_ports clk_sys_50MHz]

# freq.: 100 MHz, duty-cycle: 50%
create_clock -name system_clock_fast -period 10 [get_ports clk_sys_100MHz]
```



6-20. ábra. *A `create_clock` SDC-parancs használata*

Az órajelekkel kapcsolatban tehát definiálnunk kell azok periódusidejét és kitöltési tényezőjét, ha az 50%-tól eltérő. A `name` kapcsolóval megadott névre azért van szükség, mert ez az az egyedi azonosító, amellyel a további SDC-parancsok hivatkozhatnak az adott órajelre.

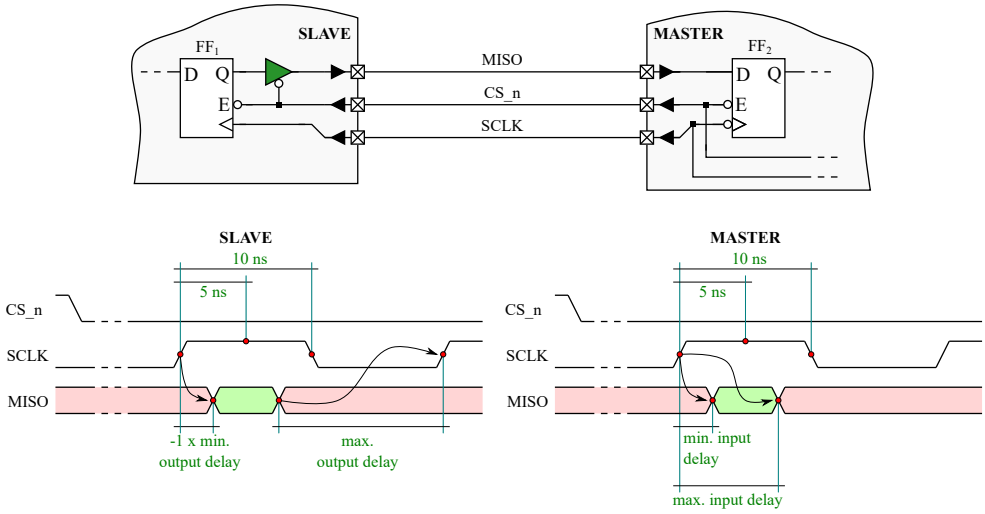
Bemeneti és kimeneti késleltetés *(input delay, output delay)* ⬩ A bemeneti és kimeneti késleltetésekre vonatkozó peremfeltételeket rendre a `set_input_delay` és a `set_output_delay` SDC-parancsokkal definiálhatjuk. A 6-21. ábrán látható példa két, SPI interfészen összeł kapcsolt áramkörnek a MISO jel időzítése szempontjából releváns részleteit mutatja be. Ahogy a 6-17. ábrán látható példában is, ebben az esetben is azt szeretnénk elérni, hogy a slave eszköz MISO kimenete a master eszköz által meghajtott SCLK jel pozitív félperiódusa közepének 2,5 ns-os környezetében változzon meg. Korábban ennek az időzítési

követelménynek a kifejezésére a slave-oldali áramkör SCLK→MISO adatútjára definiált *max delay* és *min delay* kritériumokat használtuk. Ugyanezen időzítési viszonyokat a slave áramkör MISO kimenetére felírt kimeneti és a master áramkör MISO bemenetére felírt bemeneti késleltetésekkel is megfogalmazhatjuk.

SDC

```
# SLAVE
create_clock -name spi_clock -period 20 [get_ports SCLK]
set_output_delay -clock spi_clock -min -2.5 [get_ports MISO]
set_output_delay -clock spi_clock -max 12.5 [get_ports MISO]

# MASTER
create_clock -name spi_clock -period 20 [get_ports SCLK]
set_input_delay -clock spi_clock -min 2.5 [get_ports MISO]
set_input_delay -clock spi_clock -max 7.5 [get_ports MISO]
```



6-21. ábra. *A set_output_delay és a set_input_delay parancsok használata*

Maximális/minimális késleltetés *(max/min delay)* • Egy adatútra vonatkozóan a maximális és minimális késleltetések közvetlenül rendre a `set_max_delay` és a `set_min_delay` SDC-parancsok segítségével adhatók meg. A 6-22. ábra azt mutatja be, hogy e parancsok segítségével hogyan fogalmazhatók meg az SPI slave áramkör MISO kimenetére vonatkozó, korábban is használt időzítési kritériumok.

SDC

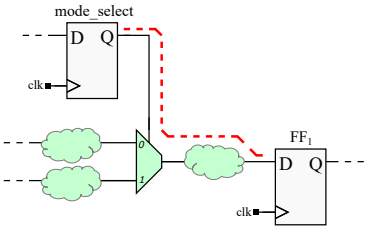
```
create_clock -name spi_clock -period 20 [get_ports SCLK]
set_min_delay 2.5 -from [get_clocks spi_clock] -to [get_ports MISO]
set_max_delay 7.5 -from [get_clocks spi_clock] -to [get_ports MISO]
```

6-22. ábra. *A* ***`set_min_delay`*** *és* ***`set_max_delay`*** *SDC-parancsok használata*

Időzítési szempontból érdektelen adatút *(false path)* ⬩ Az időzítés szempontjából érdektelen adatutakat a `set_false_path` SDC-paranccsal jelölhetjük ki. A parancs használatát a 6-23. ábra szemlélteti.

SDC

```
set_false_path -from [get_cells {mode_select*}] -to [get_cells {FF1*}]
```



6-23. ábra. *A set_false_path SDC-parancs használata*

A `set_false_path` parancsot gyakran használjuk arra, hogy két aszinkron órajeltartomány határán átmenő adatutakra vonatkozóan kikapcsoljuk az időzítési kritériumokat. Ebben az esetben ti. az időzítési problémákat architekturális szinten, ún. órajel-szinkronizáló áramkörökkel kezeljük. Ezt a problémakört a 6.3. alfejezetben részletesen is tárgyaljuk. A 6-24. ábra azt mutatja be, hogy egyetlen SDC-parancs segítségével hogyan lehet bármely olyan adatúton kikapcsolni az időzítési kritériumok ellenőrzését, amelyek kibocsátó flip-flopja az egyik, a fogadó flip-flopja pedig egy másik órajelforrásról jár.

SDC

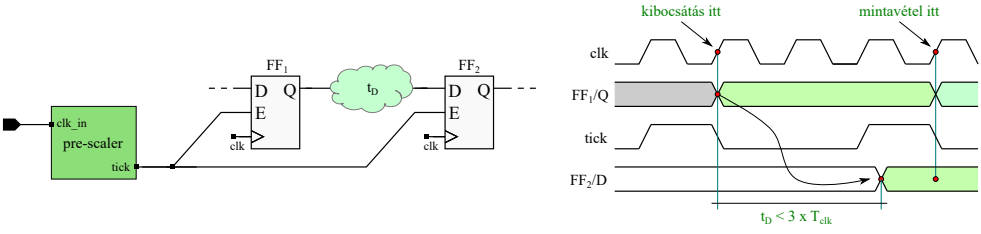
```
create_clock -name system_clock_slow -period 20 -waveform {0 5} [get_ports clk_sys_50MHz]
create_clock -name system_clock_fast -period 10 [get_ports clk_sys_100MHz]
set_false_path -from [get_clocks system_clock_fast] -to [get_clocks system_clock_slow]
```

6-24. ábra. *Órajeltartományok közötti adatutak időzítési vizsgálatának kikapcsolása egyetlen SDC-paranccsal*

Többciklusú adatút *(multicycle path)* ⬩ Többciklusú adatútra vonatkozó időzítési kritériumot a `set_multicycle_path` SDC-paranccsal állíthatunk be. E parancs használatát a 6-25. ábra szemlélteti.

SDC

```
set_multicycle_path -setup 3 -from [get_cells {FF1*}] -to [get_cells {FF2*}]
set_multicycle_path -hold 2 -from [get_cells {FF1*}] -to [get_cells {FF2*}]
```



6-25. ábra. *A `set_multicycle_path` SDC-parancs használata*

A fenti példában az látható, hogy egyazon többciklusú adatútra vonatkozóan két SDC-parancsot fogalmaztunk meg. Az első parancs azt rögzíti, hogy mivel a fogadó flip-flop csak minden harmadik órajel-felfutóélnél fogja mintavételezni a D bemenetét, ezért a beállítási időre vonatkozó ellenőrzési pontot el kell tolni. A `set_multicycle_path` parancs alapesetben – tehát ha nem használjuk sem a `setup` sem a `hold` kapcsolót – a tartási időre vonatkozó ellenőrzési pontot is eltolja, ami az esetek többségében nem megfelelő; a tartási időket most is a kibocsátás pillanatában kell ellenőrizni a fogadó flip-flop bemenetén. Ennek kifejezéséhez a `set_multicycle_path` parancsot ketté kell bontani a fenti példának megfelelően. A tartási időre vonatkozó `set_multicycle_path` parancsban a hold kapcsoló argumentumát az adatút ciklusszámánál eggyel kisebbre kell állítani.

A regiszterek közötti kombinációs hálózatok becsült méretén és az előzetes szintézis eredményein alapuló optimalizáción, valamint az időzítési peremfeltételek rögzítésén kívül az RTL tervező feladatai közé tartozik egy másik, ugyancsak a D flip-flopok időzítési jellegzetességeire visszavezethető problémakör, az órajeltartományok közötti adatátvitel megfelelő kezelése is. Ennek részleteit tárgyaljuk a következő alfejezetben.

6.3 Adatátvitel órajeltartományok határán, reset-szinkronizáció

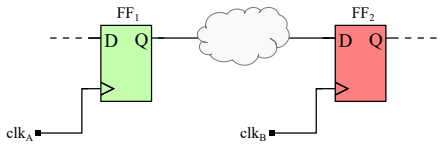
A korábbi fejezetekben bemutatott példák mind egyetlen órajelforrás létezését feltételezték, és funkciójuk ellátásához ennél többre nem is volt szükség. Összetett makrocellák esetén, amelyekben több, eltérő feladatot ellátó almodul működik együtt, általában több órajelforrás van. Erre jó példa egy mikrokontroller, amely egy mikroprocesszor magból és egy sor perifériavezérlő áramkörből áll. A fogyasztás csökkentése érdekében a processzormag és az összes perifériavezérlő áramkör általában egy-egy saját órajelforrással van ellátva. Ez a megoldás lehetővé teszi, hogy a lassú perifériavezérlők alacsonyabb frekvenciájú, a gyors perifériavezérlők, valamint a processzormag pedig magasabb frekvenciájú órajelről járjanak, sőt, a nem használt részáramkörök órajelhálózatát teljesen ki is kapcsolhatjuk az eszköz fogyasztásának drasztikus csökkentése érdekében.

Az egy órajelforrás által meghajtott flip-flopok összességét órajeltartománynak, vagy órajel-doménnek nevezzük. A különböző órajeltartományok órajeleinek frekvencia- és fázisviszonyai alapján a következő fogalmakat definiálhatjuk:

- **Szinkron órajeltartományok**: Két, egymáshoz képest szinkron órajeltartomány órajeleinek frekvenciája lehet különböző, de fázisviszonyuk ismert és állandó. Ha egy makrocella két almodulját egyazon PLL *(Phase-Locked Loop)* két különböző frekvenciájú kimenetével hajtjuk meg, akkor e két almodul flip-flopjai két különböző, de egymáshoz képest szinkron órajeltartományt képeznek. A két órajel fázisa állandó lesz, mert a PLL mindkét kimenete ugyanahhoz a referencia-órajelhez van szinkronizálva.
- **Aszinkron órajeltartományok**: Két, egymáshoz képest aszinkron órajeltartomány órajeleinek frekvenciájára, fázisviszonyára és ezek időbeli viselkedésére, esetleges változására semmi megkötés nincs, ezeket a paramétereket ismeretlennek kell te-

kintenünk. Ha egy makrocella két almodulját két külön órajelforrással, például két külön PLL-lel vagy két külön kristályoszcillátorral hajtjuk meg, akkor a két almodul flip-flopjai egymáshoz képest teljesen aszinkron működésűek lesznek.

Ha egy makrocellán belül több órajeltartomány van, akkor az ezek közötti adatátvitelt az időzítési hibák elkerülése végett igen körültekintően kell megvalósítani. A 6-26. ábra ezt a helyzetet mutatja be egy egyszerű, egybites jel átvitelének példáján.



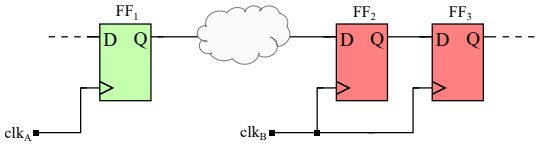
6-26. ábra. *Egybites jel átvitele órajeltartományok határán*

A fenti helyzetet nevezzük az órajeltartományok közötti adatátvitel (*Clock Domain Crossing*, CDC) problémájának; az FF_1 jelű, a clk_A órajelforrásról járó flip-flop kimenetét az FF_2 jelű, a clk_B órajelforrásról járó flip-flop mintavételezi. Ha a két órajeltartomány szinkron, akkor a helyzet nem különösebben érdekes, hiszen ekkor pontosan ismerjük a kibocsátó és fogadó órajel-felfutóélek egymáshoz viszonyított helyzetét, tehát a két flip-flop közötti vezetékezés és kombinációs hálózat ismeretében az időzítési viszonyok a korábban bemutatott módszerekkel kezelhetők. Ha azonban a két órajeltartomány aszinkron, akkor ezek a módszerek nem használhatók, ekkor ui. a kibocsátó flip-flop órajelének felfutása a fogadó flip-flop órajelének felfutó éléhez képest bármikor bekövetkezhet, ami azt jelenti, hogy a fogadó flip-flop D bemenetén a jel bármikor megváltozhat, a flip-flop beállítási és tartási időkre vonatkozó időzítési kritériumainak betartását tehát semmi sem biztosítja.

Aszinkron órajeltartományok közötti adatutak esetén tehát az időzítési peremfeltételek nem is értelmezhetők. Az ilyen adatutakat rendszerint az időzítési viszonyok szempontjából érdektelen adatutakként *(false path)* definiáljuk, az adatátvitelt pedig a két órajeltartomány határán elhelyezett, ún. órajelszinkronizáló áramkörökkel oldjuk meg.

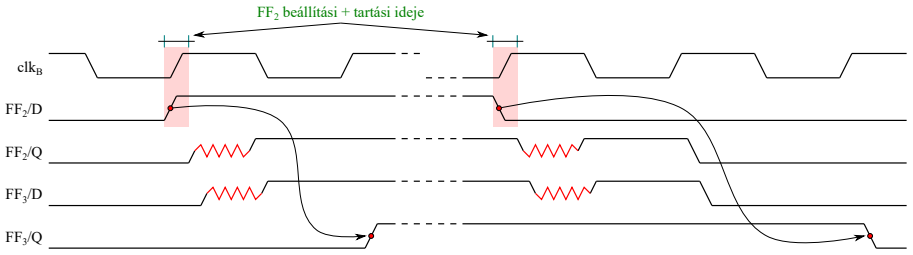
6.3.1 Órajelszinkronizáló áramkörök

Double-flop szinkronizáló fokozat • Az ún. statikus, vagyis az áramkörünk órajeleihez képest ritkán változó jelek átvitelének legalapvetőbb eleme a double-flop szinkronizáló fokozat (lásd 6-27. ábra).



6-27. ábra. *A double-flop szinkronizáló fokozat*

Statikus jelnek azokat a jeleket tekintjük, amelyeknek két, egymást követő változása között a fogadó oldal (az ábrán clk_B) órajelének legalább két teljes periódusa (T_B) telik el. Ez a feltétel a double-flop szinkronizáló fokozat működése alapján válik érthetővé: Az FF_1 jelű, a clk_A órajelről járó flip-flop kimenete tetszőleges időpontban megváltozhat. Ha az FF_2 jelű, a clk_B órajelről járó flip-flop D bemenete a flip-flop beállítási és tartási időkre vonatkozó időzítési kritériumait nem megsértve változik meg, akkor a mintavétel sikeres lesz, a megváltozott jel pedig az FF_3 flip-flop Q kimenetén a clk_B órajel következő felfutó élénél, immár a clk_B órajelhez szinkronizálva megjelenik. Ha az FF_2 flip-flop D bemenete a clk_B órajel felfutó éléhez közel, a flip-flop beállítási vagy tartási idejére vonatkozó időzítési kritériumok valamelyikét megsértve változik meg, akkor az FF_2 flip-flop bizonyos valószínűséggel metastabil állapotba kerül. A flip-flop kimenete egy idő után magától stabilizálódik. Ha a stabilizálódás még a clk_B órajel következő felfutó éle előtt – pontosabban az FF_3 flip-flop beállítási időablaka előtt – bekövetkezik, akkor az FF_3 flip-flop már stabil jelet mintavételez, tehát nem kerül metastabil állapotba. Azonban az, hogy az FF_3 flip-flop nem kerül metastabil állapotba még nem jelenti azt, hogy a jel megváltozása sikeresen átjutott a clk_B órajeltartományba, hiszen a metastabil állapotból visszatérve az FF_2 flip-flop Q kimenetének értéke független a bemenetének értékétől. Ha a Q kimenet véletlenül éppen azt az értéket veszi fel, ami a bemenet értéke, akkor a mintavétel sikerült, a jel tehát átjutott a clk_B órajeltartományba. Ha a metastabil állapotból visszatérve az FF_2 flip-flop a bemenetével ellentétes értéket vesz fel, akkor a jel megváltozása csak egy órajelciklus késéssel jelenik meg az FF_3 flip-flop kimenetén. E két esetet a 6-28. ábra szemlélteti.



6-28. ábra. *A double-flop szinkronizáló fokozat működése*

A double-flop szinkronizáló fokozat tehát valójában nem akadályozza meg a metastabilitás kialakulását az áramkörben, de izolálja azt; a fenti kapcsolásban csak az FF_2 flip-flop kerülhet metastabil állapotba. Elvileg – mivel a flip-flopok stabilizálódási ideje *(resolution time)* nem determinisztikus – elképzelhető, hogy az FF_2 flip-flop metastabil állapota a clk_B órajel periódusidejénél tovább fennáll, aminek hatására az FF_3 flip-flop is metastabil állapotba kerülhet. Ez azt jelentené, hogy a szinkronizáló fokozat kudarcot vallott, hiszen az FF_3 flip-flop kimenetét már bármely másik, clk_B órajeltartománybeli flip-flop mintavételezhetné, ami – a 6-8. ábrán látottakhoz hasonló módon – funkcionális hibákat okozhatna. Ennek a valószínűsége azonban rendkívül csekély, különösen akkor, ha az FF_2 flip-flop terhelését minimalizáljuk, például azáltal, hogy nagyon közel helyezzük el az FF_3 flip-flophoz. Ennek ellenére magas órajel-frekvenciákon indokolt lehet a fogadó órajeltartományban nem

kettő, hanem három flip-flopot használni szinkronizációra, de a mai modern technológiák esetén, 100 MHz nagyságrendű órajel-frekvenciákon az egyszerű double-flop szinkronizáló fokozat is kielégítő megbízhatóságot nyújt.

A double-flop szinkronizáló fokozatot nemcsak órajeltartományok határán, hanem az áramkör bemenetein is alkalmazzuk olyan jelek esetén, amelyek változásai a lokális órajelhez nem szinkronizáltak. Ezt a megoldást láthatjuk a 4.7. alfejezetben bemutatott pergésmentesítő, késleltetésmérő és BFSK demodulátor áramkörökben. Ezekben az alkalmazásokban a bejövő jel tipikusan nincs a lokális órajelhez szinkronizálva, ezért ezek fogadása mindig egy-egy double-flop szinkronizálóval történik.

Impulzusszinkronizálók *(pulse synchronizers)* • Az egyszerű double-flop szinkronizáló statikus jelszint átvitelére alkalmas. A visszacsatolatlan impulzusszinkronizáló *(toggle pulse synchronizer)* fokozat a clk_A órajeltartományban generált, egyetlen periódusnyi (T_A) pozitív impulzust a clk_B órajeltartománybeli, ugyancsak egyetlen periódus hosszúságú (T_B) impulzussá alakítja át (lásd 6-29. ábra).

VHDL

```
architecture rtl of pulse_sync is

  signal FF1, FF2, FF3, FF4: std_logic;

begin

  process ( clkA, rstA )
  begin
    if ( rstA = '1' ) then FF1 <= '0';
    elsif ( rising_edge(clkA) ) then
      if ( pulseA = '1' ) then
        FF1 <= not FF1;
      end if;
    end if;
  end process;

  process ( clkB, rstB )
  begin
    if ( rstB = '1' ) then
      FF2 <= '0'; FF3 <= '0'; FF4 <= '0';
    elsif ( rising_edge(clkB) ) then
      FF2 <= FF1;
      FF3 <= FF2;
      FF4 <= FF3;
    end if;
  end process;

  pulseB <= FF4 xor FF3;

end architecture rtl;
```

Verilog

```
module pulse_sync (...);

  logic FF1, FF2, FF3, FF4;

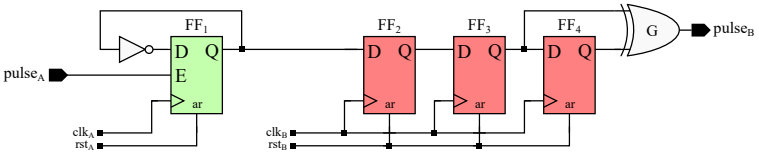
  always @ ( posedge clkA, posedge rstA )
  begin
    if ( rstA == 1'b1 ) FF1 <= 1'b0;
    else
      if ( pulseA == 1'b1 ) FF1 <= ~FF1;
  end

  always @ ( posedge clkB, posedge rstB )
  begin
    if ( rstB == 1'b1 ) begin
      FF2 <= 1'b0; FF3 <= 1'b0; FF4 <= 1'b0;
    end else begin
      FF2 <= FF1;
      FF3 <= FF2;
      FF4 <= FF3;
    end
  end

  assign pulseB = FF4 ^ FF3;

endmodule
```

Áramköri struktúra



6-29. ábra. *Visszacsatolatlan impulzusszinkronizáló fokozat*

Az adó a $pulse_A$ bemeneten egy T_A időtartamú, pozitív impulzust generál, amelynek hatására az FF_1 flip-flop kimenetének értéke megváltozik. A megváltozó jelet a clk_B órajeltartományban az FF_2 és FF_3 flip-flopokból álló double-flop fokozat fogadja és szinkronizálja a clk_B órajelhez. Az FF_4 flip-flop és a G XOR kapu együttesen egy felfutó és lefutó élekre egyaránt érzékeny éldetektort alkotnak. Ennek az éldetektornak a kimenetén T_B időtartamú pozitív impulzus jelenik meg. Az adó oldalon két egymást követő impulzus között legalább $2 \times T_B$ időnek kell eltelnie.

VHDL

```
architecture rtl of pulse_sync_handshake is

  signal FF1, FF2, FF3, FF4, FF5, FF6: std_logic;

begin

  process ( clkA, rstA )
  begin
    if ( rstA = '1' ) then
      FF1 <= '0'; FF5 <= '0'; FF6 <= '0';
    elsif ( rising_edge(clkA) ) then
      if ( pulseA = '1' ) then FF1 <= '1';
      elsif ( FF6 = '1' ) then FF1 <= '0';
      end if;
      FF5 <= FF3;
      FF6 <= FF5;
    end if;
  end process;

  busy <= FF6 or FF1;

  process ( clkB, rstB )
  begin
    if ( rstB = '1' ) then
      FF2 <= '0'; FF3 <= '0'; FF4 <= '0';
    elsif ( rising_edge(clkB) ) then
      FF2 <= FF1;
      FF3 <= FF2;
      FF4 <= FF3;
    end if;
  end process;

  pulseB <= not FF4 and FF3;

end architecture rtl;
```

Verilog

```
module pulse_sync_handshake (...);

  logic FF1, FF2, FF3, FF4, FF5, FF6;

  always @ ( posedge clkA, posedge rstA )
  begin
    if ( rstA == 1'b1 ) begin
      FF1 <= 1'b0; FF5 <= 1'b0; FF6 <= 1'b0;
    end else begin
      if ( pulseA == 1'b1 ) FF1 <= 1'b1;
      else if ( FF6 == 1'b1 ) FF1 <= 1'b0;
      FF5 <= FF3;
      FF6 <= FF5;
    end
  end

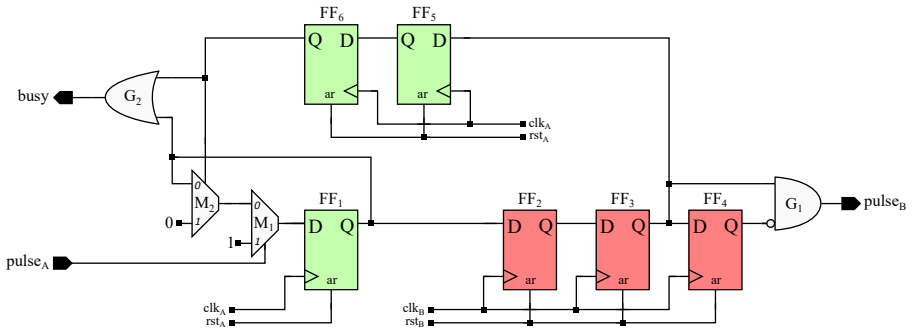
  assign busy = FF6 | FF1;

  always @ ( posedge clkB, posedge rstB )
  begin
    if ( rstB == 1'b1 ) begin
      FF2 <= 1'b0; FF3 <= 1'b0; FF4 <= 1'b0;
    end else begin
      FF2 <= FF1;
      FF3 <= FF2;
      FF4 <= FF3;
    end
  end

  assign pulseB = ~FF4 & FF3;

endmodule
```

Áramköri struktúra



6-30. ábra. *Visszacsatolt impulzusszinkronizáló fokozat*

Az ilyen jellegű korlátok azért jelenthetnek nehézséget egy újrafelhasználható HDL modell készítésekor, mert ilyenkor tipikusan sem az adó, sem a vevő oldal órajelének periódusidejét, sőt ezek viszonyát sem ismerjük. Ilyen esetekben visszacsatolást kell létrehoznunk, amely segítségével az adó oldal megbizonyosodhat róla, hogy az általa generált jelzés átért a fogadó órajeltartományba. Az ilyen megoldásokat hand-shake, vagy visszacsatolt szinkronizálóknak nevezzük. A 6-30. ábra egy ilyen, visszacsatolt impulzusszinkronizáló *(hand-shake pulse synchronizer)* áramkört mutat be.

Kiindulási állapotban az áramkör minden flip-flopja logikai alacsony szinten áll. Az adó ezúttal is egy T_A időtartamú, pozitív impulzust generál a $pulse_A$ bemeneten, amelynek hatására az FF_1 flip-flopba logikai magas szint íródik. Mivel az FF_5 és FF_6 flip-flopok ekkor alacsony szintűek, az FF_1 flip-flopban lévő magas szint az M_1 és M_2 multiplexereken keresztül önfenntartó lesz, egyúttal az áramkör busy kimenete is magas szintű lesz, jelezve az adó oldal számára, hogy az impulzus átvitele még nem történt meg. Az FF_1 flip-flop kimenetére íródó magas szintet az FF_2 és FF_3 flip-flopokból álló double-flop szinkronizáló fogadja és szinkronizálja a clk_B órajelhez. Az FF_4 flip-flop és a G_1 kapu együttesen egy felfutóél-detektort alkotnak, amelynek kimenetén egy T_B időtartamú pozitív impulzus jelenik meg, az impulzus tehát átért a clk_B órajeltartományba. Az FF_3 flip-flop kimenetét az FF_5 és FF_6 flip-flopok által megvalósított double-flop szinkronizáló fokozaton keresztül visszacsatoljuk a clk_A órajeltartományba. Amikor a visszacsatolt logikai magas szint megjelenik az FF_6 flip-flop kimenetén, akkor az M_1 és M_2 multiplexereken keresztül az FF_1 flip-flop nullázódik, de mivel az FF_6 flip-flop kimenete továbbra is magas szintű, ezért a busy jel továbbra is aktív. Az átvitel akkor tekinthető csak teljesnek, ha az FF_1 flip-flopba beírt nulla érték szintén átért a clk_B órajeltartományba, majd a visszacsatoláson keresztül FF_5 és FF_6 is nullázódik. Ekkor a busy jel deaktiválódik, jelezvén az adó oldalnak, hogy az impulzus átvitele maradéktalanul megtörtént, a szinkronizáló fokozat tehát készen áll egy újabb impulzus átvitelére.

Befogó szinkronizáló *(capture synchronizer)* • A 6-31. ábrán látható, ún. befogó szinkronizáló fokozat rendkívül rövid impulzusok „megszelídítésére", azokból egy, a lokális órajelforrás periódusidejének megfelelő hosszúságú, pozitív impulzus előállítására alkalmas.

A rövid impulzusokat generáló bemenetet az áramkör $pulse_A$ bemenetére kapcsoljuk. A korábbi példákkal ellentétben a bemenet ezúttal nem egy flip-flop adatbemenetére, hanem annak órajelbemenetére kapcsolódik. Ha a $pulse_A$ bemeneten megjelenik egy rövid impulzus, akkor az FF_1 flip-flop kimenete megváltozik. Az áramkör működése innentől kezdve tulajdonképpen a visszacsatolatlan impulzusszinkronizálóéval egyezik meg; az FF_2 és FF_3 flip-flopok által megvalósított double-flop fokozat a clk_B órajelhez szinkronizálja az FF_1 flip-flop kimenetének változását, az FF_3 flip-flop és a G kapu pedig előállítja a clk_B órajelhez szinkronizált, T_B időtartamú pozitív impulzust. A bemeneti jel minimális impulzusszélessége attól függ, hogy az FF_1 flip-flop milyen széles impulzust képes még órajel-felfutóélként értelmezni. Ez az időtartam rendszerint lényegesen kisebb, mint amilyen órajel-frekvencián az adott flip-flopot tipikusan alkalmazzuk.

Az eddig bemutatott szinkronizáló áramkörök egybites jelek átvitelére képesek. Az órajeltartományok határán azonban természetesen több bites vektorokat is át kell tudnunk

VHDL

```
architecture rtl of capture_sync is

  signal FF1, FF2, FF3, FF4: std_logic;

begin

  process ( pulseA, rstA )
  begin
    if ( rstA = '1' ) then FF1 <= '0';
    elsif ( rising_edge(pulseA) ) then
      FF1 <= not FF1;
    end if;
  end process;

  process ( clkB, rstB )
  begin
    if ( rstB = '1' ) then
      FF2 <= '0'; FF3 <= '0'; FF4 <= '0';
    elsif ( rising_edge(clkB) ) then
      FF2 <= FF1;
      FF3 <= FF2;
      FF4 <= FF3;
    end if;
  end process;

  pulseB <= FF3 xor FF4;

end architecture rtl;
```

Verilog

```
module capture_sync (...);

  logic FF1, FF2, FF3, FF4;

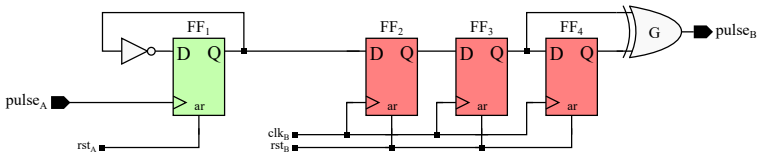
  always @ ( posedge pulseA, posedge rstA )
  begin
    if ( rstA == 1'b1 ) FF1 <= 1'b0;
    else FF1 <= ~FF1;
  end

  always @ ( posedge clkB, posedge rstB )
  begin
    if ( rstB == 1'b1 ) begin
      FF2 <= 1'b0; FF3 <= 1'b0; FF4 <= 1'b0;
    end else begin
      FF2 <= FF1;
      FF3 <= FF2;
      FF4 <= FF3;
    end
  end

  assign pulseB = FF3 ^ FF4;

endmodule
```

Áramköri struktúra



6-31. ábra. *Befogó szinkronizáló fokozat*

vinni. Első gondolatunk az lehet, hogy az átvivendő vektor minden egyes bitjét egy-egy double-flop szinkronizáló fokozattal fogadva a probléma könnyen megoldható, a helyzet azonban sajnos nem ilyen egyszerű. Ahogy az a 6-28. ábrán is látható, a double-flop szinkronizáló fokozat átviszi ugyan a statikus jeleket, de a késleltetése nem determinisztikus. A változások átvitelében T_B időtartamnyi bizonytalanság van, ami azt jelenti, hogy ha a vektornak egyszerre több bitje is megváltozik – ami nyilván egy általános, több bites busz esetén így is van –, akkor a vektor egyes bitjei a többihez képest egy órajel-periódussal megkésve jelennének meg a clk_B órajeltartományban. A vektor minden változása a clk_A órajeltartományban egy $2 \times T_B$ időtartamú tranziens után jelenne csak meg a clk_B órajeltartományban, vagyis a clk_B órajelnek lesz egy olyan felfutó éle, amelynél a vektor nem összetartozó biteket tartalmaz. A probléma ezzel az, hogy a fogadó oldalnak nem áll módjában megállapítani, hogy a vektor pontosan melyik órajel-felfutóélnél van inkonzisztens állapotban és mikor tekinthető érvényesnek az értéke.

Gray szinkronizáló • A vektorok átvitelével a probléma tehát az, hogy általános esetben egy vektornak több bitje is változhat egyazon órajel-felfutóél hatására. Létezik azonban egy speciális eset, amikor ez nem igaz. Tegyük fel, hogy a vektor, amit át akarunk vinni

VHDL

```
architecture rtl of gray_sync is

  function gray_encoder (bin_in:  unsigned) return
                         unsigned is
    variable tmp: unsigned (w-1 downto 0);
  begin
    tmp := (others => '0');
    for i in 0 to (w-2) loop
      tmp(i) := bin_in(i) xor bin_in(i+1);
    end loop;
    tmp(w-1) := bin_in(w-1);
    return tmp;
  end gray_encoder;

  function gray_decoder (gray_in:  unsigned) return
                         unsigned is
    variable tmp: unsigned (w-1 downto 0);
  begin
    tmp := (others => '0');
    tmp(w-1) := gray_in(w-1);
    for i in (w-2) downto 0 loop
      tmp(i) := tmp(i+1) xor gray_in(i);
    end loop;
    return tmp;
  end gray_decoder;

  signal V1, V2, V3:  unsigned (w-1 downto 0);

begin

  process ( clkA, rstA )
  begin
    if ( rstA = '1' ) then V1 <= (others => '0');
    elsif ( rising_edge(clkA) ) then
      V1 <= gray_encoder(dataA);
    end if;
  end process;

  process ( clkB, rstB )
  begin
    if ( rstB = '1' ) then
      V2 <= (others => '0');
      V3 <= (others => '0');
    elsif ( rising_edge(clkB) ) then
      V2 <= V1;
      V3 <= V2;
    end if;
  end process;

  dataB <= gray_decoder(V3);

end architecture rtl;
```

Verilog

```
module gray_sync #(...)(...);

  function [w-1:0] gray_encoder;
    input [w-1:0] bin_in;
    logic [w-1:0] tmp;
    begin
      tmp = 0;
      for ( int i = 0; i <= w-2; i = i+1 )
        tmp[i] = bin_in[i] ^ bin_in[i+1];
      tmp[w-1] = bin_in[w-1];
      gray_encoder = tmp;
    end
  endfunction

  function [w-1:0] gray_decoder;
    input [w-1:0] gray_in;
    logic [w-1:0] tmp;
    begin
      tmp = 0;
      tmp[w-1] = gray_in[w-1];
      for ( int i = w-2; i >= 0; i = i-1 )
        tmp[i] = tmp[i+1] ^ gray_in[i];
      gray_decoder = tmp;
    end
  endfunction

  logic [w-1:0] V1, V2, V3;

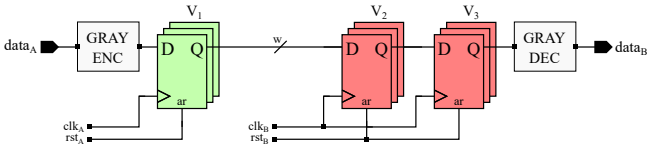
  always @ ( posedge clkA, posedge rstA )
  begin
    if ( rstA == 1'b1 ) V1 <= 0;
    else V1 <= gray_encoder(dataA);
  end

  always @ ( posedge clkB, posedge rstB )
  begin
    if ( rstB == 1'b1 ) begin
      V2 <= 0; V3 <= 0;
    end else begin
      V2 <= V1;
      V3 <= V2;
    end
  end

  assign dataB = gray_decoder(V3);

endmodule
```

Áramköri struktúra



6-32. ábra. *Gray-szinkronizáló fokozat*

egyik órajeltartományból a másikba, egy számlálót valósít meg, amelyre igaz, hogy értéke csak kétféleképpen tud megváltozni: vagy eggyel nő, vagy eggyel csökken. Az egyes bitek változása szempontjából ez önmagában még nem jelent számunkra semmi könnyebbséget, hiszen az egész számokat binárisan ábrázolva két szomszédos érték Hamming-távolsága

VHDL

```
architecture rtl of vector_sync is

  signal FF1, FF2, FF3, FF4:  std_logic;
  signal V1, V2: std_logic_vector (w-1 downto 0);

begin

  process ( clkA, rstA )
  begin
    if ( rstA = '1' ) then
      FF1 <= '0'; V1 <= (others => '0');
    elsif ( rising_edge(clkA) ) then
      if ( strobeA = '1' ) then FF1 <= not FF1;
      end if;
      V1 <= dataA;
    end if;
  end process;

  process ( clkB, rstB )
  begin
    if ( rstB = '1' ) then
      FF2 <= '0'; FF3 <= '0'; FF4 <= '0';
      V2 <= (others => '0');
    elsif ( rising_edge(clkB) ) then
      FF2 <= FF1;
      FF3 <= FF2;
      FF4 <= FF3;
      if ( strobeB = '1' ) then V2 <= V1;
      end if;
    end if;
  end process;

  strobeB <= FF3 xor FF4;
  dataB <= V2;

end architecture rtl;
```

Verilog

```
module vector_sync #(...)(...);

  logic FF1, FF2, FF3, FF4;
  logic [w-1:0] V1, V2;

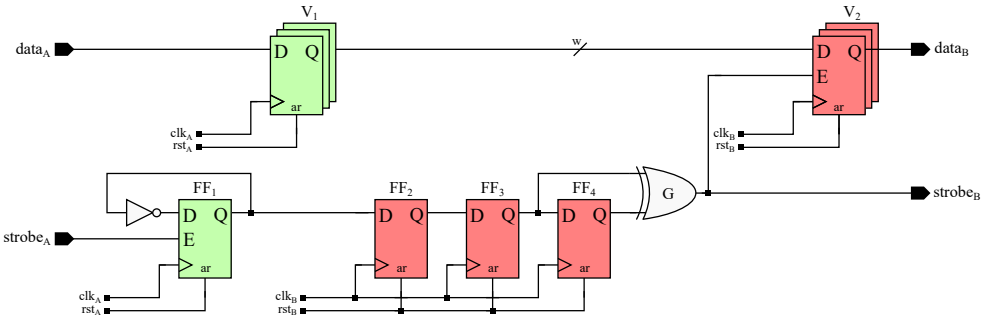
  always @ ( posedge clkA, posedge rstA )
  begin
    if ( rstA == 1'b1 ) begin
      FF1 <= 1'b0; V1 <= 0;
    end else begin
      if ( strobeA == 1'b1 ) FF1 <= ~FF1;
      V1 <= dataA;
    end
  end

  always @ ( posedge clkB, posedge rstB )
  begin
    if ( rstB == 1'b1 ) begin
      FF2 <= 1'b0; FF3 <= 1'b0; FF4 <= 1'b0;
      V2 <= 0;
    end else begin
      FF2 <= FF1;
      FF3 <= FF2;
      FF4 <= FF3;
      if ( strobeB == 1'b1 ) V2 <= V1;
    end
  end

  assign strobeB = FF3 ^ FF4;
  assign dataB = V2;

endmodule
```

Áramköri struktúra



6-33. ábra. *Visszacsatolatlan vektorszinkronizáló fokozat*

akár a bitek számát is elérheti (amikor a számláló átfordul). Ha azonban a számlálót Gray-kódolással ábrázoljuk, a helyzet megváltozik. A Gray-számlálókra az jellemző, hogy ha a számláló eggyel nő, vagy eggyel csökken, akkor a számláló értékét reprezentáló bitsorozatban csak egyetlen bit változik meg (lásd 4-3. táblázat).

Ha tehát az órajeltartományok közötti átvitel előtt a számláló értékét a kibocsátó órajeltartományban Gray-kódolásnak vetjük alá, akkor egy sor double-flop szinkronizálóval is megoldható az átvitel. A fogadó órajeltartományban a double-flop fokozatok kimenetét

dekódoljuk, így visszakapjuk a számláló bináris értékét. A teljes Gray-szinkronizáló fokozatot a 6-32. ábra mutatja be, felhasználva a 4-33. és 4-34. ábrákon bemutatott Gray-enkóder és dekóder logikákat leíró függvények generikus, bitszélesség szerint parametrizált változatát.

Ha a számláló bináris értéke megváltozik a clk_A órajeltartományban, akkor a Gray-kódolt vektorban pontosan egy bit fog megváltozni. A fogadó órajeltartományban csak az éppen megváltozó bithez rendelt double-flop szinkronizáló fog működésbe lépni. Ha ez a double-flop szinkronizáló késve viszi át a megváltozott bemenetet, akkor a fogadó órajeltartományban csupán annyit tapasztalunk, hogy a szinkronizáló fokozat kimenetén még mindig a régi Gray-kódot, a Gray-dekóder kimenetén pedig a régi bináris értéket látjuk. Nincs tehát olyan felfutóél a clk_B órajeltartományban, amelynél a számláló értéke inkonzisztens lenne. A vektor átvitelének ütemezésében tehát az egybites, double-floppal átvitt statikus jeleknél tapasztalt T_B bizonytalanság ugyanúgy megfigyelhető, de az adat maga mindig érvényes lesz. Külön kiemelendő a V_1 flip-flopok jelentősége; ha a Gray-enkóder kimenetét közvetlenül csatolnánk át a clk_B órajeltartományba, akkor az e kombinációs hálózat kimenetén megjelenő, esetleges glitch-ek ugyanolyan problémát okozhatnának, mintha több bit egyszerre változna meg az átvivendő vektorban. A Gray-szinkronizáló tehát csak a V_1 kimeneti bufferekkel együtt működőképes.

Általános vektorszinkronizálók *(recirculation MUX synchronizers)* • A Gray-szinkronizáló fokozat csak speciális, inkrementálást vagy dekrementálást megvalósító számlálókat képes átvinni két aszinkron órajeltartomány között. Szükség van tehát olyan szinkronizáló megoldásra, amely bármilyen több bites busz esetén működőképes. A 6-33. ábrán látható, visszacsatolatlan vektorszinkronizáló áramkör pontosan erre alkalmas.

A vektorszinkronizáló áramkör valójában nem a vektort magát szinkronizálja. A vektor mellé az adó oldal egy strobe jelet is szolgáltat, és amikor a V_1 vektorba új adatot helyez el, akkor annak stabilizálódása után egy T_A időtartamú pozitív impulzussal jelez a strobe jelen, hogy új adat van a buszon. A strobe jel egybites, tehát ezt átvihetjük a clk_B órajeltartományba egy egyszerű impulzusszinkronizáló fokozattal. Ha a strobe jel átért a clk_B órajeltartományba, akkor a double-flop kimenetén lévő éldetektor (FF_4, G) engedélyezi a clk_A órajeltartománybeli vektor mintavételezését végző flip-flopokat, amelyek D bemenete a mintavétel pillanatában már garantáltan stabil. Fontos kiemelni, hogy ez a szinkronizáló megoldás csak akkor működhet, ha a szinkronizáló V_1 és FF_1 jelű flip-flopjait vezérlő áramkör megfelelően használja azt, vagyis a strobe jelet ténylegesen csak akkor küldi el, ha a vektor értékét már beállította.

Ahogy az egybites, visszacsatolatlan impulzusszinkronizáló esetén is, úgy a visszacsatolatlan vektorszinkronizálónál is fennáll az a nehézség, hogy az adó oldal nem feltétlenül ismeri a vevő oldal órajelének periódusidejét, így nem tudhatja, hogy a strobe jel mikor érkezik meg a clk_B órajeltartományba, vagyis mikor tekintheti befejezettnek a vektor átvitelét. Ahogy a probléma forrása, úgy a megoldás is ugyanaz, mint egybites esetben; visszacsatolást kell létrehoznunk. A 6-34. ábra a vektorszinkronizáló fokozat visszacsatolt változatát mutatja be.

VHDL

```
architecture rtl of vector_sync_handshake is
  signal FF1, FF2, FF3, FF4, FF5, FF6: std_logic;
  signal V1, V2: std_logic_vector (w-1 downto 0);
begin

  process ( clkA, rstA )
  begin
    if ( rstA = '1' ) then
      FF1 <= '0'; FF5 <= '0'; FF6 <= '0';
      V1 <= (others => '0');
    elsif ( rising_edge(clkA) ) then
      if ( strobeA = '1' ) then FF1 <= '1';
      elsif ( FF6 = '1' ) then FF1 <= '0';
      end if;
      FF5 <= FF3;
      FF6 <= FF5;
      V1 <= dataA;
    end if;
  end process;

  busy <= FF6 or FF1;

  process ( clkB, rstB )
  begin
    if ( rstB = '1' ) then
      FF2 <= '0'; FF3 <= '0'; FF4 <= '0';
      V2 <= (others => '0');
    elsif ( rising_edge(clkB) ) then
      FF2 <= FF1;
      FF3 <= FF2;
      FF4 <= FF3;
      if ( strobeB = '1' ) then V2 <= V1;
      end if;
    end if;
  end process;

  strobeB <= not FF4 and FF3;
  dataB <= V2;

end architecture rtl;
```

Verilog

```
module vector_sync_handshake #(...)(...);

  logic FF1, FF2, FF3, FF4, FF5, FF6;
  logic [w-1:0] V1, V2;

  always @ ( posedge clkA, posedge rstA )
  begin
    if ( rstA == 1'b1 ) begin
      FF1 <= 1'b0; FF5 <= 1'b0; FF6 <= 1'b0;
      V1 <= 0;
    end else begin
      if ( strobeA == 1'b1 ) FF1 <= 1'b1;
      else if ( FF6 == 1'b1 ) FF1 <= 1'b0;
      FF5 <= FF3;
      FF6 <= FF5;
      V1 <= dataA;
    end
  end

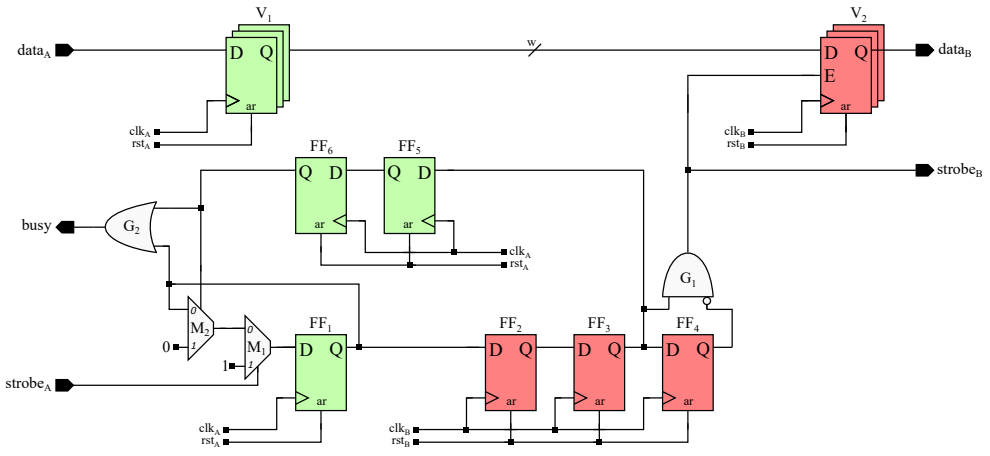
  assign busy = FF6 | FF1;

  always @ ( posedge clkB, posedge rstB )
  begin
    if ( rstB == 1'b1 ) begin
      FF2 <= 1'b0; FF3 <= 1'b0; FF4 <= 1'b0;
      V2 <= 0;
    end else begin
      FF2 <= FF1;
      FF3 <= FF2;
      FF4 <= FF3;
      if ( strobeB == 1'b1 ) V2 <= V1;
    end
  end

  assign strobeB = ~FF4 & FF3;
  assign dataB = V2;

endmodule
```

Áramköri struktúra



6-34. ábra. *Visszacsatolt vektorszinkronizáló fokozat*

A visszacsatolt vektorszinkronizáló tehát a visszacsatolt impulzusszinkronizáló és a visszacsatolatlan vektorszinkronizáló ötvözése. Az adó elhelyezi az átvivendő vektort a V_1 flip-flopokban, majd egyetlen, clk_A órajeltartománybeli pozitív impulzust generál a $strobe_A$ bemeneten. Ennek hatására a busy jelzés egybe vált, jelezve az adó oldalt vezérlő áramkör felé, hogy a vektor átvitele még folyamatban van. A jelzés a clk_B órajeltartományba átérve engedélyezi az immár stabil V_1 flip-flopok mintavételezését, az adat tehát hibátlanul átkerül a clk_B órajeltartománybeli V_2 flip-flopokba. Az új adat jelenlétét jelző strobe jel a clk_B órajeltartományból az FF_5, FF_6 flip-flopokon keresztül visszajut a clk_A órajeltartományba, automatikusan nullázza az FF_1 flip-flopot és aktívan tartja az adó-oldali busy jelzést. Amikor az FF_1 flip-flopba írt nulla érték is átért a clk_B órajeltartományba, majd onnan visszajutott ismét a clk_A órajeltartományba, akkor az adó oldali busy jelzés megszűnik, a fokozat tehát készen áll újabb vektor átvitelére.

Aszinkron dual-clock FIFO szinkronizáló • Vektorokat aszinkron órajeltartományok között legkényelmesebben – bár az előzőeknél kétségkívül némileg összetettebb és erőforrásigényesebb módon – aszinkron FIFO használatával tudunk átvinni. Az aszinkron kifejezés itt arra utal, hogy a FIFO push és pop interfésze két, egymáshoz képest aszinkron órajeltartomány része. Ennek szükséges feltétele az, hogy rendelkezésünkre álljon olyan dual-port memóriamodul, amely két különálló órajelbemenettel rendelkezik, és amelyben a beírás és kiolvasás egymástól teljesen függetlenül tud végbemenni. Ilyen memóriamodulok természetesen léteznek, az FPGA-k blokk RAM moduljai jellemzően képesek ennek a működésnek a megvalósítására. Ha egy ilyen memóriamodul felhasználásával készítünk a 4-55. ábrán láthatóhoz hasonló módon FIFO-t, akkor megtehetjük, hogy a belső RAM címzését végző, write pointer és read pointer logikákat rendre a push és a pop interfész órajeleivel hajtjuk meg, így a két interfész egymástól független lesz. Van azonban egy apró bökkenő, nevezetesen, hogy a full és empty állapotjelzők nem állíthatók elő csak az egyik pointer ismeretében, mindkettőhöz szükség van a write és read pointerre egyaránt. Ezeket a pointereket tehát a RAM modult kiegészítő FIFO logikán belül biztosan át kell vinnünk az egyik órajeltartományból a másikba. Ezek a pointerek önmagukban is vektorok, tehát vektorok átvitelére alkalmas szinkronizáló fokozatot kell beépítenünk.

Készítsünk el egy generikus, aszinkron FIFO modult a fenti alapelv szerint. A 6-35. ábra a modul interfészdeklarációját mutatja be.

VHDL

```
entity async_fifo is
  generic (
    aw:               integer range 3 to 16;
    dw:               integer range 4 to 32
  );

  port (
    wclk, rclk:  in  std_logic;
    rst:         in  std_logic;
    full, empty: out std_logic;
    push, pop:   in  std_logic;
    dout:        out std_logic_vector (dw-1 downto 0);
    din:         in  std_logic_vector (dw-1 downto 0)
  );
end entity async_fifo;
```

Verilog

```
module async_fifo
#(
  parameter aw,
  parameter dw
)(
  input   logic wclk, rclk,
  input   logic rst,
  output  logic full, empty,
  input   logic push, pop,
  output  logic [dw-1:0] dout,
  input   logic [dw-1:0] din
);
```

6-35. ábra. *Az aszinkron FIFO modul interfésze*

A modul azáltal lesz generikus, újrafelhasználható, hogy a FIFO tároló mélysége és a benne tárolt bejegyzések bitszélessége szintézis-paraméter. Valósítsuk meg először a FIFO alapfunkciót, majd ezután vizsgáljuk meg, hogy hová kell beépítenünk a megfelelő órajelszinkronizáló fokozatokat.

A modul tároló funkcióját a 4-54. ábrán bemutatott példával megegyező módon valósítjuk meg, azonban a write pointer és read pointer logikák az ott látottaktól kissé el fognak térni. A write pointer és a read pointer egy-egy számláló, amelyeknek bitszélessége a belső memória címbuszánál eggyel nagyobb. Az állapotjelzők előállítása ekkor a következő logika alapján valósulhat meg:

- **FULL**: A FIFO tele van, ha a write pointer és a read pointer legfelső bitjei különbözőek, alsó bitjeik viszont azonosak.
- **EMPTY**: A FIFO üres, ha a write pointer és a read pointer minden bitje megegyezik.

A 6-36. ábra a FIFO push-logikáját leíró kódrészleteket, és az általuk definiált áramköri szerkezetet mutatja be.

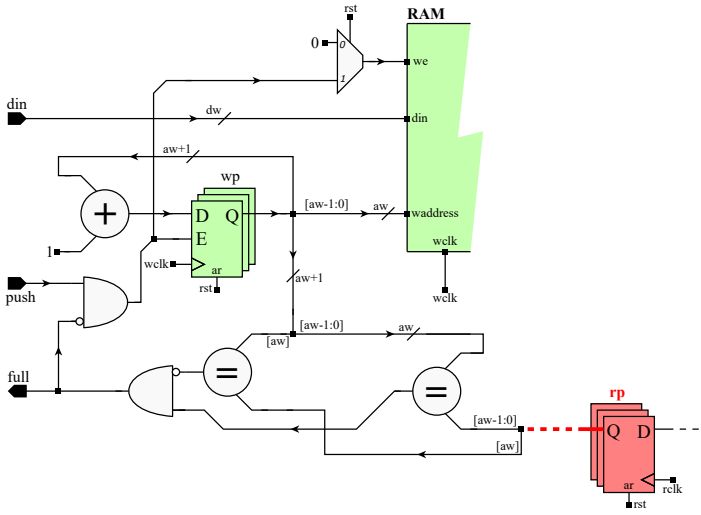
VHDL

```
process ( wclk, rst )
begin
  if ( rst = '1' ) then wp <= (others => '0');
  elsif ( rising_edge(wclk) ) then
    if ( push = '1' and full = '0' ) then
      content(to_integer(wp(aw-1 downto 0))) <= din;
      wp <= wp + 1;
    end if;
  end if;
end process;
full <= '1' when (wp(aw-1 downto 0) = rp(aw-1 downto 0))
                and (wp(aw) /= rp(aw)) else '0';
```

Verilog

```
always @ ( posedge wclk, posedge rst )
begin
  if ( rst == 1'b1 ) begin wp <= 0;
  end else begin
    if ( push == 1'b1 && full == 1'b0 ) begin
      content[wp[aw-1:0]] <= din;
      wp <= wp + 1;
    end
  end
end
assign full = ((wp[aw-1:0] == rp[aw-1:0])
              && (wp[aw] != rp[aw])) ?
              1'b1 : 1'b0;
```

Áramköri struktúra



6-36. ábra. *A FIFO push-logikája*

A pop-logika felépítése hasonló, ezt a 6-37. ábra szemlélteti.

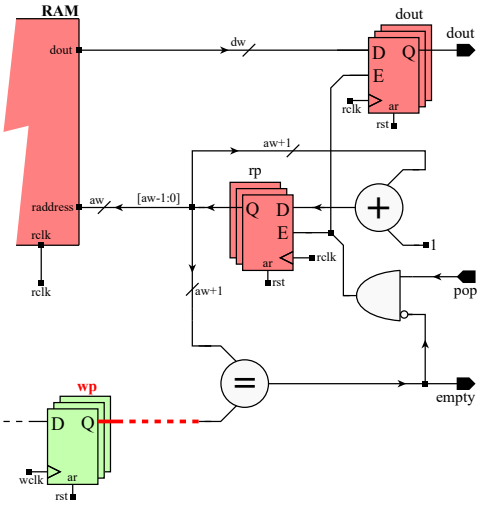
VHDL

```
process ( rclk, rst )
begin
  if ( rst = '1' ) then
    rp <= (others => '0');
    dout <= (others => '0');
  elsif ( rising_edge(rclk) ) then
    if ( pop = '1' and empty = '0' ) then
      dout <= content(to_integer(rp(aw-1 downto 0)));
      rp <= rp + 1;
    end if;
  end if;
end process;
empty <= '1' when (wp = rp) else '0';
```

Verilog

```
always @ ( posedge rclk, posedge rst )
begin
  if ( rst == 1'b1 ) begin
    rp <= 0;
    dout <= 0;
  end else begin
    if ( pop == 1'b1 && empty == 1'b0 ) begin
      dout <= content[rp[aw-1:0]];
      rp <= rp + 1;
    end
  end
end
assign empty = (wp == rp) ? 1'b1 : 1'b0;
```

Áramköri struktúra



6-37. ábra. *A FIFO pop-logikája*

A kapcsolási sémákon világosan látszik, hogy a full és empty állapotjelzők előállításához a write és read pointerek aktuális értékére egyaránt szükség van, e két regiszter azonban nem ugyanabban az órajeltartományban van. A write pointert az `rclk`, a read pointert pedig a `wclk` órajelhez kell szinkronizálnunk, mielőtt bármilyen logikát az adott órajeltartományban meghajtunk velük. Kihasználjuk e két pointernek azt a tulajdonságát, hogy valójában felfelé számlálás funkciót látnak el, tehát ha változás történik rajtuk, akkor új értékük bináris számként értelmezve biztosan eggyel nagyobb lesz az előző értéküknél. Ahogy a 6-32. ábrán is láthattuk, számlálók átvitelére alkalmazhatjuk a Gray-szinkronizáló fokozatot, ezért a push-logikát a 6-38. ábrának megfelelően egészítjük ki, felhasználva a 6-32. ábrán bemutatott Gray-enkóder és dekóder áramkörökhöz készített `gray_encoder()` és `gray_decoder()` függvényeket.

Hasonlóan járunk el a pop-logika esetén is (lásd 6-39. ábra).

VHDL

```
process ( wclk, rst )
begin
  if ( rst = '1' ) then
    wp <= (others => '0');
    wp_gray <= (others => '0');
  elsif ( rising_edge(wclk) ) then
    if ( push = '1' and full = '0' ) then
      content(to_integer(wp(aw-1 downto 0))) <= din;
      wp <= wp + 1;
    end if;
    wp_gray <= gray_encoder(wp);
  end if;
end process;
full <= '1' when (wp(aw-1 downto 0) =
                  rp_wclk(aw-1 downto 0))
                  and
                  (wp(aw) /= rp_wclk(aw)) else '0';

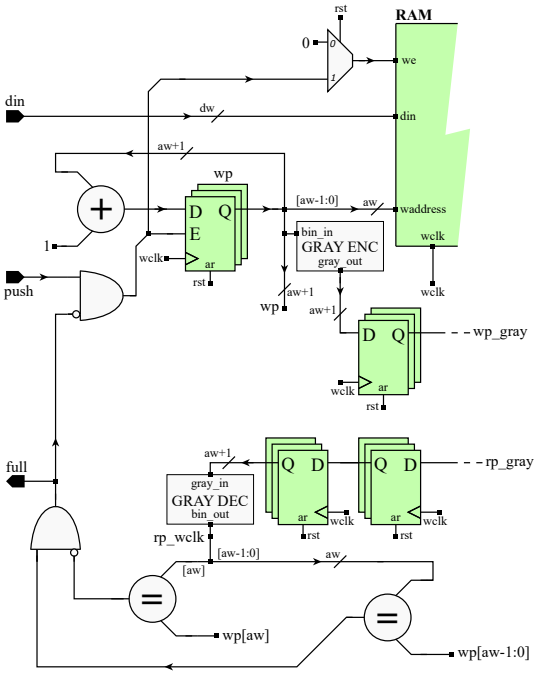
process ( wclk, rst )
begin
  if ( rst = '1' ) then
    rp_gray_d <= (others => '0');
    rp_gray_dd <= (others => '0');
  elsif ( rising_edge(wclk) ) then
    rp_gray_d <= rp_gray;
    rp_gray_dd <= rp_gray_d;
  end if;
end process;
rp_wclk <= gray_decoder(rp_gray_dd);
```

Verilog

```
always @ ( posedge wclk, posedge rst )
begin
  if ( rst == 1'b1 ) begin
    wp <= 0;
    wp_gray <= 0;
  end else begin
    if ( push == 1'b1 && full == 1'b0 ) begin
      content[wp[aw-1:0]] <= din;
      wp <= wp + 1;
    end
    wp_gray <= gray_encoder(wp);
  end
end
assign full = ((wp[aw-1:0] == rp_wclk[aw-1:0])
               &&
               (wp[aw] != rp_wclk[aw])) ?
               1'b1 : 1'b0;

always @ ( posedge wclk, posedge rst )
begin
  if ( rst == 1'b1 ) begin
    rp_gray_d <= 0;
    rp_gray_dd <= 0;
  end else begin
    rp_gray_d <= rp_gray;
    rp_gray_dd <= rp_gray_d;
  end
end
assign rp_wclk = gray_decoder(rp_gray_dd);
```

Áramköri struktúra



6-38. ábra. *A FIFO push-logikája Gray-szinkronizáló fokozattal kiegészítve*

VHDL

```
process ( rclk, rst )
begin
  if ( rst = '1' ) then
    rp <= (others => '0');
    rp_gray <= (others => '0');
    dout <= (others => '0');
  elsif ( rising_edge(rclk) ) then
    if ( pop = '1' and empty = '0' ) then
      dout <= content(to_integer(rp(aw-1 downto 0)));
      rp <= rp + 1;
    end if;
    rp_gray <= gray_encoder(rp);
  end if;
end process;
empty <= '1' when (wp_rclk = rp) else '0';

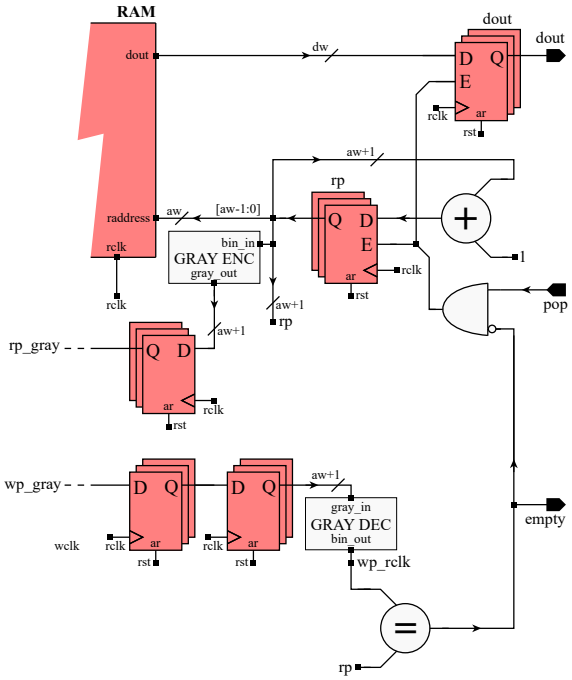
process ( rclk, rst )
begin
  if ( rst = '1' ) then
    wp_gray_d <= (others => '0');
    wp_gray_dd <= (others => '0');
  elsif ( rising_edge(rclk) ) then
    wp_gray_d <= wp_gray;
    wp_gray_dd <= wp_gray_d;
  end if;
end process;
wp_rclk <= gray_decoder(wp_gray_dd);
```

Verilog

```
always @ ( posedge rclk, posedge rst )
begin
  if ( rst == 1'b1 ) begin
    rp <= 0;
    rp_gray <= 0;
    dout <= 0;
  end else begin
    if ( pop == 1'b1 && empty == 1'b0 ) begin
      dout <= content[rp[aw-1:0]];
      rp <= rp + 1;
    end
    rp_gray <= gray_encoder(rp);
  end
end
assign empty = (wp_rclk == rp) ? 1'b1 : 1'b0;

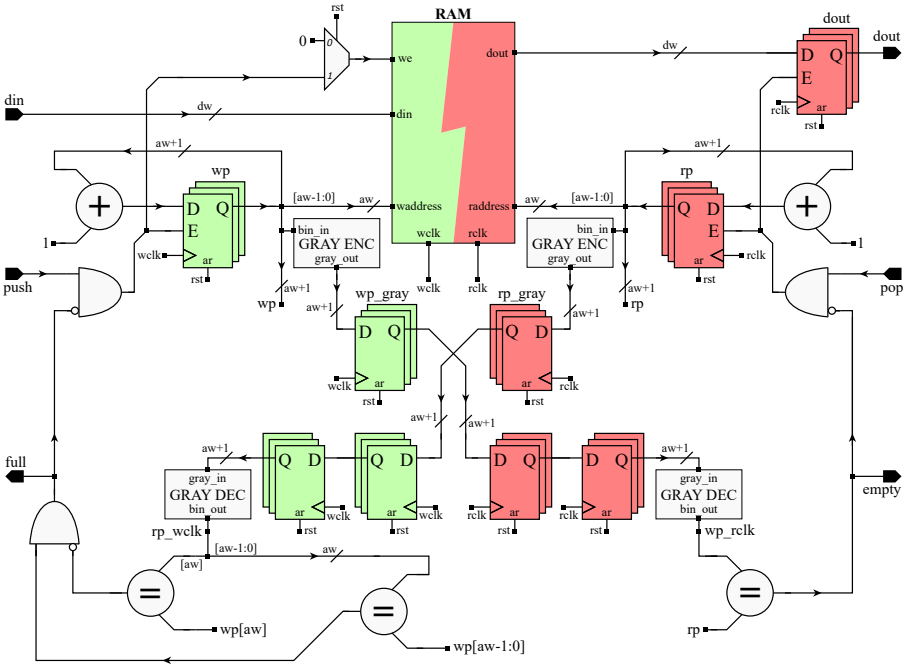
always @ ( posedge wclk, posedge rst )
begin
  if ( rst == 1'b1 ) begin
    wp_gray_d <= 0;
    wp_gray_dd <= 0;
  end else begin
    wp_gray_d <= wp_gray;
    wp_gray_dd <= wp_gray_d;
  end
end
assign wp_rclk = gray_decoder(wp_gray_dd);
```

Áramköri struktúra



6-39. ábra. *A FIFO pop-logikája Gray-szinkronizáló fokozattal kiegészítve*

A 6-40. ábra a FIFO modul teljes áramköri sémáját mutatja be. Az ábrán jól kivehető a két aszinkron órajeltartomány határa, amelyen a write és read pointerek értékét a Gray-szinkronizáló fokozattal vittük át.



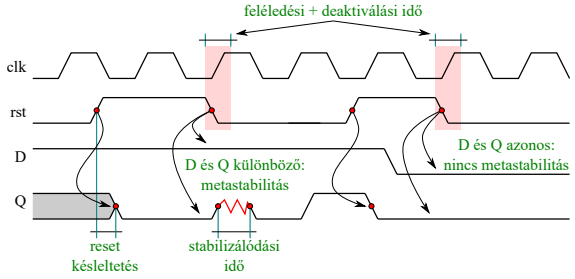
6-40. ábra. *Az aszinkron FIFO modul teljes áramköri sémája*

Az aszinkron dual-clock FIFO szinkronizáló fokozat drága megoldásnak tűnhet a korábban bemutatott, visszacsatolt vektorszinkronizáló fokozathoz képest, ugyanakkor funkció szempontjából is túlmutat rajta. Nemcsak az órajeltartományok közötti szinkronizációt oldja meg, hanem – ahogy egy FIFO általában – a külön órajeltartományban működő almodulok adatfeldolgozási sebességeinek különbségét is képes – a tárolt bejegyzések számától függő mértékben – kiegyenlíteni. A visszacsatolt vektorszinkronizáló fokozat erre nem alkalmas, hiszen ott az adó oldalnak – a visszacsatolás által – csak arról van információja, hogy maga a szinkronizáció mikor történt meg, de azt nem tudhatja, hogy az előzőleg átvitt vektort a clk_B órajeltartományban működő almodul érdemben feldolgozta-e. Aszinkron, dual-clock FIFO-t alkalmazva, azt az egyszerű szabályt betartva, hogy aktív full jelzés esetén az adó oldal meg sem próbál új adatot írni a FIFO-ba, az adatvesztés elkerülhető.

6.3.2 Reset-szinkronizáció

Az időzítésvizsgálatokról és az órajelszinkronizáló áramkörökről elmondottak alapján jól látható, hogy a digitális áramkörökben központi szerepet betöltő D flip-flopok beállítási és tartási idejére vonatkozó időzítési kritériumokat igen komolyan kell venni, ha megbízhatóan, determinisztikusan működő áramköröket akarunk tervezni. Időzítési kritériumot azonban – ahogy az a 6-4. ábrán is látszik – nemcsak a D bemenetre, hanem a reset bemenetre vonatkozóan is meg lehet fogalmazni. A feléledési idő *(reset recovery time)*

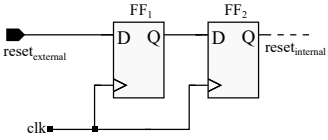
tulajdonképpen a reset bemeneten értelmezett beállítási idő, a deaktiválási idő *(reset removal time)* pedig a reset bemeneten értelmezett tartási idő. A reset bemenet a D bemenettől annyiban különbözik, hogy időzítési szempontból vagy csak a felfutása vagy csak a lefutása érdekes, attól függően, hogy negált vagy ponált resetet használunk, vagyis hogy melyik logikai szint jelenti a reset aktiválását és melyik a deaktiválását. Mivel ez egy áramkörre jellemző tulajdonság, ezért a konkrét logikai szintek, felfutó vagy lefutó élek helyett a reset aktiválásáról vagy meghúzásáról *(assert)* és deaktiválásáról vagy elengedéséről *(release)* beszélünk. Ezt a terminológiát használva azt mondhatjuk, hogy időzítési szempontból a reset meghúzása nem különösebben érdekes; a meghúzás után némi késleltetéssel (reset késleltetés) a flip-flop kimenete alapállapotba áll. A reset elengedése azonban időzítés szempontjából kritikus, ha ugyanis a reset bemenet a feléledési vagy deaktiválási időkre vonatkozó időzítési kritériumokat megsértve, az órajel felfutó éléhez túl közel változik meg, és ebben a kritikus pillanatban a flip-flop bemenete és kimenete eltérő logikai szinten van, akkor a flip-flop kimenete ugyancsak metastabil állapotba kerülhet. Ezt a folyamatot a 6-41. ábra egy aszinkron, magas-aktív resettel ellátott D flip-flop példáján keresztül mutatja be.



6-41. ábra. *Metastabilitás kialakulása aszinkron reset elengedésének hatására*

A reset bemenetet tipikusan egy külső eszköz, például nyomógomb, reset IC vagy egy mikrokontroller hajtja meg, amelynek működése nincs a mi áramkörünk órajeléhez szinkronizálva, a resetet tehát egy teljesen aszinkron bemenetnek kell tekintenünk. A metastabilitás elkerülése érdekében tehát a reset jel változásait a lokális órajelhez kell szinkronizálnunk. Ezt a feladatot látják el az ún. reset-szinkronizáló áramkörök. A reset elengedését mindenképpen szinkronizálni kell, a meghúzás szinkronizálása nem kritikus. Ezek alapján kétféle reset-szinkronizáló megoldást különböztethetünk meg:

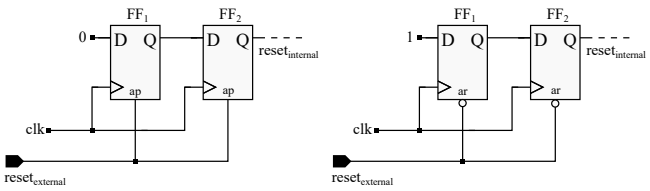
- **Szinkron meghúzott, szinkron elengedett reset**: A 6-42. ábra olyan reset-szinkronizáló áramkört mutat be, amely a külső resetnek mind a meghúzását, mind az elengedését szinkronizálja a lokális órajelhez.



6-42. ábra. *Szinkron meghúzott, szinkron elengedett reset előállítása*

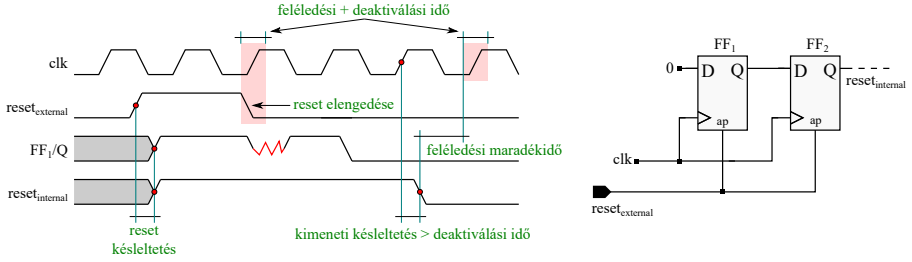
A teljesen szinkron resetet előállító áramkör lényegében egy double-flop szinkronizáló fokozat, amelynek bemenetére a külső reset jelet kapcsoljuk. Magas-aktív és alacsony-aktív belső reset előállítására egyaránt alkalmas.

- **Aszinkron meghúzott, szinkron elengedett reset**: A teljesen szinkron reset működéséhez aktív órajelforrásra van szükség. A 6-43. ábra olyan megoldásokat mus tat be, amelyek a reset meghúzását nem, csak annak elengedését szinkronizálják a lokális órajelhez, vagyis áramkörünk flip-flopjai akkor is alapállapotba állíthatók vele, ha az órajel még nincs jelen. Az ábra bal oldalán a magas-aktív, jobb oldalán az alacsony-aktív kivitel látható. A flip-flopok *ap* bemenete az aszinkron preset bemenet, amelynek meghúzása a kimenet magas szintre állását idézi elő, az *ar* pedig az aszinkron reset, ennek meghúzása a kimenet nullázását idézi elő.



6-43. ábra. *Aszinkron meghúzott, szinkron elengedett reset előállítása*

A fenti reset-szinkronizáló áramkörök kimenetét az áramkörünk többi flip-flopjának aszinkron reset bemenetére kapcsolva elkerülhető a reset elengedésekor kialakuló metastabilitás. A 6-44. ábra az aszinkron meghúzott, szinkron elengedett, magas-aktív reset-szinkk ronizáló működésének részleteit szemlélteti.



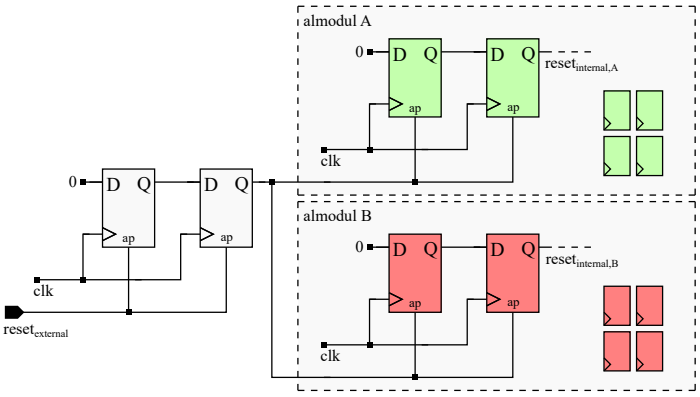
6-44. ábra. *Az aszinkron meghúzott, szinkron elengedett, magas-aktív reset-szinkronizáló működése*

Az ábrán a következő folyamatot figyelhetjük meg: A $reset_{external}$ jel felfutása hatására az FF_1 és FF_2 flip-flopok kimenete logikai magas szintre áll be az órajel állapotától teljesen függetlenül. A $reset_{external}$ jelet az órajel felfutó éléhez közel, az FF_1 és FF_2 flip-flopok feléledési idejére vonatkozó időzítési kritérium megsértésével engedjük el. Az FF_1 flip-flop bemenetén ekkor alacsony, kimenetén viszont magas szint van, ezért ez a flip-flop bizonyos valószínűséggel metastabil állapotba kerülhet. Ugyanebben a pillanatban az FF_2 flip-flop bemenetén és kimenetén egyaránt logikai magas szint van, vagyis az FF_2 flip-flop nem kerülhet

metastabil állapotba. A szinkronizáló fokozat tehát egy egyszerű double-flop fokozathoz hasonlóan lokalizálja a metastabilitást. Miután az FF_1 flip-flopnak sikerült a bemenetére kapcsolt konstans nulla értéket mintavételeznie és kimenete logikai alacsony szinten stabilizálódott, az órajel következő felfutó éle hatására az FF_2 flip-flop kimenetén is megjelenik a logikai alacsony szint, a $reset_{internal}$ jelen tehát a reset elengedése már a lokális órajelhez szinkronizáltan jelenik meg.

A meghajtott flip-flopok feléledési és deaktiválási időire vonatkozó időzítési kritériumok teljesülése a korábban tárgyalt időzítésvizsgálati módszerekkel ellenőrizhetők. Ahhoz, hogy a szinkronizáló fokozat által meghajtott flip-flopok deaktiválási ideje ne sérüljön, arra van szükség, hogy az FF_2 flip-flop kimeneti késleltetésének és a reset jelet terjesztő vezetékezés késleltetésének az összege a meghajtott flip-flopok deaktiválási idejét meghaladja. Mivel a flip-flopok kimeneti késleltetése rendszerint már önmagában is nagyobb, mint a deaktiválási idő, ezért ennek a kritériumnak a teljesülése biztosítottnak tekinthető. Ahhoz, hogy a meghajtott flip-flopok feléledési idejére vonatkozó kritérium se sérüljön, az kell, hogy a $reset_{internal}$ jel még az órajel következő felfutó éle előtt, a feléledési időablak előtt megérkezzen minden egyes flip-flop aszinkron reset bemenetére, tehát az ábrán feléledési maradékidőként *(recovery slack)* jelölt, a reset-hálózat vezetékezésének késleltetésétől függő időtartam nullánál nagyobb legyen.

A reset jel az órajelhez hasonlóan ún. globális jel, az áramkörünk minden flip-flopjához el kell juttatni, ráadásul minimális elcsúszással. Ez úgy valósítható meg, hogy a reset jelet az órajelfákhoz hasonló szerkezetű, nagy meghajtóképességű buffereket és de-skew áramköröket tartalmazó reset-hálózaton keresztül vezetjük el a chip különböző részeibe. A reset jel forrásának terhelése csökkenthető olyan módon, hogy az áramkörünk egyes részleteihez, almoduljaihoz saját reset-szinkronizáló fokozatot rendelünk, amelyeket egy globális reset-szinkronizálóval hajtunk meg (lásd 6-45. ábra).



6-45. ábra. *Reset fan-out kiegyenlítés*

Ezt a megoldást reset fan-out kiegyenlítésnek nevezzük. Az így létrehozott, hierarchikus reset-szinkronizáló struktúrát akár több szintre is bonthatjuk, azonban mindig szem előtt

kell tartanunk, hogy a különböző hierarchiaszinteken előállított, belső reset jeleken eltérő késleltetéssel jelenik meg a külső reset jel elengedése. Ha tehát két részáramkört eltérő hiearchiaszinten előállított, belső reset jelekkel hajtunk meg, akkor e két részáramkör nem egyazon időpontban éled fel a külső reset elengedése után, ami funkcionális hibákhoz vezethet.



7

A digitális áramkörök megvalósításának technológiái

A mai modern integrált áramkörök esetén a kapcsolástechnikát tekintve kedvező fogyasztási és integrálhatósági tulajdonságai miatt a CMOS technológia dominál. Különösen igaz ez a digitális rendszerekre, amelyeket csak egyedi igények esetén valósítanak meg a CMOS-tól eltérő technológián. Nagyfrekvenciás alkalmazásokban elterjedtek a source-csatolt logikai áramkörök (*Source-Coupled Logic*, SCL), nagy meghajtóképesség-igény esetén – pl. buszmeghajtó áramkörökben – pedig a bipoláris és MOS tranzisztorokat egyaránt tartalmazó BiCMOS *(Bipolar CMOS)* kapcsolások, de a digitális rendszerek túlnyomó része statikus vagy dinamikus CMOS áramkörökből épül fel. A „megvalósítás technológiáján" nemcsak az alkalmazott kapcsolástechnikai megoldásokat érthetjük, hanem azt, hogy hogyan használjuk a rendelkezésre álló legalapvetőbb, eszközszintű erőforrásokat (tranzisztor, ellenállás, kapacitás, esetleg induktivitás, vezetékek) a kívánt funkció megvalósítására. Ilyen szempontból kétféle megközelítést különböztetünk meg; az alkalmazás-specifikus integrált áramkört (*Application-Specific Integrated Circuit*, ASIC) és az újrakonfigurálható hardvert. ASIC technológiáról beszélünk akkor, ha az áramkör funkcióját az eszközszintű erőforrások közvetlen felhasználásával, azok példányosításával és összekapcsolásával valósítjuk meg. Ezzel ellentétben az újrakonfigurálható hardver kifejezést arra használjuk, amikor az eszközszintű erőforrásokat egy általános célú, rugalmasan újrakonfigurálható, összetettebb erőforrás-készlet (logikai, tároló és huzalozási erőforrások) kialakítására használjuk fel. Ennek elemeiből nagyszámú példányt előre legyártva az áramkör konkrét funkcionalitása a gyártást követő konfigurációs lépésben áll elő. A különböző programozható logikai eszközök (*Programmable Logic Device*, PLD), mint a PLA-k *(Programmable Logic Array)*, CPLD-k *(Complex Programmable Logic Device)*, FPGA-k *(Field-Programmable Gate Array)* tulajdonképpen ezeknek az előre legyártott újrakonfigurálható erőforrás-készleteknek a különböző változatai, általános célú, tokozott chip formájában.

Egyazon gyártástechnológián – pl. AMS 0,35 μm, TSMC 28 nm HKMG *(High-K Metal Gate)*, Intel 14 nm Tri-Gate stb. – és kapcsolástechnikával (pl. SCL, CMOS stb.) megvalósítva egy adott funkcionalitás ASIC és újrakonfigurálható hardver verzióját, a kapott áramkörök tulajdonságai jelentősen eltérnek. E szempont alapján az ASIC kategória tovább osztható az áramkörök tervezésének módja szerint. Ez a felosztás a full-custom

ASIC és a semi-custom[61] (standard cellás) ASIC fogalmakhoz vezet, amelyek jellemzőit – és összehasonlításukat egymással, valamint az újrakonfigurálható hardverre épülő megközelítéssel – a következőképpen foglalhatjuk össze:

Full-custom ASIC ⬩ A CAD rendszereket legkevésbé alkalmazó tervezési módszer, amely során az áramkör logikai és fizikai tervezése – beleértve a floor-plan és a teljes layout elkészítését – kézi úton valósul meg. Természetesen ez a megközelítés sem nélkülözi teljes mértékben a számítógépes módszereket, de az itt alkalmazott eszközök nem a tervezést magát, hanem a kézzel megtervezett áramkör ellenőrzését, validálását segítik. Ilyen eszközök pl. a tervezésiszabály-ellenőrzés (*Design Rule Check*, DRC), layout-visszafejtés és összehasonlítás a kapcsolási/logikai sémával (*Layout vs. Schematic*, LVS).

- **Előnyök**: Mivel az így előállított áramkörök a megvalósítás alapjául szolgáló gyártástechnológiát teljes mértékben ki tudják használni, erőforrás-felhasználás, számítási teljesítmény és fogyasztás szempontjából is felülmúlják azokat a megvalósításokat, amelyeket valamilyen automatizált módszerrel terveztek.
- **Hátrányok**: A fejlesztési folyamat rendkívül időigényes, hibalehetőségekkel terhelt, a végtermék, vagyis az áramkör terve pedig technológiafüggő.
- **Példa**: Az analóg áramköröket, valamint az A/D és D/A átalakítókat tipikusan full-custom módszerrel tervezzük. Ezt a megközelítést alkalmazzuk nagysebességű kommunikációs interfészek fizikai rétegének, a külvilággal kapcsolatot tartó PAD áramköröknek (jelszint-illesztés, slew-rate vezérlés, ESD védelem stb.) és az újrakonfigurálható hardverelemeknek a tervezésére is.

Semi-custom (standard cellás) ASIC ⬩ Digitális áramkörök megvalósítására használt módszer, amelyben egy RTL HDL modellből automatizált szintézissel állítjuk elő a kapuszintű reprezentációt, felhasználva a gyártástechnológiai adatbázis részeként rendelkezésre álló cellakönyvtár elemeit (a cellakönyvtár full-custom módszerrel készül, a teljes funkció ebből automatizált úton, innen az elnevezés). A fizikai tervezés nagy része – floor-plan tervezés, cellák elhelyezése, huzalozás, PAD áramkörök elhelyezése, táp- és földsínek elhelyezése stb. – szintén automatizált módszerekkel történik, bár ezen a ponton már nagy jelentősége van a kézi úton végzett apróbb módosításoknak, finomításoknak.

- **Előnyök**: A tervezési folyamat az automatizált módszerek alkalmazása miatt lényegesen gyorsabb, mint full-custom esetben.
- **Hátrányok**: A semi-custom módszerrel előállított áramkörök időzítési, erőforrásigénybeli és fogyasztási tulajdonságai általában rosszabbak, mint az ugyanazon funkciót megvalósító, full-custom módszerrel készített áramkörökéi.

[61] A semi-custom fogalmat itt a tervezésre vonatkoztatjuk; egy összetett funkcionalitás megvalósításához előre megtervezett alapelemkészletet, szabványos cellakönyvtárat használunk. Maga a semi-custom kifejezés a gyártástechnológiai terminológiában is használatos, ebben az esetben a gyártási lépések összességének részben minden áramkörnél közös, részben áramkör-specifikus részfolyamatokra bontását jelenti.

- **Példa**: A SoC áramkörök periféria- és buszvezérlő áramkörei, alkalmazás-specifikus makrocellái – a szélsőséges időzítési, fogyasztási vagy erőforrásigényre vonatkozó követelményeknek megfelelni kénytelen egységektől eltekintve – semi-custom módszerrel készülnek.

Újrakonfigurálható hardver • Nagyfokú rugalmasságot, pl. gyártás utáni, a funkcionalitás nagy részét érintő változtathatóságot igénylő esetekben az alkalmazás-specifikus funkciót újrakonfigurálható hardverelemek felhasználásával valósítjuk meg.

- **Előnyök**: Gyártás utáni módosíthatóság, a funkcionalitás egésze kicserélhető.
- **Hátrányok**: A konfigurálhatóság ára a magas fogyasztás és a viszonylag alacsony számítási teljesítmény. Ezek oka a rugalmasságot biztosító összeköttetéshálózat nagy mennyiségű vezetékezésének kapacitása, valamint a kapcsoló tranzisztorok által okozott többlet késleltetés.
- **Példa**: A gyorsan változó, egyre kifinomultabb szabványoknak – pl. video, audio dekódolás, vezeték nélküli kommunikáció, titkosítás – való megfelelés az SoC áramkörök központi mikroprocesszorain futó firmware módosításával egyes esetekben megoldható, de ha a módosítandó funkció a nagyfokú párhuzamosság igénye miatt szoftveres úton nem valósítható meg, akkor a nagy számítási teljesítménynek és rugalmasságnak effajta együttese csak újrakonfigurálható hardverrel érhető el.

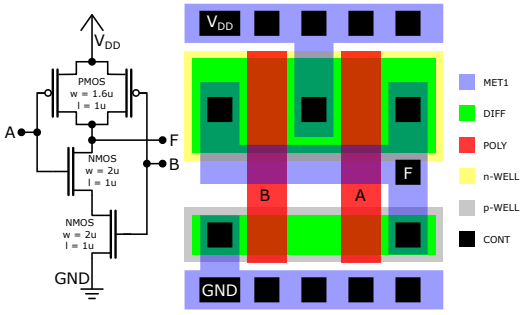
A következő alfejezetekben a standard cellás ASIC és az FPGA áramkörök felépítésével foglalkozunk részletesen.

7.1 Standard cellás integrált áramkörök

Standard cellás ASIC alatt azokat az integrált áramköröket értjük, amelyek egy kapcsolási séma és layout-rajzolat szintjén előre megtervezett, optimalizált, időzítés és fogyasztás alapján karakterizált alapelem-készlet, az ún. cellakönyvtár elemeinek megfelelő összekapcsolásával készülnek, az esetek többségében automatizált RTL szintézis és fizikai szintézis útján.

A standard cellák egy-egy alapvető logikai funkciót (logikai kapu, flip-flop, multiplexer, RAM cella) megvalósító, áramköri szintű kapcsolási sémával és layout-rajzolattal adott építőelemek, amelyek összekapcsolásával összetettebb funkciójú áramköröket építhetünk. A standard cellák elhelyezését és a köztük lévő összeköttetések létrehozását automatizált úton végzik az e célra kifejlesztett szoftver eszközök. Az automatizált eljárások hatékonyságának növelése érdekében az egyes cellák fizikai tulajdonságaira és egymáshoz viszonyított helyzetükre korlátozásokat vezettek be. A standard cellás áramkörökben a cellák azonos magasságú sorokban, ún. cellasorokban helyezkednek el, amelyek között az összeköttetések számára fenntartott üres sávok, huzalozási csatornák foglalnak helyet. Ebből az következik, hogy az egyes cellák – funkciójuktól és meghajtóképességüktől függetlenül – mind azonos

magasságúak. Továbbá, a cellasorok felső és alsó részén, meghatározott helyen húzódik a csatorna táp- és földsínje, amelynek folytonosságát az adott sorban elhelyezett celláknak nem szabad megszakítaniuk. A táp- és földsínek pozíciója tehát a cellán belül ugyancsak kötött. A 7-1. ábra egy standard cellaként megvalósított NAND kapu kapcsolási sémáját és egy lehetséges layout-rajzolatát mutatja.

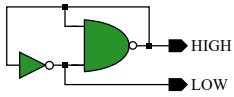


7-1. ábra. *Egy NAND kapu kapcsolási sémája és layout rajzolata*

A standard cellakönyvtár az egyes cellákat sokféle szempontból leírja. Az egyazon cellára vonatkozó, más-más tulajdonságot leíró adathalmazokat a cella nézeteinek nevezzük. A cella egy nézete pl. a 7-1. ábra bal oldalán látható kapcsolási séma, egy másik a jobb oldalon látható layout-rajzolat. További nézetek írják le a cella időzítési tulajdonságait és fogyasztását, a layout parazita összetevőit, a viselkedését leíró szimulációs modelljeit (SPICE, Verilog), valamint a cella által előírt tervezési szabályokat annak érdekében, hogy a cella belső szerkezete és a felhasználó által készített huzalozás ne zavarja egymást. A cellakönyvtárak egy-egy funkcionális egységnek nemcsak egy, hanem egy egész sor változatát tartalmazzák, amelyek elsősorban fizikai méretben, fogyasztásban és időzítési tulajdonságaikban térnek el egymástól. E sokszínűség célja, hogy az ilyen cellák millióiból felépülő, összetett áramkörök optimalizációja során a fenti tulajdonságok tekintetében kompromisszumos megoldások széles skálája álljon rendelkezésre. Nagy meghajtóképességű (nagy méretű) cellákból felépítve az áramkört magas órajel-frekvencia és nagy számítási kapacitás érhető el, de a fizikai méret és a fogyasztás megengedhetetlenül nagyra adódhat. Kis meghajtóképességű (kis méretű) cellákból építkezve az áramkör fogyasztása és mérete kedvező lesz, de a maximálisan megengedhető órajel-frekvencia alacsonyabb. Természetesen a kis és nagy meghajtóképességű cellák egymással tetszőlegesen kombinálhatók egy áramkörön belül, ezért a gyakorlatban az jellemző, hogy az időzítés szempontjából kevésbé jelentős részáramköröket kis meghajtóképességű, a kritikus út elemeit azonban nagy meghajtóképességű cellákból építik fel.

A cellakönyvtár olyan cellákat is tartalmaz(hat), amelyek célja nem a logikai funkció megvalósítása. Ezek a cellák elvileg nem nélkülözhetetlenek, de gyakorlatilag minden cellakönyvtár tartalmazza őket, vagy legalább néhányat közülük. Az áramkörök összetettségének növekedésével és a csíkszélesség csökkenésével jelentőségük egyre nő.

- **TIE cellák**: Ha egy standard cella valamelyik tranzisztorának kapuelektródáját a logikai funkcióból adódóan föld- vagy tápfeszültségre kell kapcsolni, akkor a föld- és tápsínek potenciáljának esetleges ingadozása funkcionális hibákhoz (pl. a flip-flopok metastabil állapotba kerülése) vagy – kis csíkszélesség esetén – a tranzisztor gate-oxidjának átütéséhez vezethet. A táp- és földsínek feszültségingadozása elleni védekezés céljából a standard cellák kapuelektródáit nem közvetlenül, hanem e speciális cellákon keresztül kapcsolják földre vagy tápra. A 7-2. ábra egy olyan TIE cella logikai szintű kapcsolási sémáját mutatja, amely a logikai magas és alacsony szint előállítására egyaránt alkalmas.



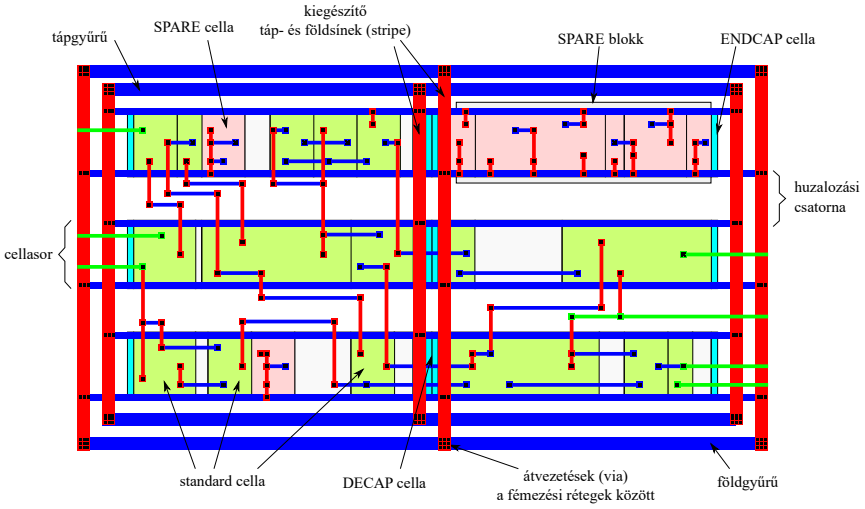
7-2. ábra. *Egy TIE cella logikai szintű kapcsolási sémája*

- **SPARE cellák**: A standard cellák elhelyezése után a cellasorokban általában marad üres hely. Ezeket az üres helyeket funkció nélküli, be nem kötött (földre vagy tápra kapcsolt) standard cellákkal töltik fel annak érdekében, hogy ha gyártás után valamilyen apróbb funkcionális hibára derül fény, akkor e cellák felhasználásával – mindössze a huzalozási rétegekhez tartozó maszkok újragyártásával – az kijavítható legyen. A standard cellák bármelyikét elhelyezhetjük SPARE cellaként az üresen maradt helyekre, de létrehozhatunk külön erre a célra ún. SPARE blokkokat is, amelyek csak ilyen funkció nélküli, „tartalék" cellákat tartalmaznak.
- **DECAP cellák**: A digitális áramkörökben a tápsín terhelése meglehetősen dinamikus. Az órajel felfutó élénél a rengeteg jelváltozás hatására a tápsínen nagy áram folyik, ami a feszültség „beeséséhez" vezethet, ami pedig funkcionális hibákat eredményezhet. A cellasorokban egymástól adott távolságra elhelyezett kapacitások alkalmasak ennek a tápfeszültség-ingadozásnak a kiegyenlítésére.
- **ENDCAP cellák**: A standard cellák önmagukban nem feltétlenül gondoskodnak saját zsebeik megfelelő előfeszítéséről (p-zsebek földpotenciálra, n-zsebek tápfeszültségre a parazita pn-átmenetek záró irányú előfeszítése céljából). Ebben az esetben a cellasorok zsebeit folytonossá kell tenni (lásd FILLER cellák) és azokat az ENDCAP cellákkal megfelelő potenciálra kell kapcsolni.
- **FILLER cellák**: A cellasorok zsebeinek, valamint a táp- és földsíneknek a folytonosságát biztosítják.
- **PAD cellák**: Az áramkör bemenetei és kimenetei PAD cellákon keresztül kapcsolódnak a külvilághoz. A legegyszerűbb PAD cella az ESD-védelmet biztosító áramköröket és a mikrohuzal-kötéshez szükséges fémfelületet tartalmazza.

A standard cellás integrált áramkörök fizikai felépítése logikai szerkezetükhöz hasonlóan hierarchikus, amelyben a hierarchiaszintek száma a megvalósított funkció összetettségétől függ. A szerkezeti felépítés magyarázatához két hierarchiaszintet fogunk feltételezni; a toplevel szintet és az azt felépítő makrocellák szintjét. Példánkban a fizikai terv előállítása

szempontjából minden makrocella ún. *„flat design”*, ami azt jelenti, hogy a logikai tervezés során az adott makrocellát egyetlen toplevel egyedként modelleztük.

A 7-3. ábra egy makrocella fizikai tervét, a chip felülnézeti képét mutatja be.



7-3. ábra. *Egy makrocella standard cellás fizikai terve*

Az ábrán jól megfigyelhetők a standard cellás technológián megvalósított makrocellák fizikai tervének jellegzetességei. A standard cellák és egyéb, speciális funkciójú cellák egyforma magasságú sorokban helyezkednek el, amelyeket a közöttük lévő, a cellákat összekapcsoló vezetékezésnek fenntartott huzalozási csatornák választanak el egymástól. A cellasorok magasságát maga a cellakönyvtár határozza meg (az AMS 350 nm-es cellakönyvtára esetén ez 13 μm), de a számuk, illetve a huzalozási csatornák magassága és a sorok hosszúsága a fizikai tervezés során meghatározandó paraméterek. A cellasorok és huzalozási csatornák alkotta, ún. cellaterületet táp- és földgyűrű veszi körül, amelyekre a cellasorok táp- és földsínjeit a gyűrűk függőleges szélénél csatlakoztatjuk. Nagyméretű, sok cellát tartalmazó makrocellák esetén a cellasorok tápsínjei olyan hosszúra adódhatnak, hogy az órajel felfutó éleinél, amikor az áramfelvétel hirtelen megugrik, akkor a belső cellák tápfeszültsége magának a tápsínnek az ellenállásán fellépő feszültségesés miatt beeshet, ami funkcionális hibához vezethet. E hatás csökkentése érdekében további függőleges vezetékekkel *(stripe)* egészítjük ki a táp- és földhálózatot, hogy csökkentsük annak eredő impedanciáját. A cellák közötti összeköttetéseket vízszintes és függőleges vezetékek valósítják meg. Ennek részben gyártástechnológiai, részben a huzalozás automatizált előállítását végző algoritmusok működésével kapcsolatos, számítástechnikai oka van:

- A szilícium hordozó kristályszerkezetének irányítottsága és a különböző fémezési rétegek maszk-alakzatainak orientációja szorosan összefügg egymással. A vezetékezést kialakító marási eljárások eredményeként adódó, háromdimenziós struktúrák jellege erősen függ a vezetékek éleinek orientációjától. A hordozó kristályrácsa és

a layout-rajzolatok orientációjának viszonyát ezért szigorúan kézben kell tartani. A függőleges és vízszintes vezetékek használatával biztosítható legjobban a chip különböző pontjain a vezetékszakaszok egyformasága.

- A cellák közötti összeköttetéseket ún. huzalozó algoritmusok *(router)* hozzák létre. Ezek az algoritmusok hasonlóak a nyomtatott huzalozású lemezek vezetékezésének megtervezésekor is használt algoritmusokhoz, azzal a különbséggel, hogy itt az áramkörök mérete, az összekötendő pontok száma sokkal nagyobb. A tetszőleges orientációjú vezetékezés kialakítására alkalmas algoritmusok egyszerűen nem tudnák kellően gyorsan megoldani a feladatot, ezért itt olyan algoritmusokat használunk, amelyek csak függőleges és vízszintes vezetékezést képesek kezelni. Bár a ferde vezetékezés használatával elvileg jobb eredményt kaphatnánk, de a megoldás, a megfelelő vezetékhálózat kialakítása túl sokáig tartana.

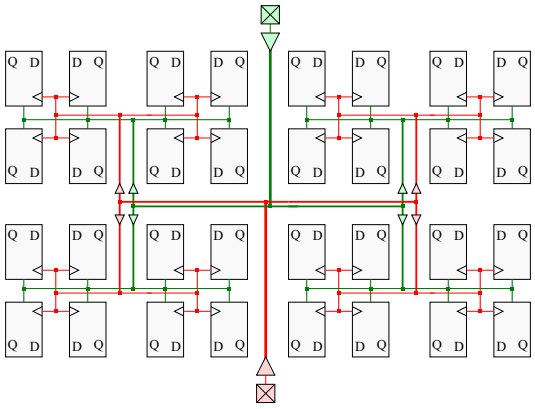
Az ábrán az is látható, hogy a cellák nincsenek szorosan egymás mellé helyezve. Bár nyilván a közvetlenül egymás mellé lerakott cellákkal az áramkör mérete kisebb lenne, ugyanakkor a cellák hozzávezetései nagyon közel kerülnének egymáshoz, ami a vezetékezés előállítását nagyon megnehezítené, adott esetben ellehetetlenítené. A cellák pozícióját meghatározó, ún. elhelyező *(placer)* algoritmusnak százalékos formában adhatjuk meg, hogy egy adott területet mennyire töltsön fel cellákkal.

Nagyméretű, sok cellát tartalmazó makrocellák esetén a nagy terhelésű, globális jelek terjesztésére külön hangsúlyt kell fektetni a vezetékezés megtervezésekor. A globális jelek, vagyis az órajelek és a reset-jelek időzítési viszonyainak fontosságát láthattuk a 6. fejezetben. E jelek terjesztésével kapcsolatban két fontos tényezőt kell figyelembe vennünk:

- A globális jeleket – ahogy elnevezésük is utal rá – az áramkör minden flip-flopjához el kell juttatni, ami azt jelenti, hogy az ezeket meghajtó celláknak a terhelése igen nagy. Ahhoz, hogy minél előbb eljusson a jel a forrástól az egyes flip-flopokig, nagy meghajtóképességű ismétlő/jelregeneráló fokozatokra van szükség.
- A 6. fejezetben azt is láthattuk, hogy mind az órajel, mind a reset esetén fontos az, hogy minden flip-flop ugyanakkor „lássa" az adott jelen bekövetkező változásokat, tehát az elcsúszás *(skew)* mértéke minimális legyen.

A fenti követelmények miatt a globális jelek chip-en belüli terjesztésére speciális vezetékhálózatokat, ún. órajelfákat alakítunk ki. Az egyik legelterjedtebben alkalmazott órajelfa-struktúrát, a H-fát mutatja be a 7-4. ábra.

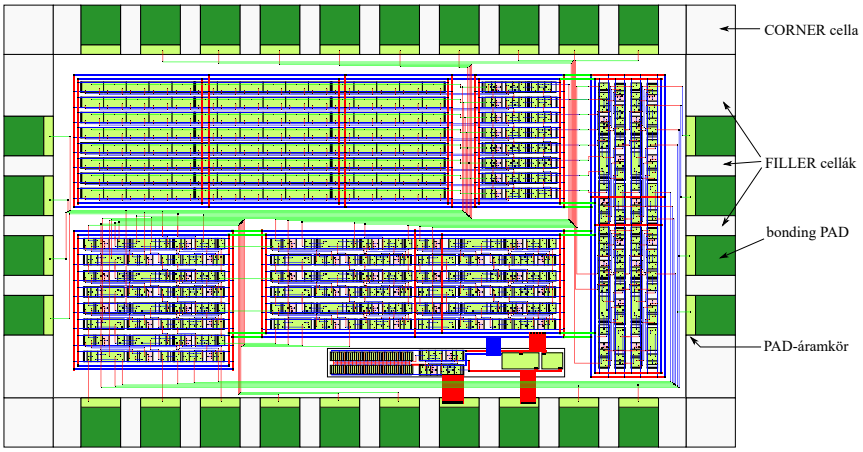
Az órajelfák jellegzetessége, hogy ideális esetben a globális jel forrása és végpontjai között az egyes vezetékszakaszok hossza teljesen azonos, így elvileg az elcsúszás mértéke is nulla. A valóságban – a chip egyéb erőforrásainak elhelyezésére vonatkozó számtalan más követelmény miatt – ez az ideális helyzet aligha teljesülhet, ezért az órajelfák nem teljesen szimmetrikusak. Az aszimmetriák kompenzálása céljából az órajelfák jelerősítő/ismétlő meghajtóáramkörei ún. *de-skew* áramköröket tartalmaznak, az egyes vezetékszakaszok impedanciáját pedig az adott chip igényei szerint egyedileg tervezzük meg. Az ilyen, kis



7-4. ábra. *A H-típusú órajelfák struktúrája*

mértékű elcsúszást és minimális késleltetést biztosító vezetékhálózatokat kiegyensúlyozott *(balanced)*, impedanciaillesztett órajelfáknak[62] nevezzük.

Egy teljes chip fizikai terve a 7-3. ábrán bemutatotthoz hasonló, különböző funkciójú, eltérő összetettségű és méretű makrocellákból áll össze (lásd 7-5. ábra).



7-5. ábra. *Egy több makrocellát tartalmazó chip fizikai terve*

Az ábrán a standard cellás integrált áramköröknek néhány újabb szerkezeti jellegzetessége is megfigyelhető. A makrocellák orientációja nem feltétlenül a 7-3. ábrán láthatónak megfelelő, a makrocellák közötti huzalozás és a teljes chip méretének minimalizálása céljából bizonyos esetekben kedvezőbb, ha egy-egy makrocellát 90 fokkal elforgatva helyezünk le. A makrocellák táp- és földgyűrűit lehetőleg több helyen is összekapcsoljuk egymással, ezzel is csökkentve a táp- és földhálózat eredő impedanciáját. Az elhelyezett makrocellák között ugyancsak szükség van huzalozási csatornákra, amelyekben a makrocellák közötti összeköttetéseket

[62] Az elnevezés kissé félrevezető olyan szempontból, hogy nemcsak az órajel, hanem a reset terjesztésére is ilyen hálózatokat használunk.

megvalósító vezetékeket helyezzük el. A huzalozási csatornák alakját leginkább a makrocellák egymáshoz viszonyított helyzete és orientációja határozza meg, az tehát nem olyan szabályos, mint a makrocellákon belül. A cellaterületet egy ún. PAD-gyűrű veszi körül, amely a chip kivezetéseihez való kapcsolódást teszi lehetővé. A PAD-gyűrűt alkotó cellák egyrészt tartalmazzák az ESD-védelmet és jelszint-illesztést ellátó áramköröket, másrészt magukban foglalják a chip tokjának kivezetéseivel való összeköttetést biztosító mikrohuzalkötéshez szükséges fémezett felületet *(bondig pad)*. A PAD cellák közötti, üresen maradt helyeket a már említett FILLER cellákhoz hasonló, de a PAD cellákkal azonos magasságú cellákkal töltjük fel. A chip sarkaiba helyezett FILLER cellák az ún. CORNER cellák.

7.2 FPGA áramkörök

A különböző programozható logikai eszközök – a standard cellás áramköröknél látottakkal ellentétben – nem „előre tervezéssel", hanem „előre gyártással" gyorsítják egy digitális funkciót megvalósító termék kifejlesztésének folyamatát. E technológiák alapötlete az, hogy a digitális áramkörök alapelemeit nagy számban előre legyártjuk, és a konkrét funkciót az alapelemek közötti, utólag konfigurálható összeköttetéshálózat megfelelő beállításával alakítjuk ki. A különböző eszközcsaládok legfőképp az előregyártott alapelem összetettségében és a konfigurálható összeköttetések megvalósításának technológiájában térnek el egymástól. A logikai alapelem összetettsége alapján az alábbi kategóriák különböztethetők meg:

- **Very fine-grained**: Az előregyártott alapelem maga a tranzisztor. Ennek megfelelően ezt a technológiát *„sea-of-transistors"* megoldásnak is szokás nevezni.
- **Fine-grained**: A mai programozható logikai eszközök túlnyomó többsége ebbe a kategóriába sorolható. Megvalósítását tekintve sokféle lehet. Ami a különböző *„fine-grained"* megoldásokat összeköti, hogy az alapelem ebben az esetben egyetlen bit információ feldolgozását/tárolását teszi lehetővé. Pl. egyetlen logikai kapu *(„sea-of-gates")*, egy-kimenetű, konfigurálható logikai függvény + egy bitnek megfelelő tárolófunkció (CPLD, FPGA).
- **Coarse-grained**: Az előregyártott alapelem több bit információ feldolgozásáért/tárolásáért felelős. Pl. egy ALU és egy hozzá tartozó regisztertömb (pl. *PipeRench Reconfigurable Computing Project*).

Ma a legszélesebb körben alkalmazott programozható logikai eszközök az FPGA-k *(Field-Programmable Gate Array*[63]*)*, ami annak köszönhető, hogy az általuk megvalósítható

[63] A *„field-programmable"* kifejezés ezeknek az áramköröknek a legfőbb jellegzetességére utal, vagyis hogy a felhasználás helyén – a felhasználó által – programozhatók, ehhez a gyártó közreműködésére már nincs szükség. Ez a terminológia a korábbi, szintén az előre gyártás elvét megvalósító technikákkal összevetve válik érthetővé. Az ún. maszk-programozott eszközök esetén például az előre gyártás a litográfiai maszkoknak csak egy részére vonatkozott, amellyel létre tudtak hozni egy olyan struktúrát, amelynek konkrét funkciója még nem volt, azt a legfelső fémezési réteg, vagyis a legutolsó maszk határozta meg. A felhasználói igények teljesítéséhez tehát elegendő volt egyetlen maszkot készíteni, az összes többit előre elő lehetett állítani, ami gyorsabb fejlesztést és olcsóbb terméket jelentett, de a funkcionálisan teljes chip előállításához továbbra is a gyártó közreműködésére volt szükség.

funkció összetettsége, a rendelkezésre álló erőforrások számítási teljesítménye, mérete, ára és fogyasztása hihetetlenül széles skálán változik, ugyanakkor rugalmasság tekintetében is eltérő megoldások közül választhatunk. Mindez arra vezet, hogy rengeteg különféle problémára találhatunk az adott helyzetben éppen megfelelő FPGA-t. E sokszínűség ellenére megfogalmazható a képességeknek egy szűk halmaza, amely minden FPGA-ra jellemző a legkisebb, kisfogyasztású, leginkább alrendszerek közötti interfészfunkciókat *(„glue-logic")* betöltő eszközöktől a legnagyobb számítási teljesítményű, többmagos mikroprocesszor-magokat és többszáz Mbit/s sávszélességű kommunikációs interfészeket kezelő nagyágyúkig.

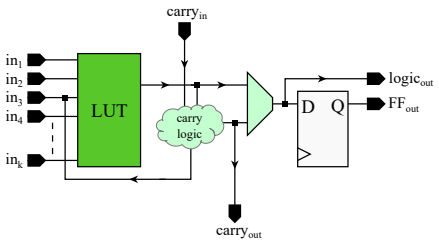
A legfontosabb közös tulajdonság természetesen a konfigurálhatóság. Minden hardver-erőforráshoz egy sor konfigurációs bit tartozik, amelyek az adott alkalmazás igényeire hangolják az áramkörök működését. E vezérlő bitek összességét konfigurációs memóriának nevezzük. A konfigurációs memóriát rendszerint JTAG interfészen keresztül tölthetjük fel tartalommal. Technológiai megvalósítását tekintve a legelterjedtebb megoldás az SRAM-alapú konfigurációs memória, amelyben minden egyes konfigurációs bitet egy-egy hat-tranzisztoros SRAM cella tárol, de ma már gyakran használnak FLASH-cellákat is erre a célra, aminek az előnye, hogy az eszköz a tápfeszültség kikapcsolása után sem „felejti el" korábbi konfigurációját, továbbá fizikailag sokkal kisebb, mint egy SRAM cella és fogyasztása is kedvezőbb.

A FLASH-alapú megoldás nemfelejtő jellegét valósítják meg azok az eszközök, amelyekben maga a konfigurációs memória SRAM, viszont a chip tartalmaz akár több FLASH-memóriát is különböző konfigurációk tárolására. A tápfeszültség bekapcsolásakor egy automatikus inicializálási folyamat során a kiválasztott konfiguráció betöltődik az SRAM-ba, az eszköz tehát tulajdonképpen „felprogramozza saját magát". E megoldás előnye, hogy akár több konfigurációt is eltárolhatunk egyazon eszközben, hátránya, hogy a kiválasztott konfiguráció FLASH memóriából SRAM-ba való áttöltése sok időt vesz igénybe, a tápfeszültség bekapcsolása után tehát csak néhány ms-mal lesz kész az eszköz a működésre.[64]

Technológiailag igen eltérő megoldásnak számítanak az ún. anti-fuse-alapú FPGA-k, amelyeknek tulajdonképpen nincsen konfigurációs memóriája, a hardver-erőforrások „véglegesítését" ún. anti-fuse kapcsolóelemekkel érik el. E megoldás előnye, hogy lényegesen kisebb méretben kivitelezhető, mint akármelyik másik konfigurációs megoldás, továbbá struktúrájából adódóan immunis a nagyenergiájú részecskék becsapódása által előidézett, ún. *soft-error* jelenségre, amely bármely másik konfigurációs memóriatípusnál igen nagy gondot jelent. Ezért az anti-fuse FPGA-kat tipikusan űripari és egyéb, sugárzásnak kitett alkalmazásokban használják. Az anti-fuse megoldás hátránya, hogy ezeket az eszközöket a gyártást követően csupán egyszer lehet felprogramozni. Az ilyen eszközöket OTP *(One Time Programmable)* FPGA-knak nevezzük.

[64] A két különböző, FLASH memóriát használó konfigurációs megoldás közötti különbség kihangsúlyozása céljából azokat az eszközöket, amelyek konfigurációs bitjeit a chip felületén elosztott, a konfigurálandó hardverelem közelében elhelyezett FLASH cellák tárolják, *„real-FLASH"* eszközöknek is nevezzük, utalva arra, hogy a másik megoldás lényegében egy FLASH-blokkokkal kiegészített SRAM FPGA annak minden méret- és fogyasztásbeli hátrányával együtt.

Konfigurálható logikai elemek • Az FPGA-k legelemibb összetevői a konfigurálható logikai elemek (*Basic Logic Element*, BLE), amelyek a digitális áramkörök kombinációs és szekvenciális alapfunkcióinak megvalósításához szükséges erőforrásokat tartalmazzák. Gyártótól és eszközcsaládtól függően a logikai elemek felépítése igen változatos lehet, bizonyos tulajdonságaik azonban közösek; egyetlen bit információ feldolgozására képesek és funkcionálisan jól modellezhetőek a 7-6. ábrán látható, egyszerű sémával.



7-6. ábra. *Egy FPGA konfigurálható logikai alapelemének egyszerűsített modellje*

A kombinációs funkciókat a BLE-k ún. LUT-okkal *(Look-Up Table)* valósítják meg, amelyek konfigurációjuktól függően bármilyen logikai összefüggést meg tudnak valósítani a bemeneteik között. A bemenetek száma eszközről eszközre változik, a legelterjedtebbek a 4 és 6 bemenettel rendelkező megoldások, de léteznek olyan architektúrák is (pl. Microsemi ProASIC család), amelyek 3 bemenetű LUT-okkal vannak felszerelve. A BLE-ket felhasználva a különböző gyártók összetettebb struktúrákat hoznak létre, amelyek elnevezése és összetettsége is nagyon különböző lehet:[65]

- **Xilinx**: 2 × BLE = slice, 2/4 slice = CLB *(Configurable Logic Block)*
- **Intel** (Altera): 16 × BLE = LAB *(Logic Array Block)*
- **Microsemi** (Actel): 2 × BLE = SuperCluster, n × 100 SuperCluster = CoreTile

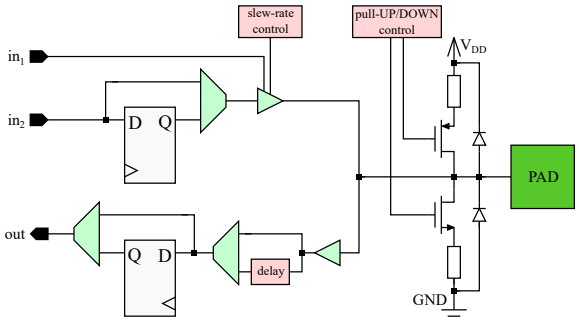
A digitális aritmetikai áramkörökben az elemi teljes összeadók *carry* kimenetének előállítása az áramkörök késleltetése, azon keresztül pedig a számítási teljesítmény szempontjából kritikus. Mivel az elemi teljes összeadó szinte minden aritmetikai áramkörnek a magját képezi, ezért a legtöbb FPGA már a legalapvetőbb építőelemek, a BLE-k szintjén tartalmaz olyan erőforrásokat, amelyek a *carry* logika hatékony kialakítását segítik. Ennek részleteit a 8-41. ábrán mutatjuk be.

A BLE-k a kombinációs funkciót megvalósító erőforrásokon kívül egy-egy tároló elemet is tartalmaznak. Ez a tároló elem ugyan csak egyetlen bit tárolására alkalmas, de igen rugalmasan konfigurálható. Használhatjuk szint- vagy élvezérelt tárolóként (latch vagy flip-flop), meghatározhatjuk, hogy az órajelnek a felfutó, a lefutó, vagy akár mindkét élére

[65] Ez sokszor megnehezíti az egyazon funkciót megvalósító, különböző FPGA-kra szintetizált áramkörök valós erőforrás-felhasználásának összehasonlítását. A gyártó saját tervezőrendszerei értelemszerűen a saját erőforrásaik számával kifejezve adják meg egy terv erőforrásigényét, ami alapján olykor igen nehéz meghatározni, hogy ugyanazt a tervet egy másik gyártó FPGA-jára szintetizálva mennyi erőforrást használnánk fel.

érzékeny legyen,[66] szükség esetén használhatunk órajel-engedélyező bemenetet és szinkron vagy aszinkron reset és preset bemeneteket.

Konfigurálható I/O elemek ⬩ Az FPGA-k általános célú eszközök, vagyis alkalmasnak kell lenniük különféle felhasználási környezetekben, sokféle, eltérő áramköri környezetben működni. Ennek érdekében az FPGA-knak a külvilággal való kapcsolat megvalósításáért felelős részáramköreik, az ún. I/O-blokkok *(IOB)* az általános logikai alapelemekhez hasonlóan rugalmasan konfigurálhatók (lásd 7-7. ábra).

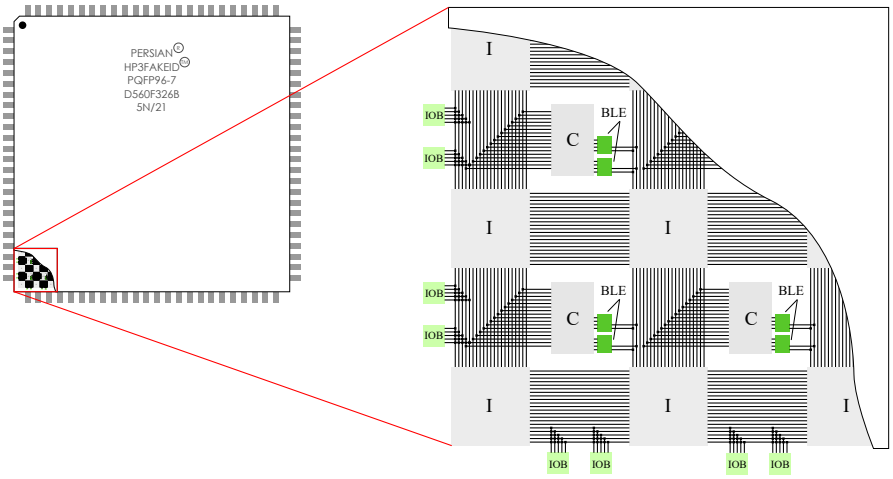


7-7. ábra. *Egy FPGA I/O blokkjának egyszerűsített modellje*

A standard cellás integrált áramkörök PAD-áramköreihez hasonlóan az FPGA-k I/O blokkjai is megvalósítanak ESD-védelmet és jelszintillesztést. Rugalmasságuk abban rejlik, hogy nemcsak egy, hanem egy egész sor jelszint-szabványt képesek támogatni, továbbá tartalmaznak ún. *slew-rate* (jelváltozási sebesség) vezérlést, amely segítségével konfigurálhatóvá válik az eszköz kimeneteinek meghajtóképessége is. A konfigurálható, passzív felhúzó/lehúzó elemek ugyancsak az áramköri környezet igényeihez való alkalmazkodást segítik. A 6.2.1. pontban tárgyalt bemeneti és kimeneti késleltetési viszonyok kialakításakor nagy segítség lehet, ha az áramkörünk kimeneteit meghajtó flip-flopok, valamint a bemeneteket mintavételező flip-flopok az eszköz kivezetéseihez fizikailag közel helyezkednek el. Ennek érdekében az I/O blokkok egy-egy flip-flopot is tartalmaznak, amelyek használata szintén opcionális. Ugyancsak az időzítések pontos beállítását segítik a konfigurálható késleltetőelemek.

Konfigurálható összeköttetéshálózat ⬩ A logikai alapelemek és I/O erőforrások konfigurálhatósága elengedhetetlen ahhoz, hogy ugyanazt az eszköztípust sokféle célra, másmás alkalmazási környezetekben használhassunk, a rugalmasság legmeghatározóbb forrása azonban az, hogy az alapvető logikai erőforrások közötti összeköttetéshálózat maga is konfigurálható. A logikai elemek az FPGA chip felületén mátrix-szerűen vannak elrendezve, és kis túlzással azt mondhatjuk, hogy bármely kettő összekapcsolható egymással a konfigurálható huzalozási hálózat segítségével. A 7-8. ábra a legelterjedtebb huzalozási megoldást, az ún. *„island”* típusú architektúrát mutatja be.

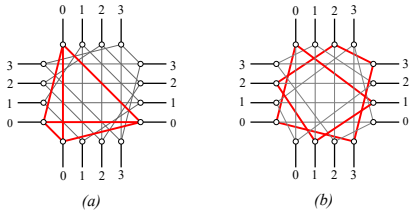
[66] A mindkét élre érzékeny, ún. *dual-edge* flip-flopoknak a DDR *(Double Data Rate)* interfészek megvalósításában van fontos szerepe.



7-8. ábra. *„Island"-stílusú összeköttetéshálózat*

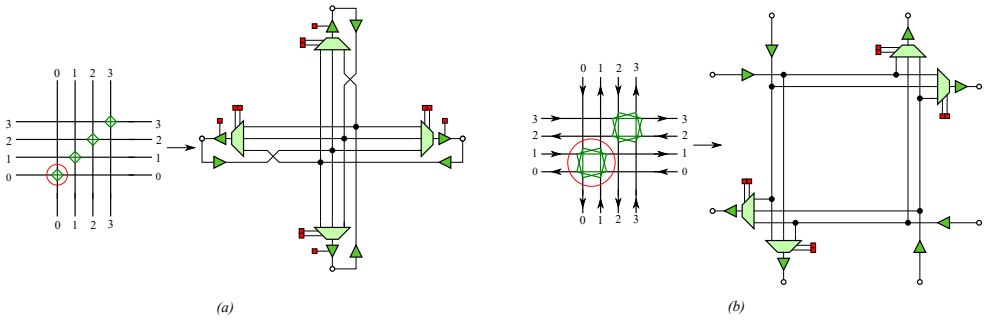
A logikai elemek között vízszintes és függőleges huzalozási csatornák vannak kialakítva, amelyek találkozási pontjaiban kapcsolómátrixok (az ábrán I-vel jelölve: *Interconnection Block*) foglalnak helyet. E kapcsolómátrixok feladata, hogy a huzalozási csatornákban kialakított vezetékszakaszok között kapcsolatot teremtsenek. A logikai elemeket a 7-8. ábrán C-vel *(Connection Block)* jelölt, a kapcsolómátrixokhoz hasonló felépítésű, de azoknál némileg egyszerűbb kapcsolóelemek kötik a huzalozási csatornákhoz.

A kapcsolómátrixok az összes lehetséges kapcsolatnak csak egy részét képesek megvalósítani, mivel a tetszőleges vezetékpár összekapcsolására alkalmas szerkezet erőforrás- és területigénye megengedhetetlenül nagy lenne. A megvalósított konfigurálható összeköttetések száma erősen befolyásolja az adott FPGA logikai erőforrásainak kihasználhatóságát. Minél több összeköttetést tesz lehetővé egy kapcsolómátrix, annál könnyebb a felhasználói terv megvalósítása, ugyanakkor annál nagyobb lesz magának a kapcsolómátrixnak a mérete, tehát kevesebb szilíciumfelület marad a logikai funkciót ellátó áramköröknek. A 7-9. ábra a Disjoint és a Wilton típusú kapcsolómátrixok felépítését mutatja be.



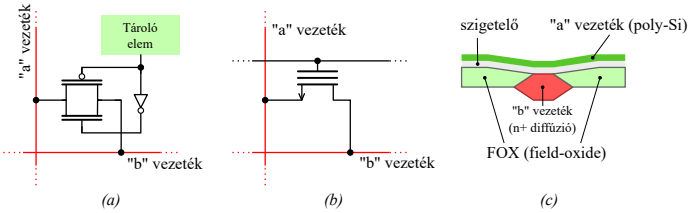
7-9. ábra. *A Disjoint (a) és a Wilton (b) típusú kapcsolómátrixok által megvalósított összeköttetések*

A kapcsolómátrixokon belüli összeköttetéseket irányuk alapján is megkülönböztethetjük. A 7-10. ábra egy kétirányú *(a)* és egy csak egyirányú *(b)* összeköttetések megvalósítására alkalmas kapcsolómátrix-struktúrát mutat be.



7-10. ábra. *Egy- és kétirányú összeköttetéseket megvalósító kapcsolómátrix-struktúrák*

A kapcsolómátrixokon belüli konfigurálható összeköttetések megvalósítására sok megoldás létezik, a ma legelterjedtebben használt megoldásokat a 7-11. ábra szemlélteti.



7-11. ábra. *Konfigurálható összeköttetések megvalósítása*

Tároló elem + transmission-gate struktúra (7-11. ábra: *a)* • A két vezetékszakasz közé egy transmission-gate cella kerül, amelynek vezérlését egy egybites memóriaelem, az eszköz konfigurációs memóriájának egy bitje végzi.

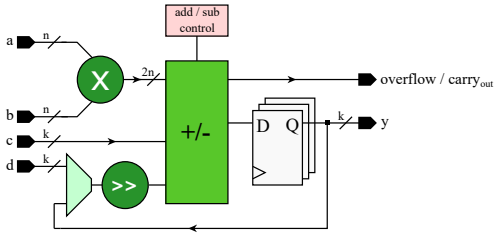
FLASH cella mint kapcsolóelem (7-11. ábra: *b)* • A két vezetékszakasz között az összeköttetést egy lebegő gate-es MOS tranzisztor valósítja meg. Ha a konfigurálási folyamat során töltést viszünk a lebegő gate-elektródára, úgy a kapcsolótranzisztor nyit és alacsony impedanciás összeköttetést valósít meg a két vezetékszakasz között. A megoldás előnye, hogy a lebegő-gate töltése a tápfeszültség kikapcsolása után is megmarad, tehát az eszközt nem kell újrakonfigurálni kikapcsolás után.

Anti-fuse (7-11. ábra: *c)* • Az összeköttetés két vezeték között úgy alakul ki, hogy két vezető réteg közé (az ábrán látható példában poly-Si és n+ diffúzió) vékony szigetelő réteget helyezünk el, amely a konfigurálási folyamat során a két vezető réteg közé kapcsolt megfelelő potenciálkülönbség hatására átüt, egy maradandó vezető csatornát képezve (innen az elnevezés, „anti-biztosíték”). Az anti-fuse eszközök fő előnye, hogy – a FLASH-alapú megoldáshoz hasonlóan – tápfeszültég nélkül is megőrzik a konfigurációt, bár – tekintve hogy az anti-fuse elem „felprogramozása” destruktív folyamat –, ezeket az eszközöket csak egyszer lehet felkonfigurálni. Megjegyzendő továbbá, hogy ez a fajta összeköttetés a legjobb minőségű a felsoroltak közül abban az értelemben, hogy a két vezeték közötti összeköttetésnek

az ellenállása itt a legkisebb. Az összeköttetés ellenállása maguknak a vezetékszakaszoknak a parazita kapacitásaival RC-tagot alkotnak, amelyek időállandója befolyásolja a maximális jelváltozási sebességet az adott vezetéken. A kis ellenállású összeköttetés tehát egyúttal gyors működést is jelent.[67] További sajátossága az anti-fuse eszközöknek, hogy az összeköttetés felépítéséből adódóan „sugárzásálló", ellentétben a másik két, itt felsorolt megoldással. Ezért űripari alkalmazásokban előszeretettel alkalmaznak anti-fuse eszközöket.

Az eddig felsorolt hardver-erőforrásokat általános célú FPGA erőforrásoknak nevezzük, ezekből ui. tetszőleges digitális funkció kialakítható. Egyes, gyakran alkalmazott konstrukciók azonban ezekből az erőforrásokból építkezve nem valósíthatók meg hatékonyan. A probléma kiküszöbölésére a mai modern FPGA-k ún. heterogén architekturális elemeket is tartalmaznak, amelyek ugyan kizárólag valamilyen speciális funkció elvégzésére alkalmasak, de azt a funkciót nagyon hatékonyan, kis méret, kis fogyasztás és nagy sebesség mellett valósítják meg.

A legelső ilyen erőforrások – az FPGA áramkörök elsődleges alkalmazási területe, a digitális jelfeldolgozás igényeinek megfelelően – a blokk-RAM és DSP modulok voltak. A BLE-kből létrehozható ugyan tetszőleges méretű memória, hiszen minden BLE tartalmaz egy tároló elemet, de éppen amiatt, hogy egy BLE mindössze egyetlen bit tárolására képes, egy nagy méretű memória összeállítása megengedhetetlenül sok BLE-t igényelne, ráadásul a tárkapacitás növekedésével az egyes bitek közötti összeköttetések megvalósítása is egyre nehezebbé válik. E probléma megoldására az FPGA-kba dedikált memória-áramköröket, ún. blokk-RAM modulokat építenek be, amelyek nagysebességű és – a BLE-kből felépíthetőhöz képest – nagy kapacitású (Mbit nagyságrend) memóriák. Bár funkciójuk alapvetően kötött, a rugalmas felhasználhatóság érdekében számtalan tulajdonságuk konfigurálható. Általában single- vagy dual-port működésre képesek, kimeneteik szinkron működésűek, az általuk tárolt vektorok mérete és száma gyakorlatilag tetszőlegesen megválasztható (természetesen vannak eszközről eszközre változó felső korlátok). Ugyancsak nagyszámú BLE-re lenne szükség a DSP alkalmazásokban igen gyakran előforduló szorzás műveletet megvalósító áramkörök megvalósításához, ezért az FPGA-k viszonylag nagy számban tartalmaznak ún. DSP slice-okat, vagyis olyan BLE-ket, amelyek blokk szorzó, szorzó-akkumulátor (MAC), barrel-shifter és ezekhez hasonló áramköri részleteket tartalmaznak (lásd 7-12. ábra).

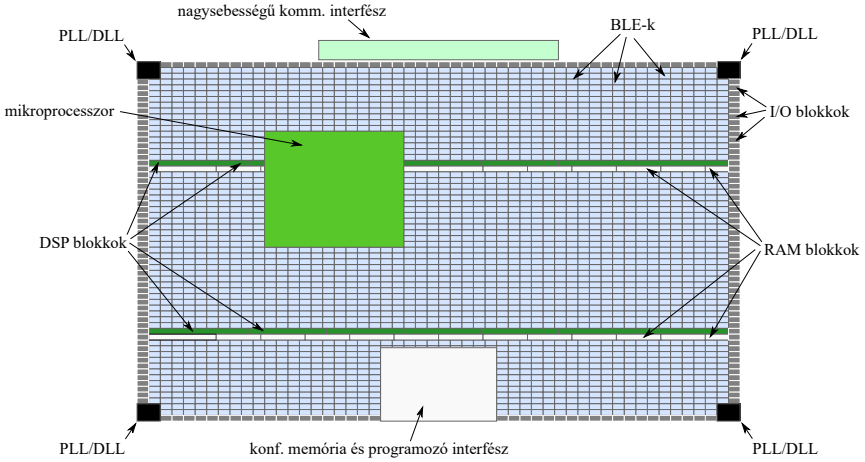


7-12. ábra. *Egy DSP-slice egyszerűsített kapcsolási sémája*

[67] Ez természetesen nem jelenti azt, hogy bármely anti-fuse eszköz gyorsabb, mint az SRAM, vagy FLASH-alapú verziók. Az eszköz működésének sebességét számtalan egyéb tényező is befolyásolja.

A nagy kapacitású FPGA-k egy vagy több, hard-makróként beépített mikroprocesszor magot is tartalmazhatnak. Ezek a mikroprocesszorok rendszerint nagy számítási teljesítményű RISC magok (ARM Cortex, IBM PowerPC stb.). Az ilyen eszközöket SoC *(System-on-Chip)* FPGA-nak nevezzük.

Az FPGA erőforrásokból megvalósított alkalmazás-specifikus adatfeldolgozó rendszerek és hard-makróként beépített mikroprocesszorok ún. SoPC *(System on a Programmable Chip)* rendszert alkotnak, amelyeknek a feladattól függően adott esetben a külvilággal is nagy sávszélességű kapcsolatot kell megvalósítaniuk. E célból az FPGA-k nagysebességű soros kommunikációs interfészáramköröket (*high-speed transceiver*, pl. PCIe, RocketIO, Gb Ethernet stb.) is tartalmaznak, amelyek nemcsak összetettségük miatt nem lennének megvalósíthatók BLE-kből, hanem a nagy átviteli sebességből adódó magas órajel-frekvenciák miatt bizonyos részleteik analóg megoldásokat is igényelnek. A 7-13. ábra egy jellegzetesnek nevezhető FPGA chip floor-planjét, vagyis felülnézeti vázlatát mutatja be, kiemelve az általános célú és heterogén architekturális elemek elhelyezkedését. Az ábra az áttekinthetőség kedvéért a huzalozási csatornákat, kapcsolómátrixokat nem tartalmazza.



7-13. ábra *– Egy FPGA-chip vázlata*

Speciális célú huzalozási erőforrásnak tekinthetők a standard cellás áramköröknél, a 7.1. pontban említett órajelfák is, amelyek szerkezete az FPGA-k esetén is hasonló az ott látottakhoz. Érdekesség, hogy az FPGA-k esetén ezeket a speciális huzalozási erőforrásokat nem annyira órajelfáknak, mint inkább globális huzalozási hálózatoknak *(global routing network)* szokás nevezni (az elnevezések természetesen gyártófüggőek). Ennek az a magyarázata, hogy bár elsősorban az órajel és a reset terjesztésére használjuk őket, gyakran előfordul, hogy ilyen hálózatokon keresztül vezetjük az általános logikai funkciót betöltő, de nagy terhelésű jeleket is. Ezeknek a vezetékhálózatoknak a száma eszközről eszközre változik. A több, különálló órajeltartományt tartalmazó áramkörök megvalósítását ezek a globális hálózatok úgy támogatják, hogy a konfigurációs bitek segítségével részekre oszthatók, az egyes részhálózatokat pedig egymástól független forrásokról hajthatjuk meg.

Az FPGA eszközök többségében az órajel terjesztésén túl annak előállítására is speciális, konfigurálható hardverelemek állnak rendelkezésre, amelyeket összefoglaló néven órajel-menedzsment moduloknak nevezünk. Megvalósításukat tekintve lehetnek analóg PLL-ek *(Phased-Locked Loop)* vagy ezek digitális változatai, DLL-ek *(Delay-Locked Loop)*. Feladatkörük a következőkre terjed ki:

- **Jitter-kompenzálás**: Az elsődleges órajelforrás remegését *(jitter)*, vagyis a periódusidő ciklusonkénti apró változásait csökkentik.
- **Frekvenciaszintézis**: Az elsődleges órajelforrásból más frekvenciájú, másodlagos órajelforrásokat állítanak elő.
- **Fázistolás**: Segítségükkel a másodlagos órajelforrásoknak az elsődlegeshez és az egymáshoz viszonyított fázisa is konfigurálható.
- **Elcsúszás-kompenzálás**: A másodlagos órajelforrásoknak az elsődlegeshez és az egymáshoz viszonyított elcsúszását *(skew)* kompenzálják.



8

Áramkörszintézis

Elérkeztünk az összetett digitális makrocellák tervezési folyamatának első nagy mérföldkövéhez. Az almodulok RTL modelljeinek helyességét verifikáltuk, elvégeztük a toplevel-integrációt, az almodulok együttműködését és a rendszerszintű funkcionális koncepciót is igazoltuk a toplevel verifikáció során, továbbá rögzítettük a leendő áramkörünkkel szemben támasztott időzítési követelményeket is az előírt órajel-frekvencia és a leendő áramköri környezet tulajdonságainak ismeretében. A következő lépés az áramkörszintézis, amely során az absztrakt RTL modellünkből fizikailag megvalósítható áramköri szerkezetet, logikai kapukból és tároló elemekből álló hálózatot készítünk, amely külső interfészein bármilyen gerjesztés hatására pontosan ugyanazt a kimenetet állítja elő, mint az RTL modell.

Mielőtt részletesen bemutatnánk az automatizált áramkörszintézis folyamatát, intuitív módon elvégezzük néhány HDL modell kézi szintézisét, vagyis a HDL modellt informális módszerrel alakítjuk át olyan áramköri struktúrává, amelynek elemei alapcellaként rendelkezésünkre állnak. E példák célja, hogy érzékeltessék az automatizált áramkörszintézis algoritmusok feladatának lényegét, és látványosan szemléltessék a HDL-alapú tervezésre és az automatizált áramkörszintézisre épülő tervezési folyamat hatékonyságát.

8.1 Manuális áramkörszintézis – példák

Ahogy azt a 4. fejezetben láthattuk, egy áramköri modul HDL modellje konkurens értékadásokból, más modulok példányosításaiból, valamint kombinációs és szinkron folyamatokból áll. A konkurens értékadások és kombinációs folyamatok szemantikáját megvalósító áramköri részletek szintézise különösebb nehézséget nem okoz, hiszen ezek a nyelvi szerkezetek mindig valamilyen kombinációs hálózatot írnak le, a példányosítás utasítások pedig már HDL kód formájukban is valójában szerkezetet definiálnak, ezek esetén tehát tulajdonképpen „nincs is mit szintetizálni". A szinkron folyamatok szintézise már egy kicsit keményebb dió. A szinkron folyamatokba foglalt értékadások – ahogy azt korábban több példában is láthattuk – a balértékeik által egyértelműen meghatározzák egyrészt azt, hogy milyen tároló elemek (flip-flopok és memóriák) lesznek az áramkörben, másrészt, közvetetten a jobbérték-kifejezéseken keresztül azt is leírják, hogy ezeknek a tároló elemeknek a bemeneteire milyen kombinációs logikát kell kapcsolnunk (lásd 4-35. ábra). Ennek értelmében a HDL leírások alapján történő áramkörszintézist annak megállapításával érdemes kezdeni, hogy milyen tároló elemekre lesz szükség. Ha meghatároztuk a tároló elemeket, akkor az azokra vonatkozó értékadások vizsgálatával megszerkeszthetjük a bemeneteiket

meghajtó kombinációs hálózatokat. Egy-egy értékadás vizsgálata persze önmagában nagyon egyszerűnek tűnik – ahogy azt láthattuk például a 4-18. ábrán látható esetben –, azonban összetettebb modellek esetén ezek az értékadások bonyolult feltételrendszerekbe, vezérlő (pl. `if-else`, `case`, `for` stb.) utasításokba lehetnek beágyazva, amelyek ugyancsak befolyásolni fogják azt, hogy az adott tároló elem bemenetét milyen kombinációs logika hajtja majd meg. Tovább bonyolítja a helyzetet, ha a VHDL modellben változókat (`variable`) vagy Verilog esetén az ezzel egyenértékű blokkoló értékadásokat használunk, az ezek viselkedését megvalósító áramköri részletek megszerkesztése ugyanis – legalábbis ha nem használunk gépi módszereket a szintézisre – igen körülményes lehet.

Az itt bemutatott példák kézi szintézise során valójában azt a folyamatot végezzük el manuálisan, amelyet az automatizált áramkörszintézis szoftverek kódelemzésnek *(elaboration, translation)* hívnak, és amelynek eredménye a 4. fejezet példáiban is látható, alapvető áramé köri felépítés. Ezek a szerkezeti modellek olyan, önmagukban is összetett, regiszter-transzfer szintű alapelemekből állnak, amelyeknek optimalizálatlan kapuszintű reprezentációja a technológiai cellakönyvtárakban rendelkezésre álló alapcellákból könnyen előállítható, és amelyeken a logikai szintézist végző szoftverek még egy sor optimalizációt elvégeznek, mielőtt a fizikai szintézisre sor kerülne.

Hangsúlyozzuk, hogy a mai mérnöki gyakorlatban a HDL modellek szintézisét az esetek túlnyomó többségében a fent említett szoftverek segítségével végezzük, az itt bemutatott példák célja csupán annak szemléltetése, hogy egy ilyen szintézis szoftver „mit lát abban a kódban", amit feldolgozunk vele. Minél jobban ismerjük e szoftverek működését, annál megbízhatóbban szintetizálható, annál jobb minőségű HDL modelleket tudunk készíteni.

Az első példánk egy egyszerű, négybites számláló áramkör lesz a következő képességekkel:

- A számláló engedélyezhető/tiltható az `ena` bemenet segítségével. Ha a számláló letiltott állapotban van, akkor a kimenete a bemenetektől független és konstans, a tiltás előtti értéket tartja.
- Ha a számláló engedélyezett, tehát az `ena` bemenet logikai magas értékű, akkor a `load` és `parallel_in` bemenetek segítségével tetszőleges értéket betölthetünk a számlálóba.
- Ha a számláló engedélyezett, és a `load` bemenet inaktív, akkor a `dir` bemenet értékétől függően a kimenet minden órajel-felfutóélnél eggyel nő vagy csökken.

A számláló HDL modellje a 8-1. ábrán látható.

Első lépésként állapítsuk meg, hogy milyen tároló elemekre lesz szükségünk. A számlálós példa ebből a szempontból igen egyszerű, a HDL modell egyetlen folyamatból áll, amely csupán egyetlen négybites vektort hajt meg (`cout`), ami ráadásul egyúttal a kimenet értéke is. A `cout` vektornak a vezérlő bemenetek értékétől függően az órajel felfutó éleinél más-más értéket adunk. A vezérlő bemenetek között – ahogy azt a számláló szöveges specifikációjában is egyértelműen rögzítettük – prioritási sorrendet állítunk fel; a legmagasabb prioritású vezérlő bemenet az engedélyező jel (`ena`), ha ennek értéke alacsony, akkor semmilyen egyéb funkció sem fog működni, a számláló tartja az értékét. A második a prioritási

VHDL

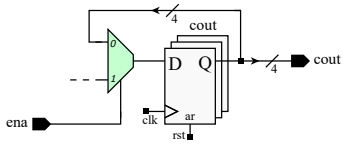
```
14 architecture rtl of counter is
15 begin
16
17   process ( clk, rst )
18   begin
19     if ( rst = '1' ) then cout <= X"0";
20     elsif ( rising_edge(clk) ) then
21       if ( ena = '1' ) then
22         if ( load = '1' ) then
23            cout <= parallel_in;
24         elsif ( dir = '0' ) then
25             cout <= cout + 1;
26         else cout <= cout - 1;
27         end if;
28       end if;
29     end if;
30   end process;
31
32 end architecture rtl;
```

Verilog

```
1   module counter (...);
...
8     always @ ( posedge clk, posedge rst )
9     begin
10      if ( rst == 1'b1 ) cout <= 4'd0;
11      else
12        if ( ena == 1'b1 )
13          if ( load == 1'b1 ) cout <= parallel_in;
14          else
15            if ( dir == 1'b0 )
16                cout <= cout + 4'd1;
17            else cout <= cout - 4'd1;
18    end
19
20 endmodule
```

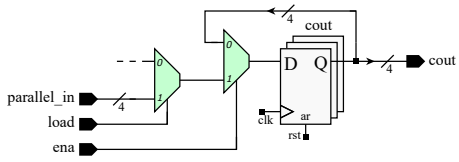
8-1. ábra. *A számláló áramkör HDL modellje*

sorban a `load` bemenet, amivel egy, a felhasználó által meghatározott értéket (`parallel_in`) tölthetünk a számlálóba. A legalacsonyabb prioritású vezérlő bemenet a számlálás irányát meghatározó `dir` bemenet. Amikor egy tároló elemet meghajtó kombinációs hálózatot szerkesztünk, akkor gyakran találkozhatunk ehhez hasonló prioritási sorokkal, amelyeket áramkörileg sorba kapcsolt multiplexerekkel tudunk megvalósítani. A HDL modell minden feltételvizsgálatához egy-egy multiplexer tartozik, amelynek kiválasztó bemenetét egy, a feltételvizsgálatot megvalósító kombinációs hálózat hajtja meg. A multiplexerhálózat megszerkesztését a meghajtott tároló elemhez legközelebb eső multiplexerrel, tehát a legkülső feltételvizsgálattal kezdjük, ami a legmagasabb prioritású vezérlő bemenetnek felel meg (lásd 8-2. ábra).



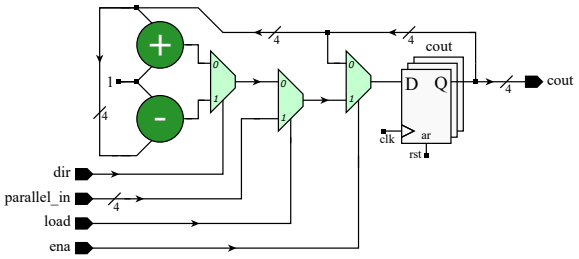
8-2. ábra. *A számláló engedélyezését/tiltását megvalósító áramköri részlet*

A számlálóhoz tartozó első multiplexer tehát gondoskodik arról, hogy ha az engedélyező bemenet alacsony értékű, akkor az órajel felfutó éle hatására a `cout` regiszterbe önmaga töltődjön be, vagyis a kimenet értéke ne változzon. Folytassuk a multiplexerhálózat elkészítését a prioritás alapján soron következő, párhuzamos betöltés funkciót megvalósító részletekkel (lásd 8-3. ábra).



8-3. ábra. *A párhuzamos betöltést megvalósító áramköri részlet*

Figyeljük meg, hogyan valósul meg a funkciók közötti prioritás; amíg az engedélyező bemenet alacsony értékű, addig a párhuzamos betöltést megvalósító áramköri részleteknek semmilyen szerep nem jut, a cout regiszter értékére csak akkor lesz hatással a második multiplexer, ha az ena bemenet magas értékű. Már csak egy megvalósítandó funkció van hátra;



8-4. ábra. *A számláló összes funkcióját megvalósító, teljes áramköri szerkezet*

VHDL

```
18 architecture rtl of fifo is
19
20   type content_t is array (0 to 255) of
21     std_logic_vector (7 downto 0);
22   signal content: content_t;
23
24 begin
25
26   process ( clk, rst )
27     variable read_pointer:  unsigned (7 downto 0);
28     variable write_pointer: unsigned (7 downto 0);
29   begin
30
31     if ( rst = '1' ) then
32       read_pointer := X"00";
33       write_pointer := X"00";
34       empty <= '1';
35       full <= '0';
36       dout <= X"00";
37     elsif ( rising_edge(clk) ) then
38
39       if ( push = '1' and full /= '1' ) then
40         content(to_integer(write_pointer)) <= din;
41         write_pointer := write_pointer + 1;
42         if ( write_pointer = read_pointer ) then
43           full <= '1';
44         else full <= '0';
45         end if;
46         empty <= '0';
47       end if;
48
49       if ( pop = '1' and empty /= '1' ) then
50         dout <= content(to_integer(read_pointer));
51         read_pointer := read_pointer + 1;
52         if ( write_pointer = read_pointer ) then
53           empty <= '1';
54         else empty <= '0';
55         end if;
56         full <= '0';
57       end if;
58
59     end if;
60
61   end process;
62
63 end architecture rtl;
```

Verilog

```
1  module fifo (...);
...
12   logic [7:0] content [0:255];
13   logic [7:0] read_pointer;
14   logic [7:0] write_pointer;
15
16   always @ (posedge clk, posedge rst)
17   begin
18
19     if ( rst ) begin
20       read_pointer <= 8'd0;
21       write_pointer <= 8'd0;
22       full <= 1'b0;
23       empty <= 1'b1;
24       dout <= 8'd0;
25     end else begin
26
27       if ( push && !full ) begin
28         content[write_pointer] <= din;
29         write_pointer = write_pointer + 8'd1;
30         if ( write_pointer == read_pointer )
31           full = 1'b1;
32         else full = 1'b0;
33         empty = 1'b0;
34       end
35
36       if ( pop && !empty ) begin
37         dout <= content[read_pointer];
38         read_pointer = read_pointer + 8'd1;
39         if ( write_pointer == read_pointer )
40           empty = 1'b1;
41         else empty = 1'b0;
42         full = 1'b0;
43       end
44
45     end
46
47   end
48
49 endmodule
```

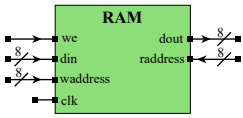
8-5. ábra. *FIFO tároló*

a `dir` bemenet állapotától függően a számláló értékét növelnünk vagy csökkentenünk kell. Az ezt az alacsony prioritású funkciót is megvalósító, immár teljes áramköri szerkezetet a 8-4. ábra szemlélteti.

A szintézis algoritmusok működésének megismerése szempontjából tanulságos lehet egy olyan példa áttekintése, amelyben a VHDL változóinak (`variable`) és a Verilog blokkoló értékadásainak viselkedését is kihasználjuk. Az előző példához hasonlóan vezessük le, hogy a 4.5.4. pontban bemutatott FIFO modell szintézise során hogyan áll elő a HDL kódból a 4-55. ábrán látható áramköri szerkezet.

Ezúttal is kezdjük azzal, hogy meghatározzuk, milyen tároló elemek lesznek a FIFO áramkörünkben:

- **`content[]`**: Regiszterek tömbje, amelynek tartalmát reset hatására nem inicializáljuk, és amelynek írását és olvasását egyaránt szinkron módon végezzük. E két tulajdonság lehetővé teszi, hogy ezt a tároló elemet memóriaként (pl. blokk-RAM-ként) valósítsuk meg. Tegyük fel, hogy a modellt FPGA technológiára szintetizáljuk, és használjunk egyidejűleg két külön interfészen írható és olvasható, ún. dual-port memóriát, mint amilyet a 4-51. ábrán látható példa is bemutat (8-6. ábra).



8-6. ábra. *Dual-port memória a* ***`content`*** *nevű regisztertömb megvalósításához*

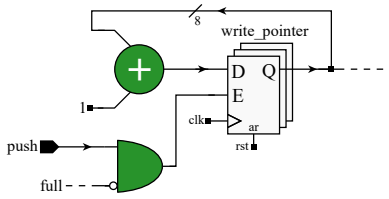
  A fenti memóriamodulról azt feltételezzük, hogy ha a `we` bemenetének értéke magas, akkor az órajel felfutó éle hatására a `din` bemeneten lévő adatot a `waddress` bemenet által megcímzett memóriarekeszben eltárolja. Ezzel párhuzamosan, ugyancsak az órajel felfutó éle hatására az `raddress` bemenet által megcímzett memóriarekesz tartamát a `dout` kimeneten megjeleníti.[68]
- **`dout`**: 8-bites regiszter, amely az órajel felfutó éle hatására a memória `dout` buszát a FIFO modul `dout` kimenetére másolja. Vegyük észre, hogy az `raddress` cím megváltozása után csak a második órajel-felfutóél hatására jelenik meg a FIFO `dout` kimenetén az adat. Ez az egy órajelciklus plusz késleltetés a címző logika működéséből adódóan a felhasználó számára nem lesz látható.
- **`read_pointer`**, **`write_pointer`**: A VHDL modellben változóként, a Verilog modellben pedig blokkoló értékadással használt jelek, amelyek a 4-24. ábrán látható esettel ellentétben ezúttal nem egyszerű vezetékként, hanem flip-flopokként valósulnak meg. A különbséget az okozza, hogy e jeleket még a rájuk vonatkozó értékadás előtt olvassuk (VHDL: 40. és 50. sor, Verilog: 28. és 37. sor), tehát olyan erőforrásoknak kell reprezentálniuk őket, amelyek képesek megőrizni az előző órajel-felfutóélnél beállt értéküket.

[68] Az egyazon címre történő egyidejű írás és olvasás esetével most nem foglalkozunk, ez ui. a modult körülvevő címző logika viselkedéséből adódóan – megfelelő felhasználás esetén – nem fog előfordulni.

- **`full`**, **`empty`**: Egybites regiszterek, amelyek a FIFO állapotát írják le. Ha a `full` állapotjelző magas, akkor a modul a `push` bemenet állapotát figyelmen kívül hagyja. Ha az `empty` állapotjelző magas, akkor a modul a `pop` bemenet értékét figyelmen kívül hagyja.

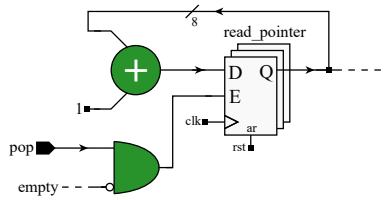
Vizsgáljuk meg az egyes tároló elemekre vonatkozó értékadásokat a szinkron folyamat törzsében, és szerkesszük meg a modell által leírt viselkedést megvalósító áramköri részleteket!

`write_pointer`: A `write_pointer`-be reset hatására nullát írunk (VHDL: 33. sor, Verilog: 21. sor). Ez a funkció a `write_pointer` jel aszinkron resettel ellátott regiszterként való megvalósításával automatikusan teljesül. A `write_pointer`-t az órajel felfutó élénél inkrementáljuk, ha a `push` bemenet értéke magas, a `full` állapotjelző értéke pedig alacsony (VHDL: 41. sor, Verilog: 29. sor). Ezt a viselkedést a 8-7. ábrán látható áramköri részlet valósítja meg.



8-7. ábra. *A FIFO **`write_pointer`** regiszterének viselkedését megvalósító áramköri részlet kapcsolási sémája*

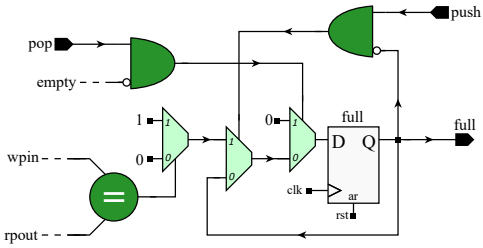
`read_pointer`: A `read_pointer`-be reset hatására nullát írunk (VHDL: 32. sor, Verilog: 20. sor). Ez a funkció a `read_pointer` jel aszinkron resettel ellátott regiszterként való megvalósításával automatikusan teljesül. A `read_pointer`-t az órajel felfutó élénél inkrementáljuk, ha a `pop` bemenet értéke magas, az `empty` állapotjelző értéke pedig alacsony (VHDL: 51. sor, Verilog: 38. sor). Ezt a viselkedést a 8-8. ábrán látható áramköri részlet valósítja meg.



8-8. ábra. *A FIFO **`read_pointer`** regiszterének viselkedését megvalósító áramköri részlet kapcsolási sémája*

`full`: A `full` regiszterbe reset hatására nullát írunk (VHDL: 35. sor, Verilog: 22. sor). Ez a funkció a `full` jel aszinkron resettel ellátott regiszterként való megvalósításával automatikusan teljesül. A `full` állapotjelző értékét a folyamaton belül több helyen is írjuk. A különálló értékadások között a folyamatok szimulációs szemantikája által meghatározott sorrendi jelleg olyan impliciten kifejezett prioritást eredményez, ahol a később leírt utasításnak magasabb a prioritása (vö. 4-45. ábra). Ez alapján belátható, hogy a `full` állapotváltozóba mindenképp nullát kell beírnunk, ha a `pop` bemenet értéke magas és az `empty` állapotjelző

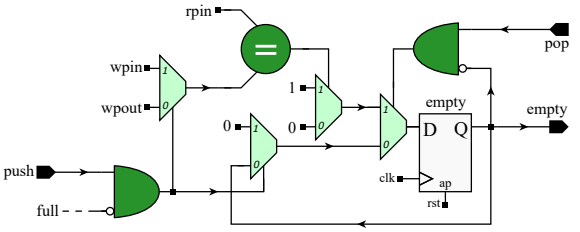
értéke alacsony (VHDL: 56. sor, Verilog: 42. sor). Ha ez a feltétel nem teljesül, akkor a `full` állapotjelzőre vonatkozó, a HDL modellekben előbb leírt értékadások fogják meghatározni a `full` állapotjelző új értékét. Ha megvizsgáljuk a `full` állapotjelzőre vonatkozó másik két értékadást (VHDL: 43. és 44. sor, Verilog: 31. és 32. sor), akkor a következőt állapíthatjuk meg: Ha a `push` bemenet alacsony értékű vagy a `full` állapotjelző magas, akkor a `full` állapotjelző tartja előző értékét. Ha a `push` bemenet magas értékű, a `full` állapotjelző pedig alacsony, továbbá a `write_pointer` értéke megegyezik a `read_pointer` értékével, akkor a `full` állapojelző új értéke magas lesz, ellenkező esetben pedig alacsony. Itt azonban észre kell vennünk, hogy amikor a VHDL modellben a 42., a Verilogban pedig a 30. sorban olvassuk a `write_pointer` értékét, akkor – mivel e jelet a VHDL modellben változóként deklaráltuk és az éppen vizsgált értékadások előtt frissítettük az értékét, a Verilog modellben pedig blokkoló értékadással, ugyancsak a vizsgált értékadás előtt frissítettük az értékét – valójában nem a `write_pointer` regiszter kimenetét olvassuk, hanem azt az értéket, amit a VHDL modellben a 41., a Verilogban pedig a 29. sorban értékül adtunk neki. Ez az érték az órajelnek ugyanannál a felfutó élénél fog csak beíródni a `write_pointer` regiszterbe, amelyiknél a `full` regiszter értékét is frissítjük, tehát áramkörileg a `full` regiszter bemenetét nem a `write_pointer` regiszter kimenete, hanem annak bemenete alapján kell meghatároznunk, ami ebben az időpillanatban már a `write_pointer` eggyel megnövelt értékét hordozza. Amikor a `write_pointer` és a `read_pointer` egyezőségét vizsgáljuk, akkor a `read_pointer` értékét még nem frissítettük, ami azt jelenti, hogy a `write_pointer`-nél látottakkal ellentétben itt a `read_pointer` regiszter kimenetét kell felhasználnunk. Mindezek alapján a `full` állapotjelző viselkedését megvalósító logika a 8-9. ábrán látható lesz. A `wpin` címke a `write_pointer` regiszter bemenetét, az `rpout` pedig a `read_pointer` regiszter kimenetét jelöli.



8-9. ábra. *A FIFO `full` regiszterének viselkedését megvalósító áramköri részlet kapcsolási sémája*

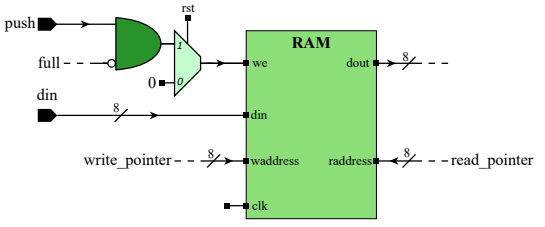
`empty`: Az `empty` regiszterbe reset hatására egyet írunk (VHDL: 34. sor, Verilog: 23. sor). Ez a funkció az `empty` jel aszinkron presettel ellátott regiszterként való megvalósításával automatikusan teljesül. Az `empty` állapotjelzőt szintén több ponton is írjuk a folyamaton belül, ezért itt is alulról felfelé haladva fejtjük vissza a prioritási sort. Ha a `pop` bemenet magas értékű és az `empty` állapotjelző alacsony, akkor ebben az esetben is meg kell vizsgálnunk a `write_pointer` és a `read_pointer` egyezőségét. Ha egyeznek, akkor egyet, ha különböznek, akkor nullát írunk az `empty` regiszterbe (VHDL: 52-54. sor, Verilog: 39-41. sor). Itt azonban ismét felmerül a kérdés, hogy a `write_pointer` és a `read_pointer` regiszternek mikor kell a bemenetét és mikor a kimenetét felhasználni az `empty` állapotjelzőre vonatkozó értékadást megvalósító logika megszerkesztésénél. A `read_pointer` értékét épp közvetlenül a kérdéses,

az `empty` állapotjelzőre vonatkozó értékadások előtt frissítjük (VHDL: 51. sor, Verilog: 38. sor), ezért a `read_pointer` esetén biztosan a regiszter bemenetét, tehát az inkrementált értéket kell használnunk, ugyanúgy, ahogy azt a `full` regiszter esetén is láttuk korábban. A `write_pointer` esetén azonban a helyzet bonyolultabb. A kérdéses feltételvizsgálatban (VHDL: 52. sor, Verilog: 39. sor) a `write_pointer` értéke attól függ, hogy azt inkrementáltuk-e korábban (VHDL: 41. sor, Verilog: 29. sor), vagy sem: Ha a VHDL modellben a 39., a Verilogban pedig a 27. sorban lévő feltétel igazra értékelődött ki, akkor a `write_pointer` inkrementált értékét, tehát a `write_pointer` regiszter bemenetét kell felhasználnunk az `empty` regiszter új értékének meghatározásánál, ellenkező esetben pedig `write_pointer` regiszter kimenetét, hiszen ekkor a nem inkrementált értékre van szükségünk. Az eddigi megállapításaink arra az esetre vonatkoznak, ha a `pop` bemenet magas értékű és az `empty` állapotjelző alacsony. Ha ez a feltétel nem teljesül, akkor az `empty` regiszter értékét a VHDL modellben az 46. a Verilogban pedig az 33. sorban lévő értékadás határozza meg a következőképpen: ha a `push` bemenet magas, a `full` regiszter pedig alacsony, akkor az `empty` regiszterbe nullát töltünk, ellenkező esetben az `empty` regiszter tartja az értékét. Mindezek alapján az `empty` állapotjelző viselkedését megvalósító logika a 8-10. ábrán lesz látható.



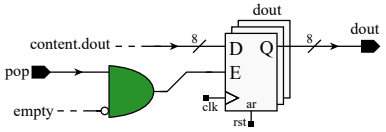
8-10. ábra. *A FIFO `empty` regiszterének viselkedését megvalósító áramköri részlet kapcsolási sémája*

`content`: Ezek után a `content` memória bemeneteit (`we`, `din`, `waddress`, `raddress`) meghajtó logika megszerkesztése már gyerekjáték. A `content` memóriát csak akkor írjuk, tehát akkor emeljük magas szintre a `we` bemenet értékét, ha nincs reset, és a `push` bemenet értéke nulla, a `full` állapotjelző értéke pedig alacsony. Ha ez nem teljesül, akkor a `we` bemenet értékét nullán tartjuk. A `waddress` bemenetre a VHDL modellben a 40. a Verilogban pedig az 28. sorban lévő értékadás alapján a `write_pointer`-t kapcsoljuk. Ugyanezen sorok alapján az is látható, hogy a memória `din` bemenetére egyszerűen a FIFO `din` bemenetét kell kötnünk. Az `raddress` bemenetre a VHDL modellben az 50. a Verilogban pedig az 37. sorban lévő értékadás alapján a `read_pointer`-t kapcsoljuk (lásd 8-11. ábra).



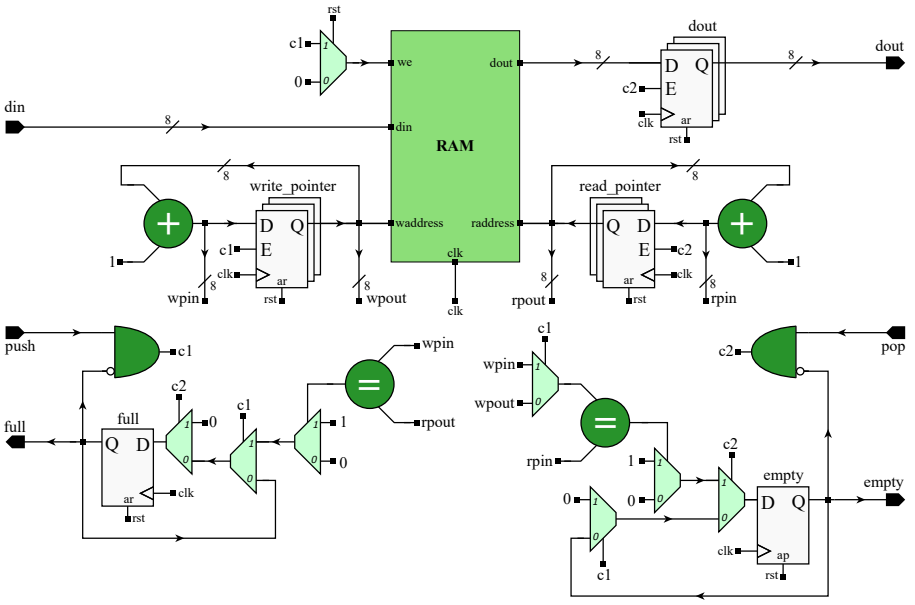
8-11. ábra. *A FIFO `content` memóriamoduljának viselkedését megvalósító áramköri részlet kapcsolási sémája*

dout: A dout regiszterbe reset hatására nullát írunk (VHDL: 36. sor, Verilog: 24. sor). Ez a funkció a dout jel aszinkron resettel ellátott regiszterként való megvalósításával automatikusan teljesül. Ezen kívül a dout regisztert a VHDL modellben csak az 50., a Verilogban a 37. sorban írjuk. Ha a pop bemenet értéke magas, az empty állapotjelző értéke pedig alacsony, akkor a dout regiszterbe a content memória adatkimenetét töltjük, ellenkező esetben a dout regiszter tartja előző értékét. A dout regiszter viselkedését megvalósító áramköri részlet a 8-12. ábrán látható.



8-12. ábra. *A FIFO **dout** regiszterének viselkedését megvalósító áramköri részlet kapcsolási sémája*

A teljesség kedvéért a 8-13. ábrán megismételtük a FIFO modul teljes kapcsolási rajzát.



8-13. ábra. *FIFO tároló teljes kapcsolási sémája*

Az áramkörszintézis alapproblémáit bemutató harmadik példánk az 4.8. pontban bemutatott, állapotgéppel megvalósított, iteratív akkumulátor lesz. Induljunk ki tehát a 8-14. ábrán látható HDL modellből.

A fenti HDL modell one-shot típusú állapotgép-leírás, tehát a belőle szintetizált áramkör szerkezete a 4-60. ábrán látható sémának fog megfelelni. A tároló elemek két kategóriáját különböztethetjük meg; az állapotgép mindenkori állapotát reprezentáló állapotregisztert, amelyet a HDL modellben a state jel képvisel, valamint a kimeneti buffereket; a

VHDL

```
19 architecture rtl of accumulator_fsm is
20
21   type state_t is ( s0, s1, s2, s3, s4 );
22   signal state: state_t;
23
24 begin
25
26   L_FSM: process ( clk, rst )
27   begin
28     if ( rst = '1' ) then
29       sum <= X"00";
30       zero <= '0';
31       finish <= '1';
32       state <= s0;
33     elsif ( rising_edge(clk) ) then
34       case state is
35         when s0     =>  if ( start = '1' ) then
36                           finish <= '0';
37                           sum <= a;
38                           state <= s1;
39                         end if;
40         when s1     =>  sum <= sum + b;
41                         state <= s2;
42         when s2     =>  sum <= sum + c;
43                         state <= s3;
44         when s3     =>  sum <= sum + d;
45                         state <= s4;
46         when s4     =>  if ( sum = X"00" ) then
47                           zero <= '1';
48                         else zero <= '0';
49                         end if;
50                         finish <= '1';
51                         state <= s0;
52         when others =>  null;
53       end case;
54     end if;
55   end process;
56
57 end architecture rtl;
```

Verilog

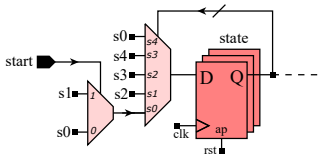
```
1  module accumulator_fsm (...);
...
10   typedef enum { s0, s1, s2, s3, s4 } state_t;
11   state_t state;
12
13   always @ ( posedge clk, posedge rst )
14   begin
15     if ( rst == 1'b1 ) begin
16       sum <= 8'd0;
17       zero <= 1'b0;
18       finish <= 1'b1;
19       state <= s0;
20     end else
21       case ( state )
22         s0: begin
23           if ( start == 1'b1 ) begin
24             finish <= 1'b0;
25             sum <= a;
26             state <= s1;
27           end
28         end
29
30         s1: begin
31           sum <= sum + b;
32           state <= s2;
33         end
34
35         s2: begin
36           sum <= sum + c;
37           state <= s3;
38         end
39
40         s3: begin
41           sum <= sum + d;
42           state <= s4;
43         end
44
45         s4: begin
46           if ( sum == 8'd0 ) zero <= 1'b1;
47           else zero <= 1'b0;
48           finish <= 1'b1;
49           state <= s0;
50         end
51
52         default:  ;
53       endcase
54   end
55
56 endmodule
```

8-14. ábra. *Az állapotgéppel megvalósított, iteratív akkumulátor HDL modellje*

`sum`, a `zero` és a `finish` regisztereket. E regiszterek vizsgálatát hasonlóan végezhetjük, mert mindegyikre igaz, hogy a HDL modellben az állapotregiszterre vonatkozó `case`-szerkezeten belül adunk nekik értéket; amikor tehát meg akarjuk állapítani, hogy az adott regiszterbe milyen értéket kell betölteni az órajel következő felfutó élénél, akkor az első kérdés mindig az, hogy éppen milyen állapotban van az állapotgép. A `state` regiszter aktuális kimenete tehát mindegyik regiszter esetén kulcsfontosságú lesz az adat- és órajel-engedélyező bemeneteket meghajtó kombinációs hálózatok szempontjából, beleértve magát a `state` regisztert is.

Kezdjük a vizsgálatot magával az állapotregiszterrel. Az RTL modellekben az állapotregisztereket az esetek többségében felsorolt típusú jelekkel modellezzük (VHDL: 21. sor, Verilog: 10. sor), amelyek bit-reprezentációinak meghatározása, vagyis az állapotkódolás az automatizált áramkörszintézis során történik meg. A leggyakoribb állapotkódolási

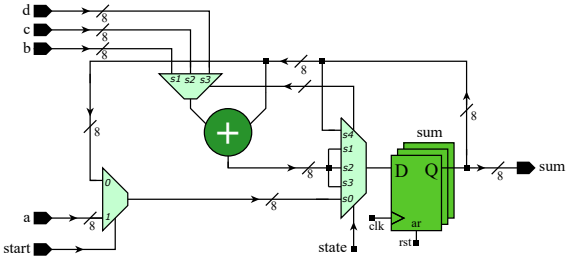
megoldás az ún. *„one-hot”*[69] kódolás, amely minden állapothoz egy olyan bináris értéket rendel, amelyben pontosan egy bit magas, a többi alacsony értékű. One-hot kódolás esetén tehát az állapotregiszterben pontosan annyi flip-flop van, ahány állapota van az állapotgépnek. Ez minket az áramkör alapvető szerkezetének meghatározásakor nem fog befolyásolni, de érdemes fejben tartani, hogy amikor az állapotregiszterről beszélünk, az fizikailag egy sokbites vektort jelent. Az állapotregiszter következő órajel-felfutóélnél felvett értékét elsősorban maga az állapotregiszter befolyásolja, hiszen a `state` jelre vonatkozó összes értékadás az ugyancsak a `state` jelre vonatkozó `case`-utasítás törzsében van (VHDL: 34. sor, Verilog: 21. sor). A `state` regiszter bemenetére tehát egy multiplexert kapcsolunk, amelynek kiválasztó bemenetére magának a `state` regiszternek az értékét csatoljuk vissza a 8-15. ábrának megfelelő módon, kifejezve, hogy a következő állapot attól függ, hogy épp most melyik állapotban van az állapotgép.



8-15. ábra. *A következőállapot-logika szerkezete*

A multiplexer bemenetei a példa egyszerűségéből adódóan könnyen meghatározhatók. Ha `s0` állapotban van az állapotgép, akkor a következő állapot a `start` bemenettől függ: ha a `start` bemenet magas, akkor az állapotgép az `s1` állapotba lép, ellenkező esetben marad `s0` állapotban (VHDL: 38. sor, Verilog: 26. sor). Az összes többi állapot minden egyéb körülménytől függetlenül mindig az eggyel nagyobb indexű állapotba irányítja az állapotgépet, az `s4` állapot pedig vissza az `s0` alapállapotba. A következőállapot-logika tehát elég egyszerűnek tűnik, de ne feledjük, hogy az állapotregiszter már ennél az egyszerű példánál is egy ötbites regiszter (one-hot állapotkódolást feltételezve). Az állapotregiszter bemenetét hajtó multiplexer tehát kapuszinten igen tekintélyes méretű lehet, hiszen annyi állapotdekódoló komparátort kell hogy tartalmazzon, ahány bemenete van.

A `sum` jelet a `state` jelhez hasonlóan szinte minden állapotban frissítjük, ezért a bemenetére egy, a `state` regiszternél látotthoz hasonló multiplexert érdemes kapcsolnunk (lásd 8-16. ábra).

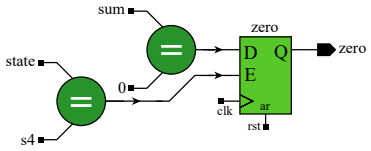


8-16. ábra. *A* `sum` *regiszter bemenetére kapcsolt logika szerkezete*

[69] Vigyázat! Nem összetévesztendő a „one-shot” típusú állapotgép-leírási módszerrel.

A `sum` regiszterbe töltött érték tehát elsősorban ugyancsak a `state` regisztertől függ. Ha `s0` állapotban van az állapotgép, akkor ezúttal is a `start` jel értékétől függ, hogy mi fog történni, ezért az állapotot dekódoló multiplexer `s0` bemenetére újabb multiplexert kapcsolunk, kiválasztó bemenetén a `start` jellel. Ha a start jel `s0` állapotban alacsony, akkor a `sum` regiszter értékét nem frissítjük, vagyis saját tartalmát csatoljuk vissza a bemenetére, mivel a HDL modellben nincs erre a helyzetre vonatkozó értékadás, ami implicit módon azt fejezi ki, hogy minden tároló tartja előző értékét. Ha a `start` jel `s0` állapotban magas, akkor a `sum` regiszterbe az `a` bemenet értékét másoljuk (VHDL: 37. sor, Verilog: 25. sor). Ha az állapotgép `s1`, `s2` vagy `s3` állapotban van, akkor a `sum` regiszterbe egy-egy összeadás eredményét töltjük. Ennek az összeadásnak az egyik operandusa mindig maga a `sum` regiszter, a másik pedig a konkrét állapottól függ, tehát az összeadó áramkör másik bemenetére újabb multiplexert kapcsolunk, amelynek kiválasztó bemenete ezúttal is az állapotregiszter; az `s1`, `s2` és `s3` állapotokban a `sum` regiszterhez rendre a `b`, `c` és `d` bemenetek értékét adjuk hozzá (VHDL: 40., 42. és 44. sor, Verilog: 31., 36. és 41. sor). Az `s4` állapothoz tartozó HDL kódrészletek nem tartalmaznak a `sum` regiszterre vonatkozó értékadást, ami ezúttal is azt jelenti, hogy annak tartania kell előző értékét. A multiplexer `s4` bemenetére tehát itt is a `sum` regiszter értékét csatoljuk vissza.

A `zero` regiszter bemenetét meghajtó logikát a korábbiaktól kissé eltérő módon szerkesztjük meg, mert ennek a regiszternek az értékét kizárólag az `s4` állapotban frissítjük (VHDL: 47. és 48. sor, Verilog: 46. és 47. sor), így jól ki tudjuk használni a `zero` regiszter órajel-engedélyező bemenetét (lásd 8-17. ábra).



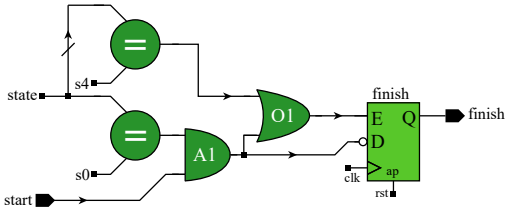
8-17. ábra. *A* ***zero*** *regiszter bemenetére kapcsolt logika szerkezete*

A `zero` regiszter tehát csak akkor kap engedélyező jelet, ha az `s4` állapotban van az állapotgép. Ezt egy különálló állapotdekóderrel valósítjuk meg.[70] Az adat, amit az `s4` állapotban a `zero` regiszterbe töltünk, a `sum` regiszter aktuális értékétől függ (VHDL: 46. sor, Verilog: 46. sor); logikai magas, ha a `sum` értéke nulla, alacsony egyébként. Ezt a feladatot ugyancsak egy komparátor tudja ellátni.

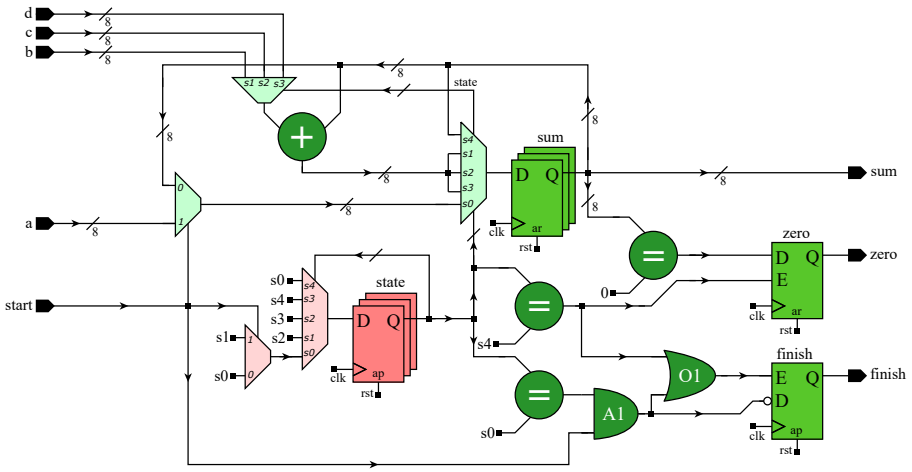
A `finish` regisztert meghajtó logika esetén hasonlóan járunk el (lásd 8-18. ábra).

A `finish` regisztert csak az `s4` és `s0` állapotokban frissítjük (VHDL: 36. és 50. sor, Verilog: 24. és 48. sor), ez utóbbiban csak akkor, ha a `start` jel magas értékű. A `finish` regiszter órajel-engedélyező bemenetének meghajtásához tehát szükségünk van két állapotdekódoló komparátorra, valamint az A1 és O1 logikai kapukra. A `finish` regiszter adatbemenetét meghajtó logika megszerkesztéséhez a következőképpen gondolkodhatunk: Ha az állapotgép `s4` állapotban van, akkor logikai magas szintet kell a `finish` regiszterbe írnunk (VHDL:

[70] Azok a multiplexerek, amelyek kiválasztó bementére az állapotregisztert kötjük, ehhez hasonló állapotdekóderekből annyit tartalmaznak, ahány adatbemenetük van.



8-18. ábra. *A `finish` regiszter bemenetére kapcsolt logika szerkezete*



8-19. ábra. *Az állapotgéppel megvalósított, iteratív akkumulátor teljes kapcsolási sémája*

50. sor, Verilog: 48. sor). Ha az állapotgép `s4` állapotban van, akkor az `s0` állapotra érzékeny komparátor kimenete mindenképpen alacsony, tehát az A1 kapu kimenete is alacsony, így a regiszterbe – a negált adatbemenet miatt – logikai magas szint íródik, ami megfelel az elvárásainknak. Ha az állapotgép `s0` állapotban van és a `start` jel logikai magas szintű, akkor a `finish` regiszterbe alacsony szintet kell írnunk (VHDL: 36. sor, Verilog: 24. sor). Ebben a helyzetben az A1 kapu kimenete magas, tehát a negált adatbemeneten keresztül a regiszterbe – az elvárásainknak ezúttal is megfelelően – alacsony szint íródik.

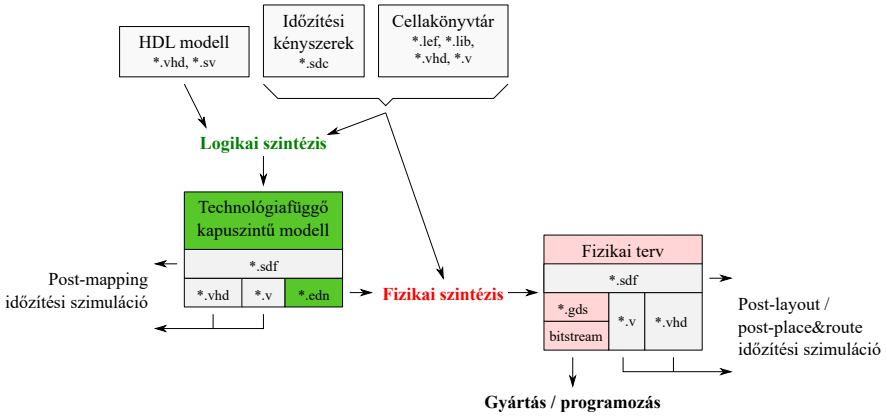
A 8-19. ábra az akkumulátor áramkör teljes kapcsolási sémáját mutatja be. Figyeljük meg, hogy a `zero` és `finish` regiszterek bemenetét meghajtó logikák megosztva használják az `s4` állapotra érzékeny állapotdekódoló komparátort.

8.2 Automatizált áramkörszintézis

A fent bemutatott egyszerű példák jól szemléltetik, hogy miért célszerű az áramkörszintézist gépi úton elvégezni. E példák esetén a szintézis folyamatának legelső lépését *(elaboration, translation)* hajtottuk végre manuálisan, aminek a célja csupán annak bemutatása volt, hogy miként születhet meg egy strukturális, áramköri elemeket és összeköttetéseiket

leíró modell egy HDL leírás alapján. Ezek a modellek azonban még mindig elég elvontak, önmagukban is összetett kombinációs hálózatokat és tároló elemeket jelképező szimbólumokat tartalmaznak, amelyek kifejtésére és optimalizációjára a szintézis folyamatának további lépéseiben kerül sor. Már e köztes, részben RTL, részben kapuszintű, de már egyértelműen strukturális jellegű modelleknek a kézi előállítása is sok befektetett munkát és időt igényel, ráadásul a folyamat rengeteg hibalehetőséget is rejt, nem csoda tehát, hogy a mai mérnöki gyakorlatban a legritkább esetben végezzük az áramkörszintézist kézi úton. Erre a feladatra több évtizede létező és fejlődő, mára már igen kiforrott szoftver eszközök állnak rendelkezésünkre.

A 8-20. ábra az automatizált áramkörszintézis folyamatának vázlatát mutatja be.



8-20. ábra. *Az automatizált áramkörszintézis folyamatának vázlata*

Az automatizált áramkörszintézis tehát két fázisból, a logikai szintézisből és a fizikai szintézisből áll. A logikai szintézis során a regiszter-transzfer szintű, hardverleíró nyelvű modellből egy ún. technológiafüggő kapuszintű modellt, netlistát állítunk elő. E netlistáról készülhet HDL reprezentáció szimulációs célra, de a további szintézis-algoritmusok bemenetéül leggyakrabban az ún. EDIF *(Electronic Design Interchange Format)* formátum (*.edn) használatos. A „technológiafüggő" kifejezés arra utal, hogy ez a netlista már a megvalósítás alapjául szolgáló cellakönyvtár alapelemeiből, logikai kapukból, flip-flopokból, multiplexerekből stb. felépülő kapcsolási séma. Ugyanakkor a szintézisnek ez a lépése még „csak logikai", mert ez a modell még nem tartalmaz információt sem az alkatrészek elhelyezkedésére, sem az őket összekapcsoló vezetékhálózatra vonatkozóan, tehát tulajdonképpen a technológiafüggő kapuszintű netlista egy ismert késleltetési, fogyasztási és geometriai tulajdonságokkal bíró alkatrészekből álló, absztrakt kapcsolási séma.

Fizikai szintézis alatt a megvalósítás alapjául szolgáló technológiától függően eltérő dolgokat értünk:

- **Standard cellás ASIC** esetén a fizikai tervezés a chip globális szerkezetének (floorplan-jének) megtervezését, majd a különböző cellák elhelyezését és a köztük lévő vezetékezés megtervezését értjük.

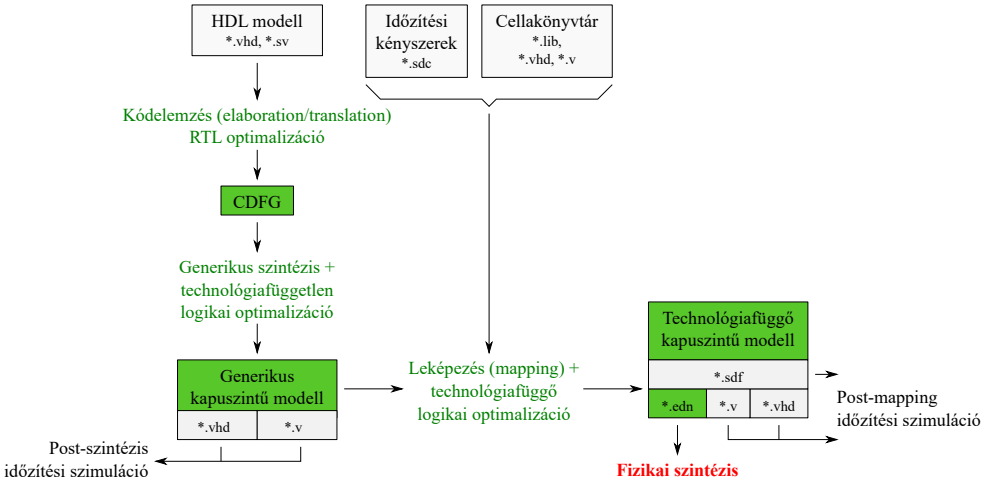
- **Programozható logikai eszközök**, például FPGA-k esetén a fizikai terv, sőt valójában a legyártott eszköz is a rendelkezésünkre áll, itt tehát nincs értelme a chip szerkezetének, a cellák elhelyezkedésének és a vezetékhálózat sajátosságainak megtervezéséről beszélni. PLD-k esetén a fizikai tervezés alatt azt értjük, hogy a technológiafüggő kapuszintű modell egyes elemeihez az adott céleszköz konkrét logikai elemeit – LUT-jait, flip-flopjait – rendeljük hozzá, majd az architektúra által rendelkezésünkre bocsátott huzalozási erőforrásokat, kapcsolómátrixokat felkonfiguráljuk úgy, hogy a logikai tervünknek megfelelő áramkört valósítsák meg.

A fizikai szintézis a tervezési folyamat végét jelenti abban az értelemben, hogy az eredményül előálló adatbázis standard cellás ASIC technológia esetén a gyártáshoz, PLD technológia esetén pedig a konkrét céleszköz felkonfigurálásához szükséges összes információt tartalmazza.

8.2.1 Logikai szintézis

A logikai szintézis elsődleges bemenete természetesen maga a HDL modell, de azon kívül egyéb információra is szükség van. Ahogy a 6. fejezetben is említettük, a szintézis algoritmusok képesek figyelembe venni a leendő áramkörrel szemben támasztott időzítési követelményeket, tehát az SDC formátumban rögzített időzítési kényszerek a HDL modell mellett ugyancsak a folyamat bemenetének részét képezik. Az időzítési kényszereket számszerűen figyelembe vevő algoritmusokat alkalmazó folyamatot időzítésvezérelt *(timing-driven)* szintézisnek nevezzük.

Az időzítésvezérelt logikai szintézis folyamatának részleteit a 8-21. ábra szemlélteti.



8-21. ábra. *A logikai szintézis folyamata*

A korai szintézis-eljárások nem konkrét időzítési követelmények alapján dolgoztak, ezek esetén a tervező csupán két lehetőség közül választhatott; vagy minimális erőforrásigényre, ez által minimális fogyasztásra, vagy maximális órajel-frekvenciára, vagyis maximális számítási teljesítményre törekedhetett. E két véglet között standard cellás technológia esetén jelentős különbség lehet, mivel egy standard cellakönyvtár az egyazon funkciót megvalósító alapcellákból meglehetősen sokat tartalmazhat, és ezeknek a funkcionálisan azonos alapcelláknak a geometriai, fogyasztási és időzítési tulajdonságai nagy mértékben eltérhetnek. Ha tehát kis erőforrásigényre törekszünk – ami a kis fogyasztással és kis területtel egyenértékű –, akkor a szintézis algoritmus a kis meghajtóképességű és kis méretű cellákat példányosítja, ha viszont nagy frekvenciát szeretnénk elérni a számítási teljesítmény növelése érdekében, akkor a szintézis során nagy meghajtóképességű, nagy méretű cellákat kell használnunk. PLD technológia esetén e két megoldás közötti különbség kevésbé jelentős, mivel a PLD-k alapelemkészlete kevésbé sokszínű. FPGA-k esetén például nem beszélhetünk egyszeres vagy kétszeres meghajtóképességű flip-flopokról, ha tehát egy adott flip-flop terhelése az átlagosnál nagyobb, akkor legfeljebb azt tehetjük, hogy egy helyett két darabot használunk belőle, és e két cella bemeneteit és kimeneteit egyaránt összekötjük (regiszter-duplikálás).

Az időzítésvezérelt szintézis módszerek a szélsőséges, „abszolút minimális terület vagy abszolút maximális frekvencia" megközelítés helyett a konkrét időzítési kényszereket figyelembe véve azt a minimális méretű struktúrát igyekeznek létrehozni, amely még éppen képes teljesíteni az időzítési előírásokat. Ezt iteratív úton érik el, úgy, hogy a legkisebb lehetséges megvalósításból kiindulva a kritikus út, vagyis a legnagyobb késleltetésű lokális adatút celláinak meghajtóképességét addig növelik, amíg az adott adatútra vonatkozó időzítési feltételek nem teljesülnek. Ha egy lokális adatútra vonatkozóan sikerült teljesíteni az időzítési feltételeket, akkor az az adatút többé már nem kritikus, az algoritmus tehát folytathatja a következő kritikus úttal. A folyamat akkor ér véget, ha az áramkör minden lokális adatútja megfelel az időzítési előírásoknak.

Jogosan merülhet fel a kérdés, hogy a logikai szintézis során miféle időzítési információkra támaszkodik egy ilyen szintézis algoritmus, hiszen a lokális adatutak késleltetései éppen ennek és az ezt követő további szintézis lépéseknek az eredményeként alakulnak ki. A logikai szintézis az alapcellák késleltetéseit pontosan ismeri, ezeket ui. a cellakönyvtár vagy PLD technológia esetén az adott eszköz gyártója karakterizálja. Standard cellás technológia esetén a cellakönyvtár időzítési paramétereit az ún. Liberty-adatbázis tartalmazza, PLD-k esetén pedig a Liberty-hez hasonló, gyártófüggő modellek. A 6. fejezetben azonban láthattuk, hogy a lokális adatutak késleltetéséért csak részben felelnek maguk a logikai cellák, ahhoz a közöttük lévő vezetékezés éppúgy hozzájárul. Ezen a ponton a szintézis algoritmusok – mivel a vezetékezésről a logikai szintézis során még semmi információ nem áll rendelkezésükre – becslésekre kénytelenek hagyatkozni. Többféle becslési módszer ismert, ezek közül a legalapvetőbb az ún. *„wire-load"* modell, amely az egyes kapukat terhelő vezetékhálózat késleltetését az adott kapu fan-out-ja, vagyis a meghajtott kapubemenetek száma alapján becsüli. Egy lokális adatút vizsgálata során a vezetékezés által hozzáadott késleltetésösszetevő a meghajtó flip-flop fan-outjának monoton növekvő függvénye, az arányossági tényező pedig egy technológiára, vagy PLD-k esetén az eszközcsaládra jellemző állandó.

8.2.1.1 Kódelemzés és RTL optimalizáció

Ahogy minden formális nyelvű modell esetén, a gépi úton történő feldolgozás első lépése a HDL nyelvű RTL leírásoknál is a nyelvtani elemzés, az ún. szintaxisfa elkészítése. E folyamat során derül fény az esetleges szintaktikai hibákra. Ezután a kód előkészítése, szerkezeti egyszerűsítése következik, amely során a függvényhívások helyére azok implementációját helyettesítjük be *(expansion)*, a ciklusokat egyszerű értékadásokká alakítjuk (*loop unrolling*, vö. 4-27. és 4-28. ábra), majd behelyettesítjük és összevonjuk a szintézis-paramétereket *(generic folding)* és a konstansokat *(constant folding)*.

A nyelvtani elemzést és a kód előkészítését követően a HDL modell alapján egy ún. vezérlési/adatfolyam gráf (*Control/Dataflow Graph*, CDFG) készül. A 8-22. ábra egy egyszerű HDL kódrészletből generált vezérlési/adatfolyam gráfot szemléltet.

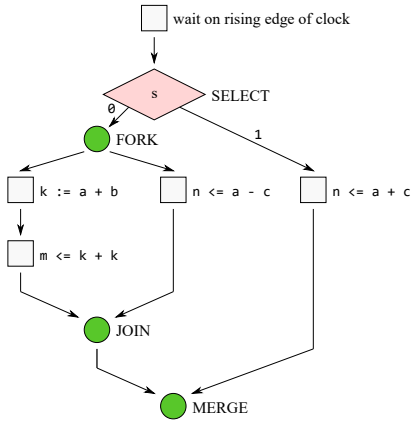
VHDL

```
process ( clk )
begin
  if ( rising_edge(clk) ) then
    if ( s = '0' ) then
      k := a + b;
      m <= k + k;
      n <= a - c;
    else
      n <= a + c;
    end if;
  end if;
end process;
```

Verilog

```
always @ ( posedge clk )
begin
  if ( s == 1'b0 ) begin
    k = a + b;
    m <= k + k;
    n <= a - c;
  end else
    n <= a + c;
end
```

CDFG



8-22. ábra. *Vezérlési/adatfolyam gráf*

A vezérlési/adatfolyam gráf egy körmentes (aciklikus), irányított gráf, amely információtartalmát tekintve magával a HDL modellel egyenértékű és amelyen a szintézis szoftver algoritmusai az egyes optimalizációs műveleteket végrehajtják. Ez a reprezentáció képezi tehát az alapját annak, hogy az itt bemutatott optimalizációs eljárások algoritmikusan kifejezhetők legyenek. Maguknak az eljárásoknak a magyarázatához azonban a továbbiakban nem a CDFG-reprezentációt, hanem HDL kódrészleteket vagy grafikus eszközöket, kapcsolási sémákat fogunk használni.

Erőforrás-megosztás • A vezérlési/adatfolyam gráfon végzett, egyik legelső és leghatékonyabb optimalizációs eljárás az ún. erőforrás-megosztás *(resource-sharing)*, amellyel az adatfeldolgozó rendszerek RTL modelljeivel kapcsolatban már találkoztunk a 4.8. pontban bemutatott iteratív akkumulátornál. Az erőforrás-megosztás az aritmetikai operátorokra vonatkozó, magas szintű optimalizációs módszer, amely során a szintézis algoritmus megállapítja, hogy a HDL leírásban szereplő, különálló aritmetikai műveletek végrehajtásához

lehetséges-e egyetlen aritmetikai műveletvégző elemet lefoglalni az erőforrásigény csökkentése céljából. Erre csak akkor kerülhet sor, ha a HDL modellben megfogalmazott logika alapján egyazon órajel-felfutóélnél a két, különálló aritmetikai művelet eredményét soha nem használjuk fel egyszerre. Ezt a szituációt mutatja be a 8-23. ábra.

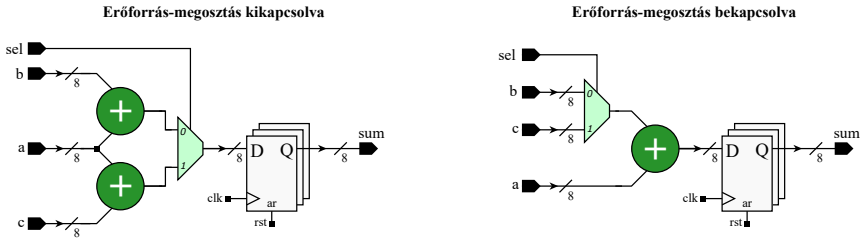
VHDL

```
process ( clk, rst )
begin
  if ( rst = '1' ) then sum <= X"00";
  elsif ( rising_edge(clk) ) then
    if ( sel = '0' ) then sum <= a + b;
    else sum <= a + c; end if;
  end if;
end process;
```

Verilog

```
always @ ( posedge clk, posedge rst )
begin
  if ( rst == 1'b1 ) sum <= 8'd0;
  else
    if ( sel == 1'b0 ) sum <= a + b;
    else sum <= a + c;
end
```

Áramköri struktúra



8-23. ábra. *Erőforrás-megosztás (1)*

A kapcsolási sémák alapján arra a következtetésre juthatunk, hogy a szintézis algoritmusok automatikus erőforrás-megosztás funkcióját mindenképpen érdemes kihasználni, hiszen az áramkörünk erőforrásigényét úgy csökkenti, hogy közben semmilyen más paramétert nem ront el. Ezt a kijelentést az RTL tervezéssel kapcsolatos eddigi tapasztalatai alapján alighanem az Olvasó már maga is gyanakvással fogadja. És valóban, a 8-24. ábrán látható példa azt támasztja alá, hogy ez a gyanakvás nem alaptalan.

Ebben a példában a korábbiakhoz hasonlóan az erőforrás-megosztás ugyancsak egy összeadó áramkör megspórolását tette lehetővé, ugyanakkor az áramkör lokális adatútjai is módosultak kissé. Az erőforrás-megosztás nélküli struktúrában a leghosszabb adatutak csak egy összeadót foglaltak magukba, az erőforrás-megosztást bekapcsolva azonban az „újrafelhasznált” összeadó egyik bemenetét multiplexálnunk kell. Ahhoz, hogy az RTL modell funkciója ne változzon, ez a multiplexer az összeadóval egy közös lokális adatutat kell, hogy képezzen, ami a késleltetést meg fogja növelni. Általánosságban az mondható el, hogy az erőforrás-megosztás az operandusok multiplexálásának szükségessége miatt a lokális adatutak késleltetésének megnövekedésével, tehát a maximális órajel-frekvencia csökkenésével járhat. Mivel azonban ez nem minden esetben igaz, ezért az egyes konkrét szituációkat ebből a szempontból egyedileg kell megvizsgálni az elérhető legjobb eredmény érdekében.

8.2.1.2 Generikus szintézis és technológiafüggetlen logikai optimalizáció

A vezérlési/adatfolyam gráfból előállított, első kapuszintű reprezentáció az ún. generikus kapuszintű modell, generikus netlista. Az elnevezés arra utal, hogy ezt a modellt általános

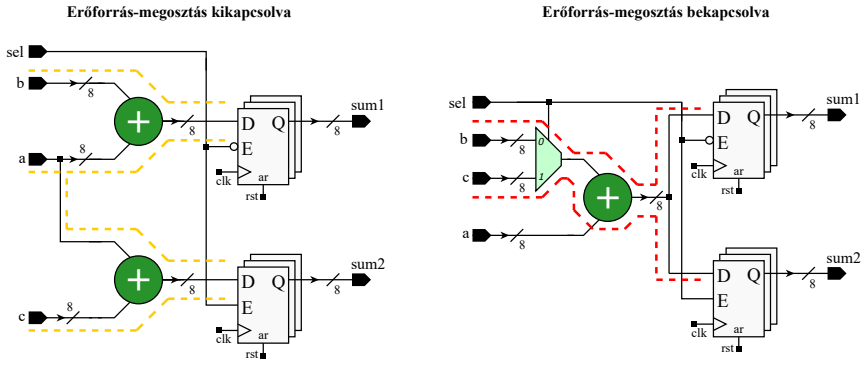
VHDL

```
process ( clk, rst )
begin
  if ( rst = '1' ) then
    sum1 <= X"00"; sum2 <= X"00";
  elsif ( rising_edge(clk) ) then
    if ( sel = '0' ) then sum1 <= a + b;
    else sum2 <= a + c; end if;
  end if;
end process;
```

Verilog

```
always @ ( posedge clk, posedge rst )
begin
  if ( rst == 1'b1 ) begin
    sum1 <= 8'd0;
    sum2 <= 8'd0;
  end else
    if ( sel == 1'b0 ) sum1 <= a + b;
    else sum2 <= a + c;
end
```

Áramköri struktúra



8-24. ábra. *Erőforrás-megosztás (2)*

logikai funkciókat megvalósító, absztrakt komponensek, AND, OR, XOR kapuk, flip-flopok és memóriák alkotják. E köztes modellen végezzük el a technológiafüggetlen logikai optimalizációt, vagyis kihasználjuk azokat az egyszerűsítési lehetőségeket, amelyek nem a megvalósítás alapjául szolgáló cellakönyvtárban, hanem az áramkörünkben mint logikai rendszerben rejlenek.

Állapotkódolás • A generikus netlista előállításához először is regiszter-szinten teljessé kell tennünk az RTL modellt, vagyis a bináris reprezentációval nem rendelkező, felsorolt típusokhoz konkrét bináris értékeket kell rendelnünk. Ezt a folyamatot állapotkódolásnak hívjuk, mivel a felsorolt típusokat rendszerint – bár nem kizárólag – állapotgépek állapotregiszterének modellezéséhez használjuk (lásd 4.5.5. pont). A legelterjedtebb állapotkódolási megoldások a következők:

- **Szekvenciális**: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111...
  A felsorolt típus különböző lehetséges értékeihez egész számok bináris reprezentációit rendeljük. Ez az állapotkódolási megoldás használja fel a lehető legkevesebb flip-flopot adott értékkészletű felsorolt típus megvalósításához. Mivel azonban az állapotkódolás megválasztása nemcsak az állapotregiszter méretére, hanem az állapotdekóderek méretére is igen nagy hatással van, és ebből a szempontból a szekvenciális kódolás nagyméretű állapotvektorok esetén nem hatékony, ezért ezt a megoldást csak kisméretű, jellemzően ötnél kevesebb állapottal rendelkező állapotgépek esetén alkalmazzuk.

- **Gray**: 0000, 0001, 0011, 0010, 0110, 0111, 0101...
  Jellemzője, hogy számlálós vezérlés esetén két egymást követő állapot között a Hamming-távolság mindig egy (vö. 4.4.4. pont: Hamming-enkóder/dekóder), így a kimeneti logika esetleges hazárdjai nem tudnak glitch-eket okozni a vezérlő vonalakon.
- **Hamming-2**: 00001, 00010, 00100, 00111, 01000, 01011...
  Bármely két kódszó között a Hamming-távolság mindig kettő, így az egybites hibák – vagyis amikor az állapotregiszter flip-flopjai közül pontosan egy hibás értéket vesz fel – mindig detektálhatók. Ennek nagymegbízhatóságú, különösen űripari alkalmazásokban van jelentősége az ún. SEU *(Single-Event Upset)* jelenségek miatt, amikor egy nagyenergiájú részecske becsapódása miatt egy flip-flop váratlanul állapotot vált (lásd még: 9.1.4. pont).
- **One-hot**: 00001, 00010, 00100, 01000...
  Bármely érvényes állapotkód csupán egyetlen helyiértéken tartalmaz logikai magas szintet. Ebben az esetben is igaz, hogy az állapotregiszter egybites hibái detektálhatók, de megvalósítása több flip-flopot igényel, mint a minimális Hamming-2 kódolásé. A one-hot kódolás legfőbb előnye, ami miatt (különösen PLD technológián) ez a leggyakrabban alkalmazott állapotkódolási módszer, hogy ha feltételezzük, hogy az állapotregiszter nem vehet fel érvénytelen állapotot (ami az alkalmazások többségében jogos feltételezés), akkor a következő-állapot logika és a kimeneti logika mérete a lehető legkisebb. Annak megállapítása, hogy az állapotregiszter egy bizonyos értéket vett fel, tulajdonképpen nem igényel semmilyen kombinációs logikát; csak meg kell vizsgálnunk az állapotregiszternek azt a bitjét, amelyik az adott állapothoz tartozó állapotkódban magas értékű. Ha ez a bit magas, akkor az állapotgép a keresett állapotban van. Ez azt jelenti, hogy a one-hot állapotkódolással megvalósított állapotgépek dekódereinek erőforrásigénye igen kedvező, ami végső soron a lokális adatutak késleltetésére, ezáltal az áramkör maximális órajel-frekvenciájára is jótékony hatással van.
- **Hamming-3**: Nagymegbízhatóságú rendszerekben alkalmazott állapotkódolási megoldás, amelynek jellemzője, hogy bármely két érvényes állapotkód között a Hamming-távolság minimum három, ami azt jelenti, hogy ha egybites hibákat feltételezünk (lásd SEU), akkor az állapotregiszter hibái nemcsak detektálhatók, de akár menet közben javíthatók is. Egyes szintézis szoftverek képesek az állapotgépek szintézise során olyan hibadetektáló logikát szintetizálni az állapotgép mellé, amely felismeri, ha az állapotregiszter valamilyen külső esemény hatására hibás értéket vett fel, és az órajel következő lefutó élénél visszaírja az állapotregiszterbe a helyes értéket.[71]

[71] Ez a technológia ma (2020) még nem nevezhető széles körben elterjedtnek, ráadásul elvi kérdéseket is felvet, főként az áramkör ütemezési és időzítési viszonyainak vizsgálata szempontjából. Ha az állapotregiszter vizsgálatát végző és az állapotgépet érvényes állapotba visszajuttató logikát a szintézis szoftver építi be az áramkörbe, akkor valójában olyan funkciót valósít meg ezzel, amit az RTL modell nem tartalmazott. Sérül tehát az az elv, amely szerint a szintetizált netlista egy zero-delay logikai szimuláció során bármely gerjesztésre az órajel megfelelő éleinél a kimenetein pontosan ugyanazokat az értékeket állítja elő, mint az RTL modell. A kifejezetten az állapotregiszterek hibáit modellező tesztesetekre e két modell nem egyformán fog reagálni. Emiatt az RTL tervezők körében az ilyen optimalizációs módszerek kevésbé népszerűek, nagymegbízhatóságú alkalmazásokban a hibás állapotok detektálásáért felelős logikát tipikusan manuálisan építjük be, szerves részévé téve azt az RTL modellnek.

Az állapotkódolást a szintézis szoftverek képesek akár teljesen automatikusan is elvégezni. Ekkor rendszerint az állapotregiszter flip-flopjainak a száma alapján döntenek; a néhány flip-flopból álló állapotregisztereket szekvenciális, az annál nagyobbakat one-hot kódolással szintetizálják. A tervezőnek természetesen van lehetősége beavatkozni ebbe. A szintézis szoftver konfigurálásával megfogalmazhatunk globális irányelvet az állapotkódolásra vonatkozóan, ekkor a szintézer a tervünk összes állapotgépét egyazon kódolással valósítja meg, de a HDL modellbe ágyazott, a szintézis szoftvernek szóló „iránymutatásokkal" az egyes állapotgépekre egyedileg is megfogalmazhatjuk a kódolásra vonatkozó igényeinket. Ezeket az iránymutatásokat VHDL-ben a nyelv részét képező, az egyes nyelvi szerkezetekhez rendelhető meta-adatok, az ún. attribútumok, Verilogban pedig a kód megfelelő részére és megfelelő formátumban elhelyezett megjegyzések, a pragmák valósítják meg[72] (lásd 8-25. ábra).

VHDL

```
type state_t is ( s0, s1, s2, s3, s4 );
signal state: state_t;
attribute syn_encoding: string;
attribute syn_encoding of state:signal is "one-hot";
```

Verilog

```
typedef enum logic [4:0] { s0, s1, s2, s3, s4 }  state_t;
state_t state; /* synthesis syn_encoding = "one-hot" */
```

8-25. ábra. *VHDL attribútum és Verilog pragma az állapotkódolás rögzítéséhez*

Az állapotkódolásra vonatkozó attribútumok és pragmák persze csak akkor használhatók, ha olyan állapotkódolást szeretnénk használni, amit az adott szintézis szoftver ismer. Ha egyedi állapotkódolást akarunk megvalósítani, akkor az állapotregisztert nem felsorolt típusú jelként, hanem egyszerű logikai vektorként írjuk le, amelynek különböző értékeit konstansokként definiáljuk. E konstansok definiálása tulajdonképpen maga az állapotkódolás, hiszen ezek határozzák meg, hogy az egyes állapotoknak az állapotregiszternek mely bináris kombinációja felel meg (lásd 8-26. ábra).

VHDL

```
constant s0:  std_logic_vector (3 downto 0) := "0001";
constant s1:  std_logic_vector (3 downto 0) := "0010";
constant s2:  std_logic_vector (3 downto 0) := "0100";
constant s3:  std_logic_vector (3 downto 0) := "0111";
constant s4:  std_logic_vector (3 downto 0) := "1000";
signal state: std_logic_vector (3 downto 0);
```

Verilog

```
parameter s0: 4'b0001;
parameter s1: 4'b0010;
parameter s2: 4'b0100;
parameter s3: 4'b0111;
parameter s4: 4'b1000;
logic [3:0] state;
```

8-26. ábra. *Állapotkódolás kézi definiálása*

Regiszter-optimalizációk ⬩ A fejezet elején bemutatott példákon manuálisan elvégzett áramkörszintézis során eredményül kapott köztes, részben RTL, részben kapuszintű modellekre tekintve azt vehetjük észre, hogy a leendő áramkör tároló elemeit az RTL modell valójában elég pontosan rögzíti, így az automatizált szintézisnek a lényegi feladata tulajdonképpen a tároló elemek közötti kombinációs hálózatok létrehozása. Valóban, az

[72] Az attribútumokat és pragmákat a szintézis szoftver definiálja, ami azt jelenti, hogy ezek használata a kódbázist részben szintézis szoftver- és/vagy technológiafüggővé teszi.

áramkör flip-flopjainak száma az állapotkódolás rögzítése után elméletileg akár az RTL modell ismeretében is egyértelműen megállapítható, vannak azonban olyan, automatikus optimalizálási eljárások, amelyek nemcsak a regiszterek közötti kombinációs hálózatokat, hanem magukat a regisztereket is érintik.

A HDL modellben előfordulhatnak olyan regiszterek, amelyeknek funkcionálisan semmi szerepük nincs. Ezeket a regisztereket a szintézis algoritmus eltávolítja *(register pruning)*. Ez a szituáció nemcsak hibás HDL kódolás következményeként állhat elő. Ha újrafelhasználható IP magokból építkezünk, akkor gyakran előfordul, hogy az adott IP bizonyos funkcióira nincs szükségünk, és az ezekhez tartozó bemeneteket a példányosításkor konstans logikai értékre kapcsoljuk,[73] a kimeneteket pedig egész egyszerűen nem kötjük be sehová. Ilyenkor az IP RTL modellje egy sor olyan regisztert tartalmazhat, ami felesleges az adott toplevel modellben, a szintézis során tehát nem kell megvalósítani. Az is elképzelhető, hogy az RTL modell logikáját követve létrehozunk két, látszólag teljesen eltérő regisztert, amelyek a logikai optimalizációk során ekvivalensnek bizonyulnak. Ekkor a szintézis szoftver az erőforrásigény csökkentése céljából fizikailag csak egy regisztert épít be az áramkörbe.

Csakúgy, mint az állapotkódolás, ez az optimalizációs eljárás is befolyásolható a szintézis szoftver globális beállításain vagy attribútumokon és pragmákon keresztül, a látszólag felesleges regiszterek eltávolítását tehát meg tudjuk akadályozni. A fent leírt szituációkban persze ennek sok értelme nincs, van azonban egy olyan helyzet, amikor a regiszterek eltávolításának hasznossága korántsem ennyire magától értetődő. Az állapotgépek modellezése során a HDL-ek `case` szerkezeteit használtuk annak kifejezésére, hogy a különböző állapotokban az áramkörnek más és más módon kell viselkednie. Ezekben a leírásokban a `case` szerkezetek `others/default` ága semmilyen utasítást nem tartalmazott, mert a modellezés során azt feltételeztük, hogy az állapotregiszter sosem vehet fel olyan értéket, ami nem egy érvényes állapotkód. Nagymegbízhatóságú rendszerekben ez a luxus nem adatik meg, ilyen alkalmazásokban az áramkört fel kell készíteni ezekre a helyzetekre is (lásd 8-27. ábra).

A 8-27. ábrán látható kódrészletekkel modellezett állapotgép egy hibajelet (`fsm_error`) állít elő, ha az állapotregiszter érvénytelen értéket vesz fel, majd belép egy várakozó állapotba, amelyben az egyetlen bemenet, amire reagál, az az állapotgép hibájának észlelését visszaigazoló `recover_fsm` bemenet. A logikai szintézer szoftver az állapotgép modelljének elemzésekor azt látja, hogy one-hot kódolást definiáltunk, vagyis az állapotregiszter 7-bites kell hogy legyen. Ugyanakkor az is igaz, hogy a `case` szerkezet `others/default` ágára semelyik másik állapot nem mutat, ami azt jelenti, hogy a végrehajtás sosem éri el ezt az ágat, ezáltal az `error1` és `error2` állapotok is elérhetetlenek lesznek. A szintézer tehát az `others/default`, az `error1` és az `error2` ágakhoz tartozó összes értékadást figyelmen kívül hagyja, mert így az állapotregiszter 2 flip-flopját megspórolhatja *(register pruning)*. A megspórolt flip-flopokkal együtt azonban minden olyan logikát is eltávolít, ami a hibás állapot detektálásához és a hibajelző kimenet előállításához kell, aminek az lesz a következménye, hogy ha valamilyen fizikai behatás (pl. SEU) hatására az állapotgépünk mégiscsak érvénytelen állapotba kerül,

[73] Ősi magyar kifejezéssel „tie-off-oljuk".

VHDL

```
architecture rtl of safe_fsm is

  type state_t is ( s0,s1,s2,s3,s4,error1,error2 );
  signal state: state_t;
  attribute syn_encoding: string;
  attribute syn_encoding of state:signal is "one-hot";

begin

  process ( clk, rst )
  begin
    if ( rst = '1' ) then
      state <= s0; fsm_error <= '0';
      ...
    elsif ( rising_edge(clk) ) then
      case state is
        when s0 => ...
        when s1 => ...
        when s2 => ...
        when s3 => ...
        when s4 => ...
        when error1 => fsm_error <= '1';
                       state <= error2;
        when error2 => fsm_error <= '1';
                       if ( recover_fsm = '1' ) then
                         fsm_error <= '0';
                         state <= s0;
                       end if;
        when others => fsm_error <= '1';
                       state <= error1;
      end case;
    end if;
  end process;

end architecture rtl;
```

Verilog

```
module safe_fsm (...);

  typedef enum logic [2:0]
     { s0,s1,s2,s3,s4,error1,error2 } state_t;

  state_t state;
    /* synthesis syn_encoding = "one-hot" */

  always @ ( posedge clk, posedge rst )
  begin
    if ( rst == 1'b1 ) begin
      state <= s0; fsm_error <= 1'b0;
      ...
    end else begin
      case ( state )
        s0: ...
        s1: ...
        s2: ...
        s3: ...
        s4: ...
        error1:   fsm_error <= 1'b1;
                  state <= error2;
        error2:   fsm_error <= 1'b1;
                  if ( recover_sfm == 1'b1 )
                    state <= s0;
        default:  fsm_error <= 1'b1;
                  state <= error1;
      endcase;
    end
  end

endmodule
```

8-27. ábra. *Nagymegbízhatóságú állapotgép egy lehetséges megvalósítása*

akkor ott is marad, mert a következő állapotot meghatározó logika az érvénytelen állapotkód esetén egyik kimenetén sem jelez. Ráadásul az állapotgép interfészének megfigyelésével erről semmilyen módon nem értesülhetünk, csak azt tapasztaljuk, hogy az állapotgép bemeneteit akárhogy rángatjuk, a kimenetei változatlanok lesznek (kivéve persze, ha meghúzzuk a reset-et). Ezt a – nagymegbízhatóságú rendszerekben feltétlenül megakadályozandó – szituációt holtpontnak *(FSM dead-lock)* nevezzük. A helyzet ellentmondásossága abban áll, hogy a HDL modellünk teljesen helyesen írja le az elvárt viselkedést, a problémát az okozza, hogy a szintézis szoftver nagy igyekezetében olyan dolgokat is eltávolított a kisebb erőforrásigény elérése érdekében, amit nem kellett volna. Ilyen helyzetekben van komoly szerepe azoknak az attribútumoknak és pragmáknak, amelyekkel megakadályozhatjuk a szintézis szoftvert ebben a logikai optimalizációs mészárlásban (lásd 8-28. ábra).[74]

A regiszterek száma nemcsak csökkenhet, hanem növekedhet is a szintézis során. Ha egy flip-flop által meghajtott kapubemenetek száma, vagyis a flip-flop terhelése, fan-outja kiugróan nagy, akkor a logikai szintézis szoftver megduplázhatja az adott flip-flopot *(register*

[74] A 8-28. ábrán látható `syn_preserve` attribútum/pragma csak az állapotregiszter flip-flopjainak az eltávolítását akadályozza meg. Ahhoz, hogy az állapotdekóderek az érvénytelen állapotokat is biztosan képesek legyenek felismerni, szintézis szoftvertől függően további beállításokra és attribútumokra/pragmákra lehet szükség (a Synopsys Synplify szintézis szoftvere esetén, Microsemi ProASIC3E technológiára szintetizálva pl. a `syn_safe_case` kapcsoló az, amelyik ezt garantálja, lásd 9.1.4.3. pont).

VHDL

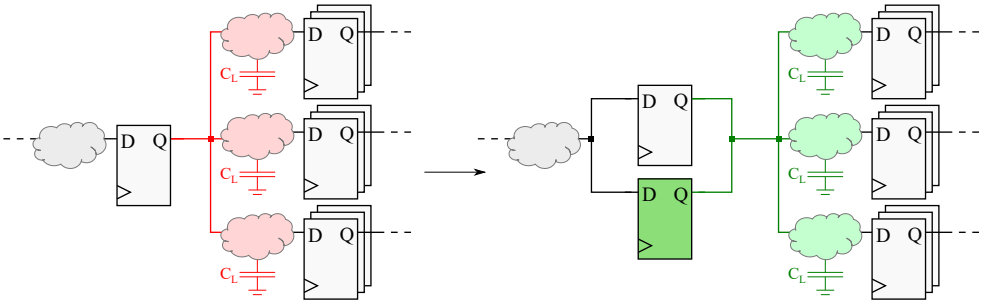
```
type state_t is ( s0, s1, s2, s3, s4, error1, error2 );
signal state: state_t;
attribute syn_encoding: string;
attribute syn_encoding of state:signal is "one-hot";
attribute syn_preserve: string;
attribute syn_preserve of state:signal is true;
```

Verilog

```
typedef enum logic [2:0] { s0, s1, s2, s3, s4, error1, error2 }  state_t;
state_t state; /* synthesis syn_encoding = "one-hot", syn_preserve = 1 */
```

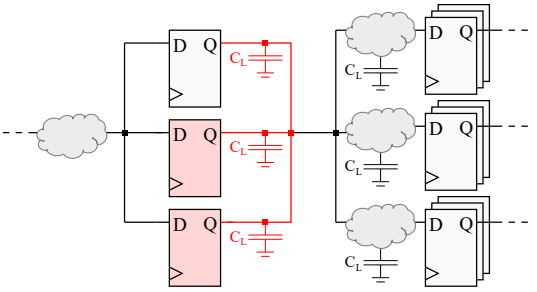
8-28. ábra. *A regiszterek eltávolításának megakadályozása a* ***syn_preserve*** *attribútum/pragma segítségével*

duplication), hogy az eredő meghajtóképesség nagyobb, a meghajtott lokális adatutak várható késleltetése pedig kisebb legyen (lásd 8-29. ábra).



8-29. ábra. *Nagy terhelésű regiszterek duplikálása*

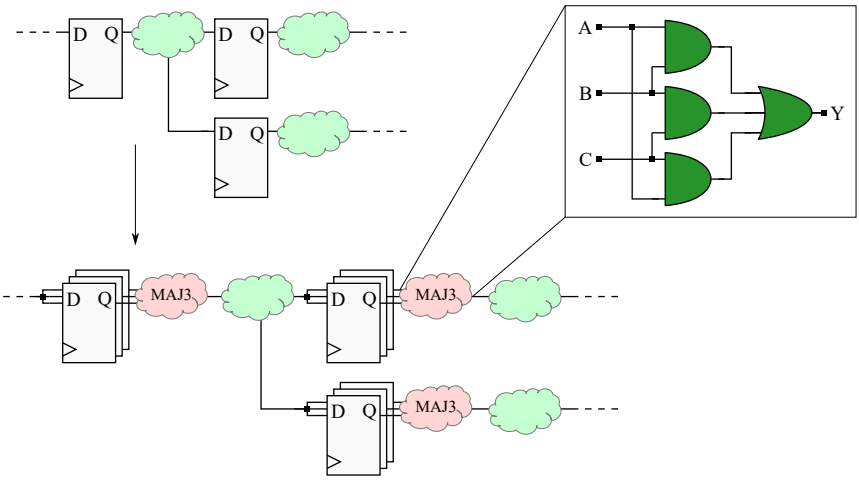
Mivel a standard cellakönyvtárakban a flip-flopoknak egy sor, különböző meghajtóképességű verziója érhető el, ez a technika inkább a PLD technológiákra jellemző. A flip-flopok párhuzamos kapcsolása persze többlet huzalozást igényel, ami egyúttal többlet terhelést és késleltetést is jelent, ezért kettőnél több regisztert a PLD szintézis szoftverek sem szoktak beépíteni (innen az elnevezés, lásd 8-30. ábra).



8-30. ábra. *A flip-flopok párhuzamos kapcsolása többlet terhelést jelent*

Egy másik logikai szintű, a flip-flopok számát igen erőteljesen befolyásoló optimalizációs technika az ún. *„soft register-TMR" (Triple Modular Redundancy)*, ami azt jelenti, hogy a korábban is említett nagyenergiájú részecskebecsapódások okozta átbillenésekből (SEU)

adódó funkcionális hibák elkerülése céljából a logikai szintézis szoftver az RTL modellünk minden egyes flip-flopját valójában három flip-flopból és egy MAJ3 típusú kombinációs hálózatból, egy ún. döntő-logikából építi fel a 8-31. ábrán látható módon.[75]



8-31. ábra. *Soft register-TMR*

A SEU-átbillenések a fenti struktúrában nem okozhatnak hibát; normális körülmények között ui. az egyazon RTL modellbeli flip-flopot megvalósító mindhárom fizikai flip-flop azonos értékű, ami egyúttal a struktúra kimenetét is jelenti. Ha az egyik flip-flop át is billen, a struktúra kimenetén ez a változás nem jelenik meg, mert ott a MAJ3 logikán keresztül a két érintetlen flip-flop értéke érvényesül. A struktúra kimenete csak akkor válthat értéket, ha a flip-flopok közül legalább kettő megváltozik, ami egyetlen becsapódó részecske hatására – legalábbis az „egybites hiba"-modellünk alapján[76] – nem következhet be.

Konstansok terjesztése *(constant propagation)* • A logikai algebra konstansokkal kapcsolatos alaptételei (lásd 8-1. táblázat) miatt a kódban előforduló konstansok rengeteg logikai optimalizációra adnak lehetőséget.

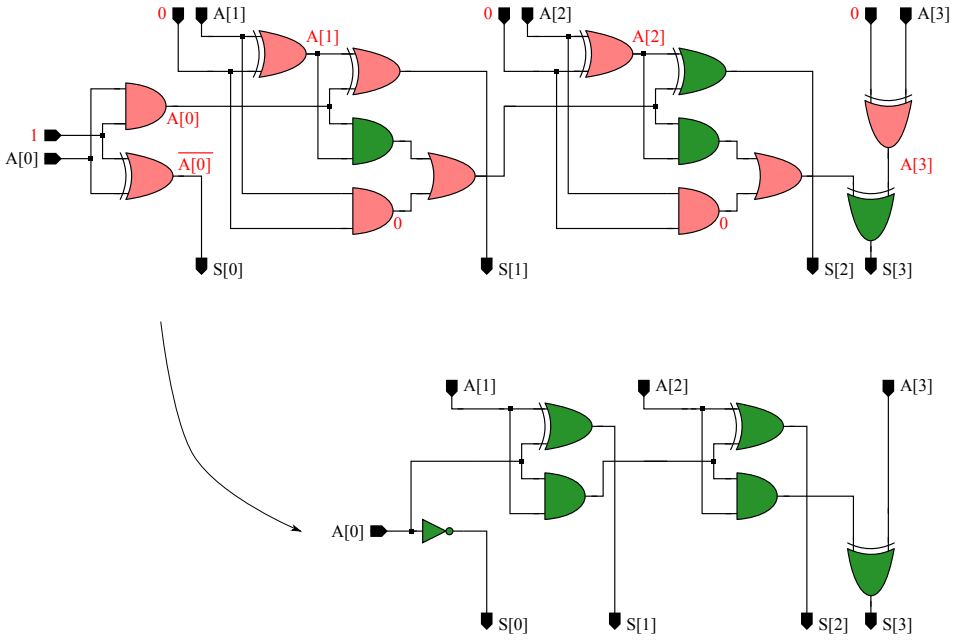
X AND 1 = X
X AND 0 = 0
X OR 1 = 1
X OR 0 = X

8-1. táblázat. *Konstansok szerepe a logikai algebrában*

[75] A „soft" jelző arra utal, hogy ebben a megoldásban nem maga az eszköz biztosítja a TMR megvalósítást, hanem azt „szoftveresen", a szintézis során kell megvalósítanunk. Egyes cellakönyvtárakban (pl. Microsemi RTAX FPGA család) a flip-flopok áramköri szinten valósítják meg a TMR-t, vagyis a szintézis során ezzel nem kell foglalkoznunk.

[76] Ez a megoldás hallgatólagosan azt feltételezi, hogy egyetlen nagyenergiájú részecske legfeljebb egy flip-flopot képes átbillenteni, ami persze csak egy korlátozott érvényességű hibamodell, de az esetek túlnyomó részében mindenképpen igaznak tekinthető. Az áramköri szinten megvalósított TMR annyiban többet nyújt, hogy ott a layout tervezése során – az adott részecskebecsapódási jelenség hatáskeresztmetszetének ismeretében –, az egyes flip-flopok egymáshoz viszonyított fizikai elhelyezésével akadályozzák meg, hogy egyetlen nagyenergiájú részecske a flip-flop-hármas több tagját is átbillentse.

A 8-32. ábra egy négybites összeadó példáján keresztül mutatja be ezt az egyszerűsítési lehetőséget.



8-32. ábra. *Konstansok hatása a szintetizált kombinációs hálózatokra*

Kombinációs hálózatok optimalizációja • A kombinációs hálózatok optimalizációja során arra törekszünk, hogy egy adott logikai összefüggést minél egyszerűbben fejezzünk ki. Egyszerűség alatt itt azt értjük, hogy a kimenet értékét meghatározó logikai kifejezésben minél kevesebb változó és minél kevesebb logikai művelet szerepeljen. Ennek eléréséhez célszerű először az egyszerűsítendő logikai kifejezés diszjunktív vagy konjunktív kanonikus alakját felírni *(flattening)* majd ezen elvégezni egy kétszintű logikai minimalizálást. A kétszintű logikai minimalizálás megvalósítására több algoritmust is kidolgoztak, amelyek funkcionálisan megegyeznek, számítási igényük azonban jelentősen eltérhet (pl. Quine-McCluskey, ESPRESSO, Scherzo). Az alábbiakban a kétszintű logikai minimalizálás folyamatát a grafikus minimalizálás módszerén keresztül mutatjuk be. A 8-33. ábra a kiindulási HDL modellt és az abból optimalizáció nélkül szintetizált áramköri szerkezetet mutatja be.

A kimeneti regisztert meghajtó kombinációs hálózatot a 8-33. ábrán a HDL modellből közvetlenül, optimalizáció nélkül rajzoltuk fel. Ez a kombinációs hálózat funkcionálisan teljesen megfelel a HDL modell által definiált specifikációnak, áramköri szempontból azonban erősen redundáns, ami a 8-34. ábrán látható Veitch-tábla alapján könnyen belátható.

VHDL

```
process ( clk, rst )
  variable s1, s2, s3, s4: std_logic;
begin
  if ( rst = '1' ) then y <= '0';
  elsif ( rising_edge(clk) ) then

    if ( a = '1' and b = '1' ) then s1 := '1';
    else s1 := '0'; end if;

    s2 := not a and b and c and not d;

    if ( a = '1' and b = '0' and d = '1' ) then
      s3 := '1'; else s3 := '0'; end if;

    s4 := not a and b and not c and not d;

    y <= s1 or s2 or s3 or s4;

  end if;
end process;
```

Verilog

```
logic s1, s2, s3, s4;
always @ ( posedge clk, posedge rst )
begin
  if ( rst == 1'b1 ) y <= 1'b0;
  else begin

    if ( a == 1'b1 && b == 1'b1 ) s1 = 1'b1;
    else s1 = 1'b0;

    s2 = ~a & b & c & ~d;

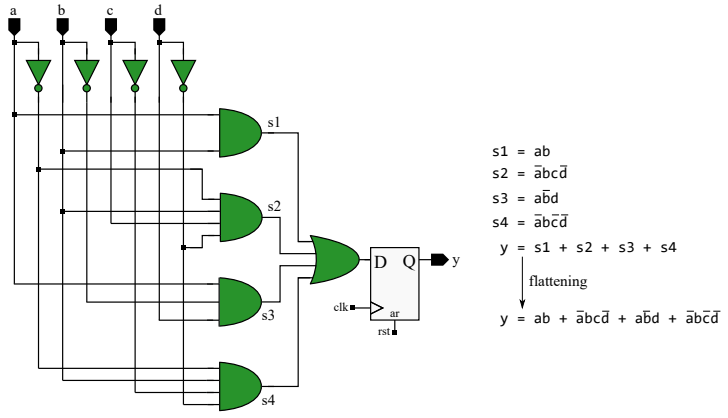
    if ( a == 1'b1 && b == 1'b0 && d == 1'b1 )
      s3 = 1'b1; else s3 = 1'b0;

    s4 = ~a & b & ~c & ~d;

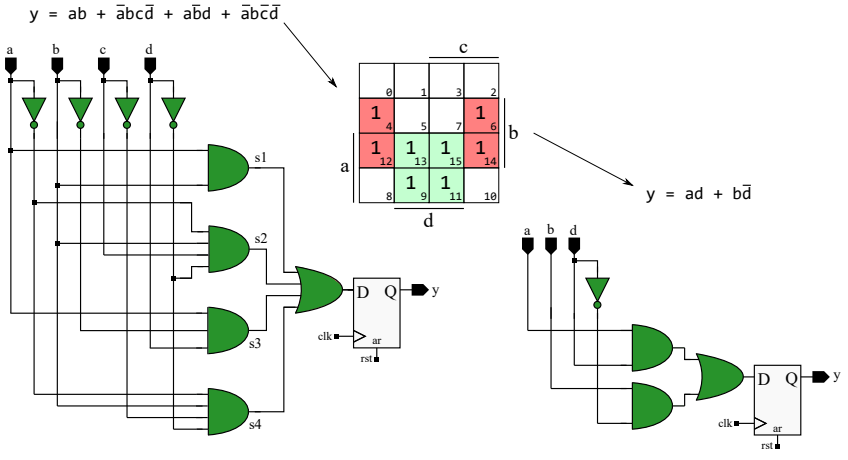
    y <= s1 | s2 | s3 | s4;

  end
end
```

Áramköri struktúra

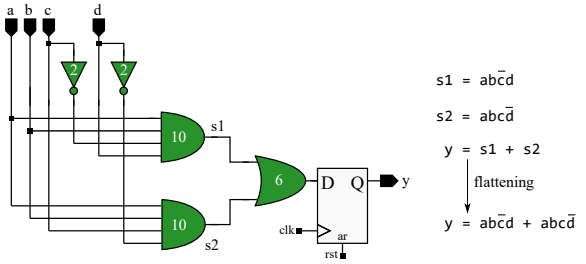


8-33. ábra. *Szintézis logikai optimalizáció nélkül*



8-34. ábra. *Kétszintű logikai minimalizálás*

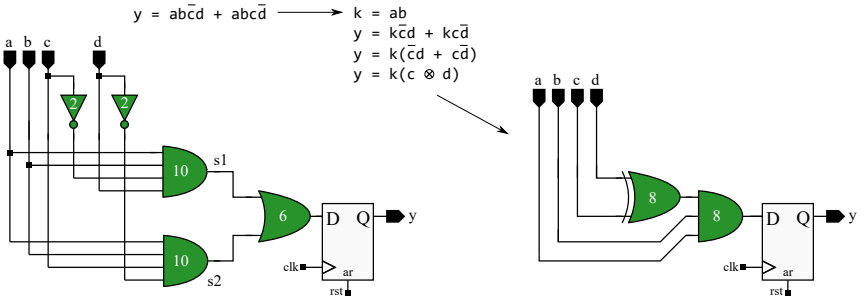
A kétszintű minimalizálással tehát jelentősen csökkenthetjük leendő áramkörünk erőforrásigényét – így méretét és fogyasztását –, ez azonban nem jelenti azt, hogy a kétszintű kanonikus alak minimalizált megvalósítása mindig a lehető legjobb megoldást jelenti. Az alábbiakban bemutatjuk, hogy a kiemelés (dekompozíció, *factorization*) nevű egyszerűsítési technika miként eredményezhet a kétszintű kanonikus alakból létrehozottnál kedvezőbb áramköri megvalósítást. A 8-35. ábra a kiindulási HDL modellt és az abból a kimeneti regisztert meghajtó kombinációs hálózat diszjunktív kanonikus alakjának közvetlen megvalósításával létrehozott áramköri szerkezetet mutatja be.



8-35. ábra. *Diszjunktív kanonikus alak közvetlen megvalósítása*

Ez a diszjunktív kanonikus alak tovább már nem egyszerűsíthető. CMOS kapcsolástechnikát feltételezve a kimeneti regisztert meghajtó kombinációs hálózat áramköri megvalósításához 30 db tranzisztorra van szükség (az egyes logikai kapuk alkatrészigényét a kapuk szimbólumaiban tüntettük fel). A 8-36. ábra azt mutatja meg, hogy megfelelő kiemeléssel hogyan egyszerűsíthető a kombinációs hálózat úgy, hogy CMOS megvalósításának tranzisztorszámban kifejezett erőforrásigénye csaknem felére csökkenjen.

A példák alapján jól látható, hogy a HDL modellből közvetlenül adódó logikai kifejezések átalakításával az áramköri megvalósítás tulajdonságai jelentősen javíthatók. Ugyanakkor a logikai optimalizációs eljárások nem függetleníthetők teljesen a megvalósítás alapjául szolgáló technológiától. Tegyük fel, hogy a 8-36. ábrán látható példaáramkört standard cellás technológián szintetizáljuk. Nyilvánvaló, hogy a kiemeléssel végrehajtott logikai optimalizáció kedvező hatással lesz az áramkör méretére, hiszen kevesebb tranzisztort kell



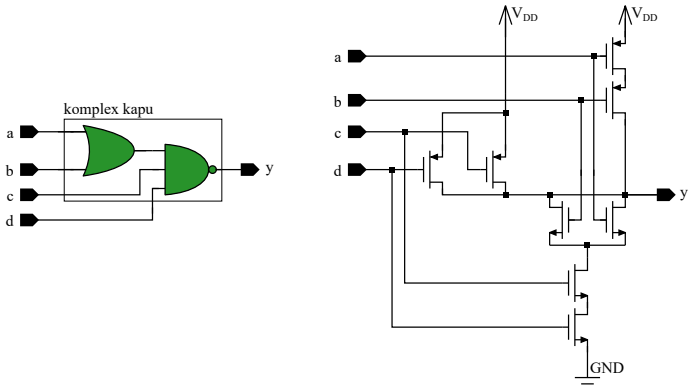
8-36. ábra. *Egyszerűsítés kiemeléssel*

felhasználnunk, sőt, a kevesebb kapubemenetnek köszönhetően a bemeneteket meghajtó fokozatok terhelése is csökken, ami az áramkör lokális adatútjainak késleltetését is csökkenti. A helyzet azonban egészen más, ha például egy olyan FPGA-ra végezzük a szintézist, amely négybemenetű LUT-okat bocsát a rendelkezésünkre kombinációs hálózatok megvalósításához. Ebben az esetben a 8-36. ábrán látható optimalizációs lépés teljesen felesleges, mert végső soron mindenképpen egy LUT-ra lesz szükségünk a fizikai megvalósításhoz, akár elvégezzük az optimalizációt, akár nem. Modellünk további finomításához tehát tovább kell lépnünk a technológiai leképezés *(technology mapping)* műveletére.

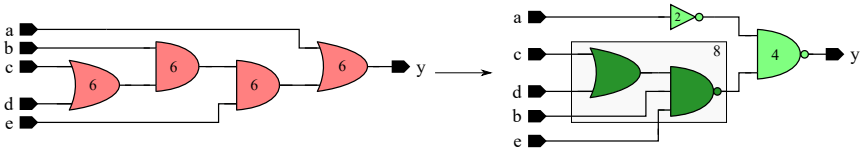
8.2.1.3 Technológiai leképezés és technológiafüggő logikai optimalizáció

A technológiai leképezés *(technology mapping)* azt jelenti, hogy a kapuszintű, de egyelőre absztrakt logikai kapukat és flip-flopokat tartalmazó kapcsolási sémánk egyes elemeihez valamely konkrét cellakönyvtár egy-egy elemét rendeljük hozzá. Ebben a lépésben lesz például egy általános NAND kapuból egy Cyclone III 4-LUT, vagy egy általános D flip-flopból a DFC1 cella az AMS 350 nm-es cellakönyvtárából.

Komplex kapuk • Standard cellás esetben a cellakönyvtárak ún. komplex kapukat is tartalmaznak, amelyek a legalapvetőbb AND, OR, XOR kapuknál némileg összetettebb logikai funkciót valósítanak meg. Az ilyen összetevőket a generikus logikai szintézis még nem tudja kihasználni, de a technológiai leképezés után kapott netlista tulajdonságain már ezeknek a celláknak a segítségével is javíthatunk. A 8-37. ábra egy ilyen komplex kapu



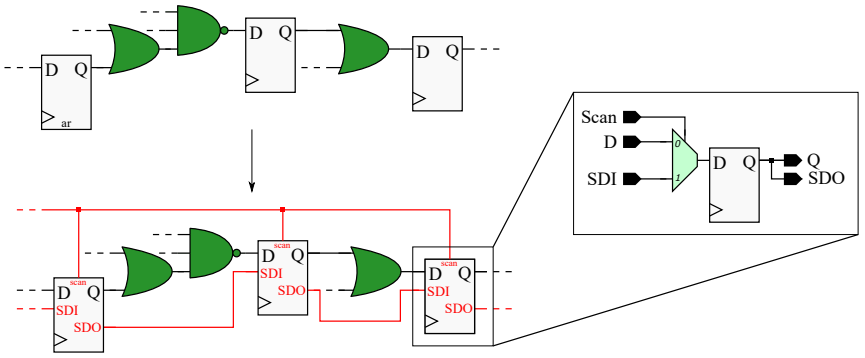
8-37. ábra. *Egy komplex kapu logikai és áramköri szintű kapcsolási sémája*



8-38. ábra. *Kombinációs hálózat méretének csökkentése komplex kapu felhasználásával*

logikai és áramköri szintű kapcsolási sémáját mutatja be, a 8-38. ábrán pedig azt láthatjuk, hogy ennek a komplex kapunak a felhasználásával miként csökkenthetjük egy kombinációs hálózat megvalósításához szükséges tranzisztorok számát.

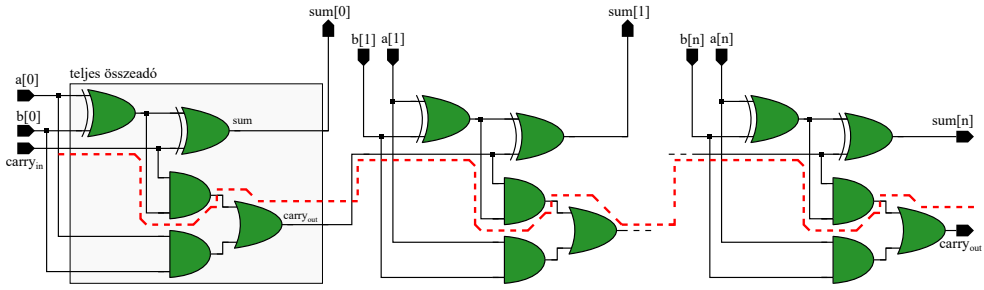
Scan-path beillesztés • Bár a korábban tárgyaltakkal ellentétben nem az áramkörünk erőforrás-felhasználásának csökkentését célozza, ettől függetlenül a technológiafüggő optimalizálási módszerek közé soroljuk az ún. „scan-path beillesztés" *(scan-path insertion)* fogalmát. Ebben a lépésben az áramkörünk flip-flopjait, vagy azoknak legalább egy részét olyan flip-flopokra cseréljük le, amelyek egy speciális vezérlő jel segítségével ún. *„scan-path"* üzemmódba kapcsolhatók át (lásd 8-39. ábra).



8-39. ábra. *Scan-path üzemmódba kapcsolható flip-flopok beillesztése*

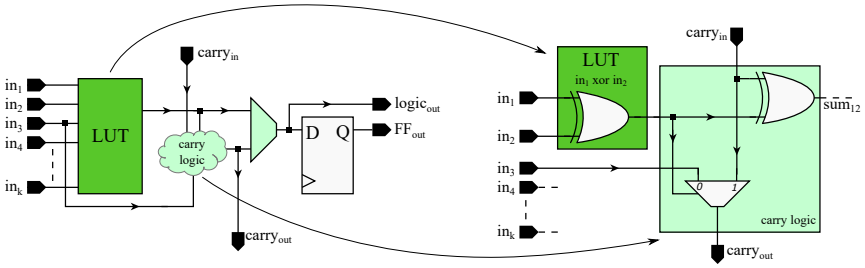
Scan-path üzemmódban az áramkörünk nem az eredeti funkciója szerint működik, hanem flip-flopjai egy léptetőregisztert képeznek, amelynek értékét egy soros bemeneten betölthetjük, egy soros kimeneten pedig ki is olvashatjuk. Ez a megoldás egy ún. DFT *(Design For Testability)* módszer, amelynek célja az áramkörünk belső állapotának beállíthatósági és megfigyelhetőségi tulajdonságainak javítása. Az áramkör minden egyes flip-flopját scan-path-kompatibilissé tenni meglehetősen nagy területigény-növekedéssel járna, ezért rendszerint csak olyan állapotgépek állapotregisztereinél alkalmazzuk ezt a technikát, amelyeknek bizonyos állapotai a külső interfészek használatával csak nehezen érhetők el, így azok tesztelése igen időigényes lenne. Ilyen esetekben az áramkört scan-path módba kapcsolva a soros bemeneten keresztül beléptethetjük az ellenőrzendő állapot bináris kódját, majd a scan-path módot kikapcsolva az áramkör ebből az állapotból fogja folytatni a működést, amit így könnyen megfigyelhetünk. A normál működés közbeni állapotváltozásokat úgy figyelhetjük meg, hogy a teszt egy pontján ismét aktiváljuk a scan-path módot, és a soros kimeneten keresztül az állapotregiszterek aktuális értékét kiléptetjük.

Carry-logika az FPGA-kban • A digitális aritmetikai áramkörökben az elemi teljes összeadóknak fontos szerepük van, mert rengeteg aritmetikai művelet erre az egyszerű struktúrára vezethető vissza. A 8-40. ábra a teljes összeadó és az abból felépített több bites, ún. *„ripple-carry"* összeadó kapcsolási sémáját mutatja be.



8-40. ábra. *Egy- és több bites (ripple-carry) teljes összeadó*

A több bites összeadó létrehozása esetén a *carry* kimenet előállításához egy olyan adatút alakul ki, amely az összeadó minden fokozatában két logikai kapun halad keresztül, ami az összes többi adatútnál sokkal nagyobb késleltetésűvé teszi. Ez az adatút lesz tehát az összeadó áramkör kritikus útja, ami meghatározza, hogy mekkora órajel-frekvencián lesz képes működni a teljes áramkör. Az FPGA áramkörök logikai alapelemei (*Basic Logic Element*, BLE) tartalmaznak olyan részáramköröket, amelyek kifejezetten ennek az adatútnak az optimalizálását szolgálják. A 8-41. ábra azt mutatja be, hogy egy FPGA esetén hogyan véb gezhetjük el a technológiai leképezést e plusz erőforrások kihasználásával.



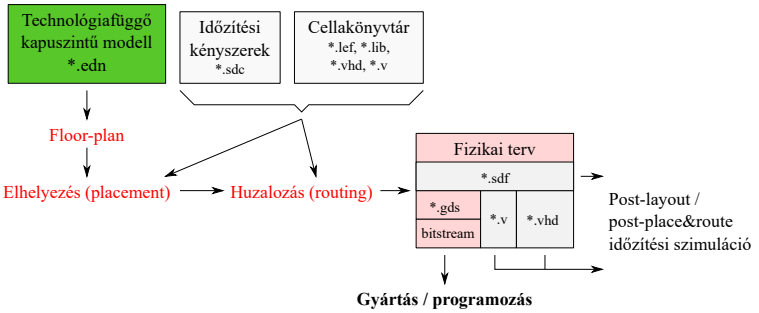
8-41. ábra. *Összeadó leképezése FPGA erőforrásokra*

A BLE-ket kiegészítő carry-logika tehát lehetővé teszi azt, hogy minden egyes BLE önmagában egy egybites teljes összeadót valósítson meg, egy n-bites összeadóhoz tehát elegendő n db BLE. Ez a megoldás lényegesen erőforrás-kímélőbb, mintha az összegbiteket előállító, valamint a carry-generáló logikai összefüggéseket külön-külön LUT-okból raktuk volna össze, ráadásul a carry-logika áramköri szintű és layoutszintű optimalizálásával annak késleltetése is lényegesen csökkenthető.

8.2.2 Fizikai szintézis

A technológiai leképezés után rendelkezésünkre áll egy kapuszintű netlista, amely már a kiválasztott technológia alapelemeiből áll. Ez a modell azonban még nem tartalmaz információt az alkatrészek elhelyezkedéséről és a közöttük lévő összeköttetésekről sem,

mindez a fizikai szintézis során alakul ki. A fizikai szintézis bemenete a logikai szintézis által generált EDIF netlista, az időzítési kényszereket leíró SDC-fájl és a cellakönyvtárt leíró adatbázis. Ez utóbbiból a logikai szintézis csak a cellák időzítési és fogyasztási adatait tartalmazó Liberty-fájlokat használta, a fizikai szintézishez azonban már szükség van azok geometriai jellemzőire is, valamint a gyártástechnológia egyéb paramétereire, mint a különböző fémezési rétegeken megvalósítható vezetékek szélessége és a huzalozási rétegek közötti átvezetések *(via)* geometriai tulajdonságai. Ezeket az információkat a cellakönyvtár részeként rendelkezésre álló, ún. LEF *(Library Exchange Format)* fájlok tartalmazzák. A 8-42. ábra fizikai szintézis folyamatát mutatja be.



8-42. ábra. *A fizikai szintézis folyamata*

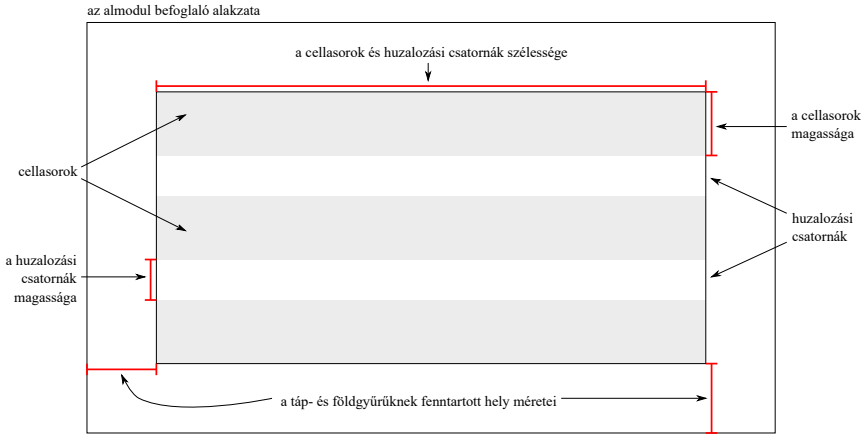
A fizikai szintézis eredménye standard cellás technológia esetén az ún. GDSII adatbázis-fájl, ami a chip teljes layout-rajzolatát strukturált formában leírja, tehát tartalmazza mindazt az információt, ami a gyártáshoz szükséges. FPGA technológia esetén a fizikai szintézis az eszköz felkonfigurálásához szükséges, gyártófüggő programozófájl *(bitstream)* előállításával ér véget.

A fizikai szintézis folyamata standard cellás és FPGA technológia esetén meglehetősen különböző, ezért a folyamatnak e két változatát külön-külön tárgyaljuk.

8.2.2.1 Fizikai szintézis standard cellás technológián

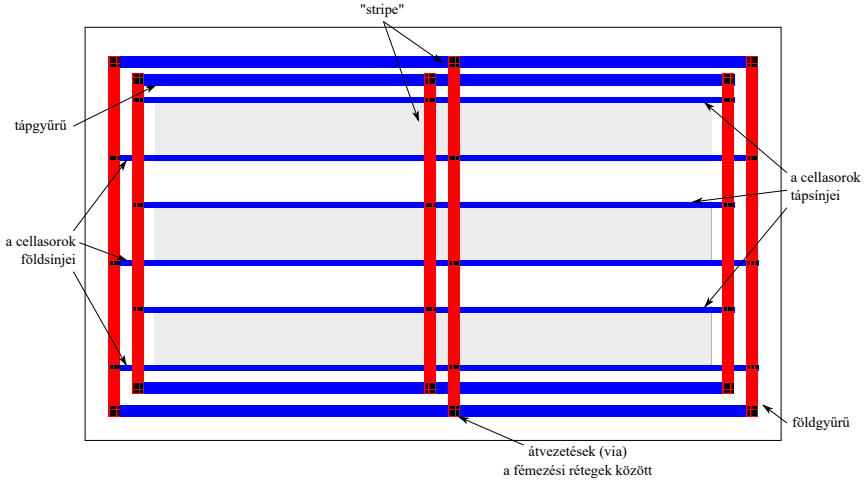
Az alábbiakban a standard cellás áramkörök fizikai szintézisének lépéseit a 7.1. alfejezetben makrocellának nevezett, belső strukturális hierachiát nem tartalmazó részegység példáján keresztül mutatjuk be.

Floor-plan • Egyszerű makrocellák esetén, amikor az áramkör összetettsége nem igényel hierarchikus fizikai tervet *(flat design)*, akkor a floor-plan alatt a makrocella befoglaló alakzatának geometriai méreteire, valamint a cellasorok és huzalozási csatornák méreteire vonatkozó információt értjük. A 8-43. ábra a példaáramkör floor-plan-jét mutatja be.



8-43. ábra *– A példaáramkör floor-plan-je*

Tápellátás • A floor-plan elkészítése után az almodul tápellátásának megtervezése következik. A táp- és földhozzávezetés ún. táp- és földgyűrűkön keresztül valósul meg, amelyek körülveszik a cellaterületet. A tápvonalak impedanciájának – ezáltal az IR-dropnak –, valamint áramsűrűségének csökkentése céljából a gyűrűkön kívül további tápvonalak *(stripe)* beépítésére is szükség lehet. A cellasorok tápellátásáról az azok felső és alsó részén végigfutó tápsínek gondoskodnak (8-44. ábra).

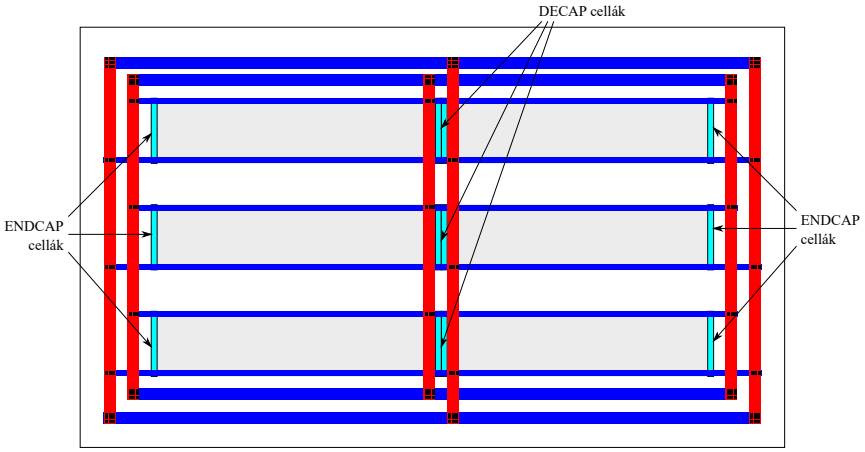


8-44. ábra. *Táp- és földgyűrűk, cellasorok táp- és földsínjei*

Elhelyezés *(placement)* • A tápellátás megtervezése után a cellák elhelyezése következik. Többféle cella elhelyezéséről kell gondoskodnunk, ennek megfelelően ez a lépés részfolyamatokra bontható:

- Fix pozíciójú cellák elhelyezése
- Standard cellák elhelyezése
- Kitöltő (FILLER) és tartalék (SPARE) cellák elhelyezése

Az ENDCAP és DECAP-cellák fix pozíción helyezkednek el. Az ENDCAP cellák mindig a cellasorok két végén, a DECAP cellák pedig általában az almodul mérete által meghatározott fix távolságonként, a cellasorok mentén. A DECAP cellák használata – funkciójukból adódóan – csak összetett, nagyméretű áramkörök esetén indokolt, és helyzetük kevésbé helyhez kötött, mint az ENDCAP celláké (8-45. ábra).



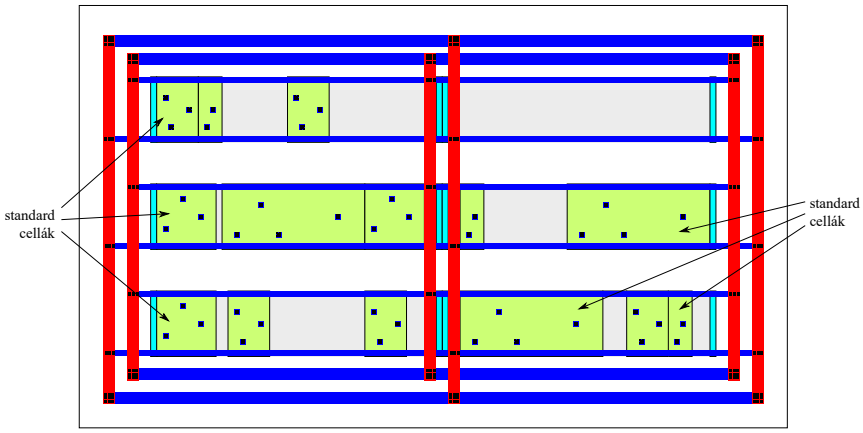
8-45. ábra. *Fix pozíciójú cellák elhelyezése*

A fix pozíciójú speciális cellák elhelyezése után kerül sor a logikai funkciót megvalósító standard cellák elhelyezésére (8-46. ábra).

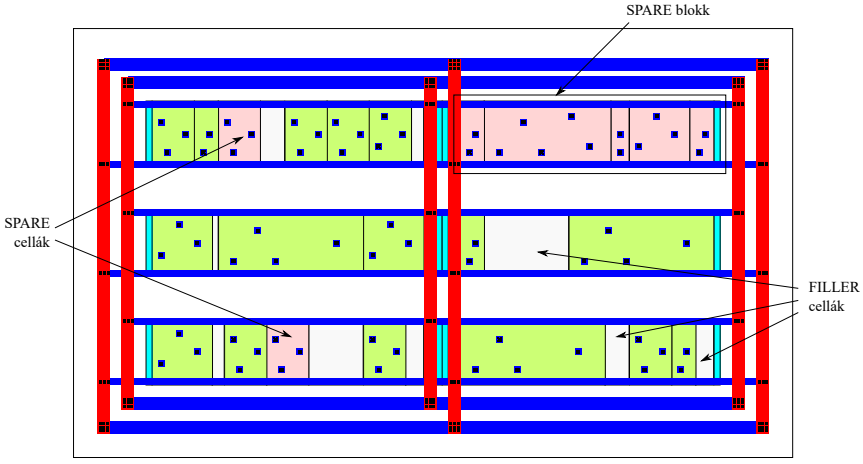
A cellasorokban a standard cellák között általában marad némi kihasználatlan hely. Ezeket a helyeket SPARE és FILLER cellákkal töltjük fel (8-47. ábra).

Huzalozás *(routing)* • Ha minden belső cella a helyére került, következhet azok összeköttetéseinek megvalósítása, a huzalozás (8-48. ábra). A cellasorokon belül érdemes a huzalozást úgy megoldani, hogy a legalsó fémezési réteget ne használjuk fel cellák közötti összeköttetések megvalósítására, mivel arra a cellák belső vezetékezéséhez van szükség. A cellasorokon kívül viszont arra kell törekedni, hogy csak akkor „jöjjünk fel magasabb huzalozási rétegre", ha az valóban indokolt. A magasabb huzalozási rétegek nagyobb méretei miatt ezek a vezetékek nagyobb terhelő kapacitást jelentenek, ami a jelutak késleltetését növeli.

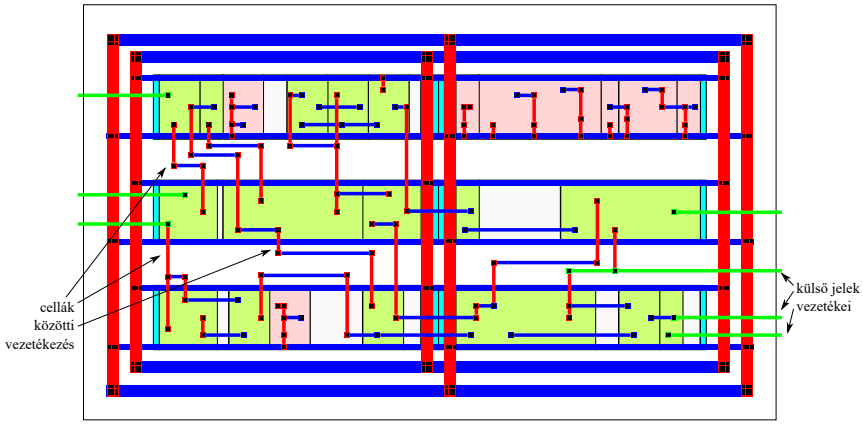
A belső szerkezeti hierarchiát nem tartalmazó makrocellák eléggé reguláris felépítésűek, aminek a hátterében az áll, hogy a fizikai tervezés részműveleteit gépi úton, automatizáltan igyekszünk elvégezni. A cellasorok és huzalozási csatornák rendszere, a csak vízszintesen és függőlegesen futó vezetékek, a fix pozíciójú táp- és földsínek mind azt a célt szolgálják, hogy a fizikai szintézis könnyen algoritmizálható legyen. Egy, a 8-48. ábrán láthatóhoz hasonló fizikai terv előállítása ugyan nem „gombnyomásra" történik, de az egyes



8-46. ábra. *Standard cellák elhelyezése*



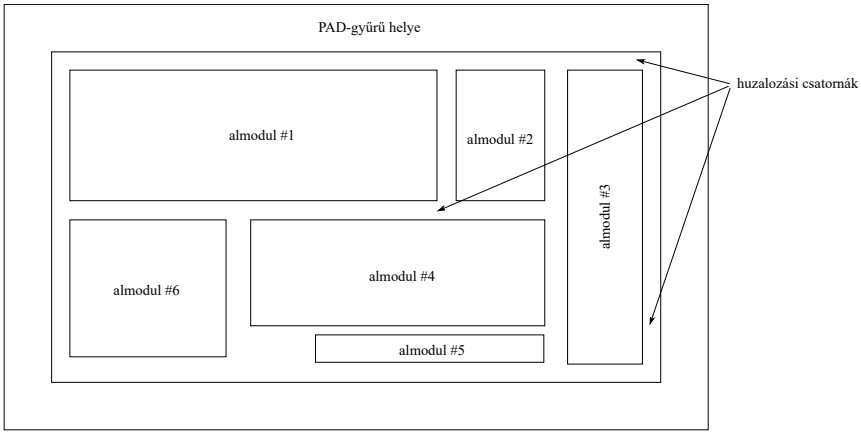
8-47. ábra. *SPARE és FILLER cellák elhelyezése*



8-48. ábra. *A cellák közötti vezetékezés megvalósítása*

részfolyamatokban igen nagy szerepük van a tervezőrendszer vezérlését automatizáló parancsfájloknak, scripteknek.

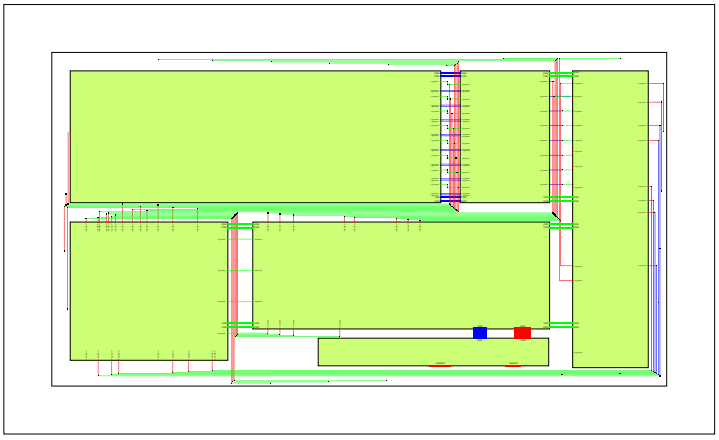
Az automatizáltságnak kisebb szerep jut, ha nem egy reguláris szerkezetű makrocella, hanem a teljes chip fizikai tervének megtervezéséről van szó. A chip almoduljainak méreteit, orientációját és egymáshoz viszonyított helyzetét a chip-szintű, globális floor-plan definiálja. Egy összetett, sok részegységből álló áramkör esetén a globális floor-plan tervezése során a fő cél az almodulok közötti vezetékezés mennyiségének minimalizálása, valamint a táp- és földhálózat terhelésének és a disszipációsűrűségnek a kiegyensúlyozása a felület mentén. A 8-49. ábra a 7-5. ábrán bemutatott, több makrocellából álló chip globális floor-plan-jét szemlélteti.



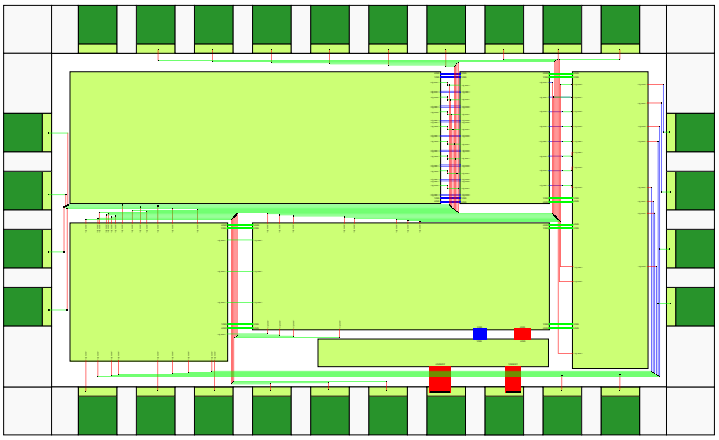
8-49. ábra. *Egy több makrocellából álló chip globális floor-plan-je*

A globális floor-plan megtervezése során a makrocellák olyan reprezentációival dolgozunk, amelyek csak azok kivezetéseinek pozícióját írják le, valamint azt, hogy az adott makrocella mely területeken használja az egyes huzalozási rétegeket belső összeköttetéseinek megvalósításához. Ezeket a területeket a makrocellák közötti, globális huzalozáskor szabadon kell hagynunk. Ennél tervezési szempontból egyszerűbb, de a végső huzalozás összetettsége szempontjából kedvezőtlenebb megoldást eredményező módszer, ha rögzítjük, hogy az alsó n fémezési réteget a makrocellák belső vezetékezésének, az afelettieket pedig a globális huzalozás megvalósítására használjuk fel. A 8-50. ábra a teljes chip fizikai tervét mutatja a makrocellák szimbólumainak elhelyezése és a közöttük lévő vezetékezés kialakítása után.

PAD-gyűrű • A huzalozás elkészítése után a kivezetések PAD áramköreit tartalmazó PAD cellák elhelyezése és bekötése következik (8-51. ábra).



8-50. ábra. *Globális huzalozás kialakítása*



8-51. ábra. *PAD-gyűrű elhelyezése és bekötése*

8.2.2.2 Fizikai szintézis FPGA technológián

A „fizikai szintézis” kifejezés alatt FPGA technológia esetén valójában teljesen más dolgokat értünk, mint standard cellás esetben. A chip fizikai terve valójában már készen van, sőt, az eszközt le is gyártották, ilyen értelemben tehát nincs értelme fizikai szintézisről beszélni. Ugyanakkor a logikai szintézis során előállított, technológiafüggő kapuszintű modell előállításával a tervezési folyamat még nyilván nem ért véget. Még hátra van az, hogy a kapuszintű modell egyes elemeihez – amelyekről egyelőre csak annyit tudunk, hogy melyik FPGA eszközcsaládnak pontosan milyen típusú erőforrása fogja megvalósítani – hozzárendeljük a konkrét céleszköz egy-egy erőforrását (pl. egy konkrét LUT-ot vagy flip-flopot), meghatározzuk annak konfigurálható paramétereit (elhelyezés, *placement*), majd a kapcsolómátrixok konfigurációs bitjeinek értékét is meghatározzuk úgy, hogy az

így előálló, egyedi vezetékhálózat pontosan azt a funkciót valósítsa meg, amelyet a kapuszintű netlista definiál (huzalozás, *routing*). FPGA technológián tehát a fizikai tervezés nem a chip szerkezetének megtervezését jelenti, hanem annak meghatározását, hogy az általunk definiált funkciót megvalósító áramkör pontosan hogyan valósuljon meg az adott programozható eszközben.

A fizikai szintézis FPGA technológián – a standard cellás esettől eltérően – akár teljesen automatizáltan, gombnyomásra is történhet, mindenféle tervezői beavatkozás nélkül. Igaz ez az elhelyezésre és a huzalozásra egyaránt, de a tervezőrendszerek bizonyos fokú kézi beavatkozást azért itt is lehetővé tesznek, amit adott esetben célszerű kihasználni a jobb eredmény érdekében.

Elhelyezés • Az FPGA tervezőrendszerek lehetővé teszik azt, hogy az RTL modellünk logikai hierarchiája szerinti almodulokhoz az eszköz floor-plan-jén kijelöljünk egy-egy összefüggő régiót, amelyen belül az adott almodult meg akarjuk valósítani. A régióknak különböző típusait különböztetjük meg aszerint, hogy oda melyik logikai almodulhoz tartozó erőforrást helyezhetjük el:

- **Kizáró** *(exclusive)* régió: Egy adott logikai almodul megvalósításához az elhelyező algoritmus kizárólag a hozzárendelt régióból foglalhat le erőforrást, és más logikai almodul számára ebből a régióból nem foglalhat le semmilyen erőforrást.
- **Befoglaló** *(inclusive)* régió: Az ilyen típusú régióhoz hozzárendelt almodul megvalósításához szükséges erőforrások csak a régióba tartozó erőforrások közül kerülhetnek ki, de ugyanennek a régiónak a fennmaradó erőforrásait más logikai almodul megvalósítására is használhatja az elhelyező algoritmus.
- **Üres** *(empty)* régió: Kijelölhetünk az FPGA floor-plan-jén olyan területet is, amelynek erőforrásait egyáltalán nem szeretnénk felhasználni saját áramkörünk megvalósításához.

A tervezőrendszerek ennél sokkal finomabb kézi beavatkozást is lehetővé tesznek. Elvileg lehetséges a kapuszintű netlista minden egyes eleméhez kézzel hozzárendelni a megfelelő FPGA erőforrást, az áramköreink jellemző összetettsége miatt azonban ennek csak elvi jelentősége van. Az elhelyezés kézi meghatározásának lehetősége azonban adott, amit gyakran ki is használunk olyan módon, hogy az automatikusan előállított elhelyezésen utólag kisebb módosításokat végzünk. Az elhelyező és huzalozó algoritmusok igen kifinomultak és hatékonyak, de nem tévedhetetlenek. Ezek az algoritmusok ún. véletleneket alkalmazó, heurisztikus optimalizációs módszerek (pl. szimulált kihűtés, *simulated annealing*), amelyek nem egy optimális, hanem egy, a peremfeltételeknek eleget tevő, „kellően jó" elhelyezés/huzalozás megtalálására törekednek. Ha egymás után kétszer, két eltérő véletlen kezdőpontból *(random seed)* indítjuk el a futásukat,[77] akkor két teljesen eltérő megoldást kapunk, és elvileg nincs rá garancia, hogy ezek a megoldások valóban megfelelnek a peremfeltételeknek, vagyis az SDC fájlban definiált időzítési kényszereknek. Ez a gyakorlatban

[77] Kezdőpont alatt érthetjük például a kiinduláskénti definiált, véletlenszerű elhelyezést.

sokszor úgy jelenik meg, hogy az előállított áramkör adatútjainak túlnyomó része megfelel az időzítési feltételeknek, de akad néhány adatút, amelyek késleltetése pár nanoszekundummal több a kelleténél. Megtehetjük persze, hogy ilyen esetekben az elhelyező algoritmus kezdőpontját kissé módosítva újraindítjuk a teljes elhelyezést (ennek a módszernek az automatizált változatát *„multipass placement"*-nek nevezzük), ez azonban igen időigényes folyamat. Ehelyett célszerű inkább a kritikus adatutakat megvizsgálni, és elemeinek elhelyezését kézzel módosítani. Az elhelyezésbe való kézi beavatkozás tulajdonképpen azt jelenti, hogy konkrét előírásokat, kényszereket fogalmazunk meg arra vonatkozóan, hogy a kapuszintű netlista bizonyos elemeihez az eszköz melyik erőforrását szeretnénk hozzárendelni. Ezeket a kényszereket az SDC fájlhoz hasonló parancsfájlokban[78] rögzíthetjük annak érdekében, hogy ha mégis a teljes elhelyezést újra le kell futtatni (például a HDL modell módosítása miatt), a kritikus adatutak akkor is a már meghatározott erőforrásokból épüljenek fel.[79]

Huzalozás • Az elhelyezésnél elmondottak a huzalozásra is érvényesek; FPGA technológián az esetek túlnyomó részében a huzalozás teljesen automatizált módon történik, a tervező ebbe igen ritkán avatkozik be, de a lehetőség itt is adott. Az áramkörök összetettségéből adódóan nyilvánvalóan nem túl életszerű, hogy a tervező egy kapcsolómátrix konfigurációját átállítva maga határozza meg, hogy két logikai kapu között melyik vezetékszakasz teremtsen galvanikus kapcsolatot. A huzalozásba való „beavatkozás" alatt semmiképp sem ezt kell érteni, ezt valójában nem is teszik lehetővé a tervezőrendszerek. Az azonban könnyen előfordulhat, hogy egy órajel vagy reset jelet annak viszonylag kis terheléséből adódóan (például egy kisméretű órajeltartomány esetén) a fizikai szintézist végző algoritmusok nem dedikált órajelhálózatra, hanem egyszerű vezetékekre képeznek le. Ez aztán az időzítések szempontjából problémákat okozhat. Ilyenkor lehetőség van arra, hogy az elhelyezésnél is említett, fizikai jellegű kényszerként fogalmazzuk meg azt, hogy egy adott jel meghajtására a célarchitektúra melyik globális meghajtóbufferét használjuk, így „kikényszeríthető" egy adott globális hálózat felhasználása akkor is, ha a kérdéses jelnek a terhelése még nem érte el a szintézis algoritmusok „ingerküszöbét" ahhoz, hogy automatikusan is eszerint járjanak el. Felmerülhet a kérdés, hogy ha egy nem órajelként kezelt vezeték később időzítési hibához vezet, akkor a huzalozó algoritmus miért nem valósítja meg mégis órajelbufferek használatával az adott jelet. A helyzet a valóságban sajnos bonyolult abból a szempontból, hogy valójában melyik algoritmus határozza meg azt, hogy egy adott vezetéket milyen buffereken keresztül hajtunk meg. Ebbe akár a logikai szintézis algoritmus és az elhelyező algoritmus is beleszólhat, és ha ezek a korai, leginkább csekély információra támaszkodó döntések végül helytelennek bizonyulnak, akkor a huzalozó algoritmus önmagában már nem biztos, hogy képes megmenteni a helyzetet a szintézisnek ebben az előrehaladott állapotában. Éppen ezért az általános irányelv az, hogy ha a szoftverrendszerünk általános logikai szintézis szoftverből (pl. Synopsys Synplify) és technológia-specifikus fizikai szintézis szoftverből áll (pl. Microsemi Designer), akkor a megfelelő kényszerek alkalmazásával

[78] Ilyen parancsfájlok például Xilinx FPGA-knál az időzítési kényszereket is tartalmazó UCF *(User Constraint File)* fájlok, vagy Microsemi FPGA-knál a kifejezetten a fizikai szintézishez tartozó PDC *(Physical Design Constraint)* fájlok.

[79] Ha a HDL modell módosítása a kézzel beállított adatutakat is érinti, akkor persze ezek a parancsfájlok érvényüket veszítik.

a globális jelekre vonatkozó mindenfajta optimalizációt bízzunk inkább a technológia-specifikus algoritmusokra, mert ezek a célarchitektúrának olyan tulajdonságait is figyelembe képesek venni, ami a logikai szintézis szoftver számára esetleg nem ismert.[80]

8.2.3 Szintézis utáni ellenőrzések

A digitális áramkörök tervezési folyamatát egy sor ellenőrzés zárja le, amelyekkel célunk, hogy megbizonyosodjunk róla, hogy az áramkör terve gyártásra alkalmas, a legyártott áramkör pedig az elvárt funkciót fogja megvalósítani az előírt időzítési peremfeltételek betartása mellett.

Időzítési szimulációk • Az automatizált áramkörszintézist bemutató ábrákon látható, hogy a szintézis folyamata során több olyan, köztes modell is előáll, amelyek alkalmasak arra, hogy bizonyos időzítési tulajdonságokat is számításba vevő logikai szimulációnak vessük alá őket, meggyőződve arról, hogy a szintézis folyamata során végzett modelltranszformációk nem rontották el magát a megvalósított funkciót. Ennek érdekében a különböző szintézis szoftverek képesek HDL modellt előállítani a szintézis adott fázisában az áramkörről. Ezeket a kapuszintű modelleket a funkcionális verifikációhoz használt tesztkörnyezetbe illesztve logikai szimulációt végezhetünk. Ha tesztkörnyezetünk automatizált regressziós tesztelésre alkalmas, akkor elméletileg a tesztek újrafuttatása során minden esetben azt kell tapasztalnunk, hogy check-jeink nem jeleznek hibát, hiszen maga a logikai funkció nem változott. A szimuláció futásideje jelentősen megnő, hiszen mint HDL modell, a kapuszintű netlista rengeteg alkatrészt tartalmaz, eseményvezérelt szimulációs környezetben tehát futtatása meglehetősen nagy számítási kapacitást igényel. Ugyanakkor a valóságot is jobban tükrözi, mint egy RTL modellen futtatott szimuláció, hiszen a logikai kapuk és/vagy vezetékek késleltetését is figyelembe tudja venni. A netlista egyes celláinak és az azok közötti vezetékezésnek a késleltetését egy ún. SDF *(Standard Delay Format)* fájl írja le, amelyet a szintézis szoftverek a HDL modellel együtt bocsátanak rendelkezésünkre. Ezeket a szimulációkat ezért időzítési szimulációnak, vagy dinamikus időzítésvizsgálatnak is nevezzük.

Attól függően, hogy a szintézis folyamatának melyik pontján állítjuk elő a kapuszintű modellt, és vetjük alá logikai szimulációnak, az alábbi eseteket különböztethetjük meg:

- **Post-szintézis időzítési szimuláció:** A szimulált modell a generikus logikai kapukból álló netlista. Ez a zero-delay szimuláció az RTL modell szimulációjához képest nem sok többlet információval szolgál. A logikai szintézis algoritmusok ún. *„correct-per-construction"* működésűnek tekinthetők, vagyis ha a szintézis algoritmus képes volt feldolgozni a HDL leírást, akkor a generikus kapuszintű modell gyakorlatilag biztosan ugyanazt a viselkedést produkálja, mint az RTL modell. Éppen ezért ezt a szimulációt rendszerint nem is hajtjuk végre.

[80] Az általános logikai szintézis szoftverek csak magukat a logikai alapelemeket ismerik, az adott céleszköz globális huzalozási architektúráját nem.

- **Post-mapping időzítési szimuláció**: A szimulált modell a technológiafüggő kapuszintű modell. A generikus kapuszintű modelltől eltérően ez már tartalmazza a logikai kapuk és flip-flopok késleltetéseire vonatkozó információt. Az időzítési feltételek teljesítése azonban erősen függ a cellákat összekötő vezetékhálózat késleltetéseitől is, sőt a csíkszélesség csökkenésével egyre inkább ez utóbbi összetevő dominál, így ezt a fajta szimulációt is egyre kevésbé használjuk.
- **Post-layout / post-place&route időzítési szimuláció:** A szimulált modell a technológiafüggő kapuszintű netlista, és a hozzá rendelt SDF fájl már a cellák közötti vezetékhálózat késleltetését is figyelembe veszi. FPGA technológián véges számú vezetékszakasz-típus áll rendelkezésre, ezek késleltetése akár karakterizálható is, standard cellás technológián azonban minden vezeték gyakorlatilag egyedi, így a fémezési rétegek LEF-fájlban definiált tulajdonságai, valamint a vezetékek hossza és környezete (szomszédos vezetékszakaszok) alapján kell megbecsülni a késleltetést. A vezetékszakaszok késleltetésének meghatározását mindkét technológia esetén *„back-annotation”*-nek nevezzük. A post-layout / post-place&route kapuszintű modell a lehető legpontosabban írja le leendő áramkörünk viselkedését, így ezen a logikai szimulációt már érdemes elvégezni.

Az időzítési szimulációk meglehetősen időigényesek, és a szimulált áramkör celláinak számával ez az idő igen gyorsan nő. Tovább nehezíti a helyzetet, hogy az áramkör belső késleltetései a hőmérséklettől, a tápfeszültségtől és a gyártási szórásoktól is függenek, ezért az SDF fájlok valójában nem egy, hanem három különböző sarokpontban *(corner: minimal, nominal, maximal)* írják le a cellák késleltetését. Az időzítési szimulációkat tehát elvileg nem egy, hanem legalább kettő *(min, max)* de inkább három alkalommal kell lefuttatni.

Statikus időzítésvizsgálat (*Static Timing Analysis*, STA) • A fizikai szintézis eredményeként előálló modell az áramkör összes adatútjára vonatkozóan tartalmaz minden időzítési információt. Ez azt jelenti, hogy az egyes lokális adatutakat sorravéve a 6. fejezetben tárgyalt időzítési kényszereknek való megfelelés elvileg maradéktalanul ellenőrizhető. Pontosan ez a statikus időzítésvizsgálat feladata. A dinamikus időzítésvizsgálattal, vagyis az időzítési szimulációval ellentétben ez a módszer nem ellenőrzi azt, hogy a szintetizált netlista funkcionálisan ugyanazt valósítja-e meg, mint az RTL modellünk, csupán egy netlista adatútjainak késleltetéseit veti össze egy, a tervező által definiált időzítési kényszerrendszer (SDC) által meghatározott maximálisan megengedhető vagy minimálisan szükséges késleltetésekkel. Előnye, hogy rendkívül gyorsan, a dinamikus időzítési szimulációval összemérhetetlenül gyorsabban elvégezhető.

DRC *(Design Rule Check)* • Standard cellás technológia esetén azt is ellenőriznünk kell, hogy a GDSII fájlban definiált, végső fizikai terv valóban hibátlanul legyártható. Ehhez a layout-rajzolat tervének egy sor, a gyártó által definiált tervezési szabálynak – például egy adott fémezési rétegen megvalósított, két párhuzamos vezeték közötti minimális távolság, fémezési rétegek közötti átvezetések minimális mérete stb. – kell megfelelnie.

LVS *(Layout vs Schematic)* • Ugyancsak standard cellás technológiára jellemző ellenőrzés, amely a layout-rajzolat parazita elemeinek visszafejtéséből *(extract)*, majd az így kapott, kibővített kapcsolási sémának a fizikai szintézis bemeneteként megadott kapcsolási sémával való összehasonlításából áll.



9

Esettanulmány

A digitális rendszerek fejlesztése egy iteratív folyamat, amely során a felhasználói specifikációból kiindulva előbb magas elvonatkoztatású, absztrakt modelleket készítünk, majd a tervet fokozatosan finomítva, egyre több megvalósítási részletet kidolgozva jutunk el a működő áramkörig. Az iterációknak, tehát a már kidolgozott modelljeink finomításának, optimalizációjának szükségessége tipikusan két okra vezethető vissza. Amikor egy tervezői döntést meghozunk, akkor valójában csak feltételezzük, hogy az a specifikációnak megfelelő megoldásra vezet. E döntés helyességéről csak úgy győződhetünk meg, ha továbblépünk a tervezési folyamatban, majd az előálló eredményt valamiképp minősítjük, összevetjük a specifikációban foglaltakkal. Ha megoldásunk nem felel meg a specifikációnak, akkor értelemszerűen más megoldást kell kitalálnunk, tervünket módosítanunk kell. A másik eset, ha maga a specifikáció változik meg időközben, amely változáshoz alkalmazkodnunk kell.

E fejezet célja, hogy szemléltesse a tervezési folyamatnak ezt az iteratív jellegét. A MiniCore mikroprocesszor már több alkalommal bizonyult kiváló állatorvosi lónak, így ezúttal is ezt a rendszert használjuk fel példaként.

9.1 A MiniCore mikroprocesszor szintézise és optimalizációja

Tekintsük kiindulási alapnak a MiniCore mikroprocesszornak a 4.9. alfejezetben elkészített RTL HDL modelljét. Erről a modellről tudjuk (pontosabban én tudom, a kedves Olvasó pedig nyugodtan feltételezheti), hogy a specifikációnak megfelel, helyessége a megfelelő verifikációs módszerekkel bizonyíttatott. Ez azonban „csupán" egy RTL HDL modell, a működő áramkörtől még igencsak messze áll. Ehhez ti. a 8. fejezetben részletesen tárgyalt áramkörszintézist kell végrehajtanunk, ami – nem titkoltan szánt szándékkal – nem lesz problémamentes.

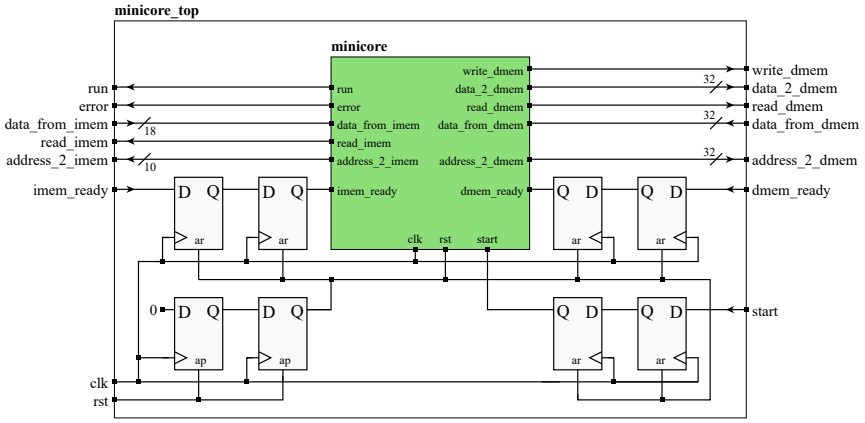
Ahogy azt a 4.9. alfejezetben a MiniCore processzor RTL modelljének elkészítése előtt is tettük, ezúttal is meg kell fogalmaznunk néhány konkrét megvalósítási részletet, amelyeket a tervezés során a specifikáció részének fogunk tekinteni. Ezek a részletek ezúttal a következők:

- **Céltechnológia**: A processzort FPGA-ban valósítjuk meg.
  - Eszköz: Microsemi RT ProASIC3 (RT3PE600L)
  - Tokozás: CQFP 256

- **Működési körülmények**: Egy áramkör belső késleltetései függenek a hőmérséklettől, a tápfeszültségtől és a gyártási szórásoktól egyaránt. A szintézis során többször is fogunk statikus időzítésvizsgálatot végezni annak megállapítására, hogy az automatikus úton előállított áramköri kapcsolás a megadott hőmérsékleti és tápfeszültség-tartományban megfelel-e az előírt időzítési feltételeknek.
  - Hőmérséklettartomány: -40 °C – 95 °C (chiphőmérséklet)
  - Tápfeszültség-tartomány: 1,425 V – 1,575 V, MIL *(military)* szabvány szerint
- **Működési frekvencia**: 20 MHz.

9.1.1 Az interfészjelek szinkronizációja a lokális órajelhez

A MiniCore processzort egymagában szintetizáljuk, feltételezve, hogy az adat- és programtárat az FPGA-n kívül, valamilyen más eszköz valósítja meg. E két külső összetevőről még azt sem feltételezzük, hogy ugyanarról az órajelről járnak, mint a processzor, ami azt jelenti, hogy minden bemenetünket a lokális órajelhez képest aszinkronnak kell tekintenünk, vagyis gondoskodnunk kell a megfelelő órajel-szinkronizáló fokozatokról a metastabilitás elkerülése érdekében. Az adat- és programtár-interfészek hand-shake jellegéből adódóan szerencsére ezek az órajel-szinkronizáló megoldások nagyon egyszerűek lehetnek; tulajdonképpen mindössze arra van szükség, hogy a hand-shake mechanizmust megvalósító bemeneteinkre egy-egy double-flop szinkronizálót építsünk be (lásd 9-1. ábra).



9-1. ábra. *Órajel-szinkronizáló fokozatok a toplevel HDL modellben*

Az ábrán azt is láthatjuk, hogy a double-flop szinkronizálókat nemcsak a memória-interfészek jelei, hanem a *start* jel esetén is alkalmaztuk, hiszen ezt is egy külső forrás hajtja meg, valamint elhelyeztük az aszinkron meghúzást és szinkron elengedést biztosító, magas-aktív reset-szinkronizáló áramkört is (vö. 6-43. ábra). Bár túlzás lenne toplevel integrációról beá szélni, hiszen a MiniCore processzort egymagában szintetizáljuk, ennek ellenére valóban létrehoztunk egy, a szinkronizáló fokozatokat és a processzor egy példányát tartalmazó,

minicore_top HDL modellt, kifejezve, hogy a szinkronizáló fokozatok nem képezik szerves részét magának a processzornak. Más körülmények között – például ha az adat- és programtárat egyazon FPGA-n belül, a processzoréval megegyező órajelről hajtva valósítanánk meg – ezekre a flip-flopokra nem lenne szükség.

9.1.2 Az időzítési kényszerek rögzítése

Ha visszatekintünk a 8-20. ábrára, láthatjuk, hogy az automatizált áramkörszintézis megz kezdéséhez a HDL modellen kívül szükségünk lesz az időzítési kényszereket formálisan rögzítő SDC fájlra is. Az órajelsémánk a lehető legegyszerűbb, egyetlen órajeltartományunk van, így a belső lokális adatutak mindegyikére egyetlen SDC parancs segítségével megfogalmazhatunk egy-egy időzítési kényszert (lásd 9-2. ábra).

VHDL

```
create_clock -name clk -period 50 [get_ports clk]
```

9-2. ábra. *A 20 MHz-es órajelet definiáló SDC fájl*

Az órajelfeltétel megfogalmazását általános esetben a ki- és bemeneti késleltetések rögzítése követi, esetünkben azonban erre nincs szükség, ui. egyetlen olyan interfészjelünk sincs, amelyiken az időzítési viszonyok pontos beállításán múlna a helyes működés. Bemeneti késleltetéseket nem határozunk meg, hiszen mindegyik bemenetünk az alábbi két kategória valamelyikébe tartozik:

- Órajelszinkronizáló fokozattal védett a metastabilitás okozta hibákkal szemben. Ezeken a bemeneteken azért felesleges az időzítési feltételek pontos megfogalmazása, mert a double-flop szinkronizáló fokozatok pontosan azért vannak ott, hogy a nem megfelelő időzítések által okozott hibákat kiszűrjék.
- Hand-shake interfész szándékosan szinkronizálatlan adatbusza. Ezeken a buszokon azok mintavételezésének pillanatában mindenképpen stabil jelszintek vannak, amire a hand-shake interfész időzítési modellje a biztosíték.

Kimeneti késleltetéseket szintén nem rögzítünk, feltételezzük, hogy a külső eszközök hasonlóképpen kezelik a processzorral közös interfészüket, mint ahogy mi is tesszük, így elvileg teljesen mindegy, hogy az általunk meghajtott jelek és a mi saját órajelünk felfutó élei milyen viszonyban állnak egymással.

Az SDC fájlban elvileg megfogalmazhatnánk időzítési kivételeket is (pl. többciklusú vagy érdektelen adatutakra vonatkozóan), ezt azonban rendszerint csak akkor tesszük, ha épp az adott adatút bizonyul a szűk keresztmetszetnek az előírt órajel-frekvencia elérésében. Elmondható tehát, hogy a 9-2. ábrán látható SDC paranccsal megfelelően rögzítettük az időzítési peremfeltételeket.

Az áramkörszintézis 8.2. alfejezetben bemutatott lépéseit szoftverek segítségével végezzük el.[81] Célunk nem e szoftverek használatának ismertetése, hanem annak bemutatása, hogy RTL tervezőként tipikusan milyen optimalizációs módszereket alkalmazhatunk és hogy ezek milyen hatással vannak az automatizált szintézis eredményére. Az egyes optimalizációs lépések után tervünk megfelelőségét a kapott áramkör erőforrásigénye és a fizikai szintézist követő statikus időzítésvizsgálat eredménye alapján állapítjuk meg.

9.1.3 *Időzítési optimalizáció* (timing closure)

A szintézist legelőször a MiniCore processzor már ismertetett HDL modelljén végezzük el. A kapott áramkör erőforrásigényét a 9-1., a statikus időzítésvizsgálat eredményét pedig a 9-2. táblázat foglalja össze.

FPGA erőforrás	Felhasznált mennyiség	Rendelkezésre álló mennyiség	Kihasználtság
VersaTile[82]	6379	13824	46,14 %
blokk RAM modul[83]	4	24	16,67 %

9-1. táblázat. *Az áramkör erőforrásigénye #1*

Órajelfeltétel		Kritikus út			
		kibocsátó flip-flop	fogadó flip-flop	t_D	maradékidő (slack)
f_{clk}	t_{Dmax}	divisor / iterator[4]	divisor / remainder_core[31]	52,059 ns	**-2,059 ns**
20 MHz	50 ns				

9-2. táblázat. *A statikus időzítésvizsgálat eredménye #1*

A 9-2. táblázatból látható, hogy a kapott áramkör legnagyobb késleltetésű lokális adatútának, vagyis a kritikus útjának késleltetése az órajelfeltételből adódó, maximálisan megengedhető 50 ns-nál nagyobb, az áramkör tehát nem felel meg az időzítési feltételeknek. Az időzítési tulajdonságokon a HDL modell módosításával szándékozunk javítani, ehhez azonban először meg kell állapítanunk, hogy a kritikus út pontosan hol van. A statikus időzítésvizsgálatot végző szoftverek szerencsére minden szükséges információval ellátnak bennünket. A legapróbb részletekig megvizsgálhatjuk az egyes lokális adatutak késleltetésének összetevőit, beleértve a logikai kapukat és vezetékszakaszokat. Az esetek többségében a HDL modell módosítandó részleteinek lokalizálásához már az is elegendő, ha a kritikus út két végpontját, a kibocsátó és a fogadó flip-flopokat beazonosítjuk. Esetünkben

[81] Logikai szintézis + technológiai leképezés: Synopsys Synplify Pro, fizikai szintézis: Microsemi Designer.
[82] A VersaTile a Microsemi ProASIC3 FPGA család konfigurálható logikai alapeleme, amely egy flip-flop vagy egy maximum három bemenetű kombinációs hálózat funkcióját tudja ellátni.
[83] 512 × 18 bit

VHDL

```
L_DIVISION_CORE: process ( clk, rst )
  variable tmp: unsigned (w-1 downto 0);
begin
  if ( rst = '1' ) then
    remainder_core <= (others => '0');
    quotient_core <= (others => '0');
  elsif ( rising_edge(clk) ) then
    if ( reset_core = '1' ) then
      remainder_core <= (others => '0');
      quotient_core <= (others => '0');
    else
      if ( ena_core = '1' ) then
        tmp := shift_left(remainder_core, 1);
        tmp(0) := dividend_abs(iterator);
        if ( tmp >= divisor_abs ) then
          remainder_core <= tmp - divisor_abs;
          quotient_core(iterator) <= '1';
        else
          remainder_core <= tmp;
        end if;
      end if;
    end if;
  end if;
end process;
```

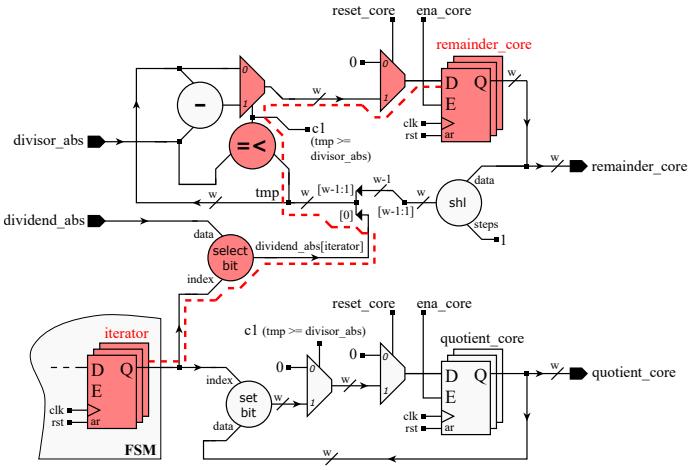
Verilog

```
// DIVISION_CORE
always @ ( posedge clk, posedge rst )
begin
  if ( rst == 1'b1 ) begin
    remainder_core <= 0; quotient_core <= 0;
  end else begin
    if ( reset_core == 1'b1 ) begin
      remainder_core <= 0; quotient_core <= 0;
    end else begin

      if ( ena_core == 1'b1 ) begin
        tmp = remainder_core << 1;
        tmp[0] = dividend_abs[iterator];
        if ( tmp >= divisor_abs ) begin
          remainder_core <= tmp - divisor_abs;
          quotient_core[iterator] <= 1'b1;
        end else begin
          remainder_core <= tmp;
        end
      end

    end
  end
end
```

Áramköri struktúra



9-3. ábra. *Kritikus út az osztó áramkörben*

a kritikus út az osztó áramkör `iterator` és `remainder_core` nevű regiszterei között van, ahogy azt a 9-3. ábra az RTL kapcsolási sémán is szemlélteti.[84]

A kritikus út két fő összetevője az RTL kapcsolási sémán `select_bit` névvel ellátott kombinációs hálózat, ami a `dividend_abs` jelnek az `iterator` által kiválasztott bitjét kapcsolja a kimenetére, valamint a `tmp` és a `dividend_abs` jelek értékét összehasonlító komparátor. A HDL modellben piros kiemeléssel jelöltük azokat a sorokat, amelyek következményeként

[84] Érdekesség, hogy a kritikus út a vezérlő állapotgép egyik kimeneti buffere és egy, a műveletvégző egység részét képező flip-flop között alakult ki. Ha összevetjük ezt a 4.5.5. alfejezetben a különböző állapotgép-megvalósításokról elmondottakkal, láthatjuk, hogy a kimeneti bufferek alkalmazása miért lehet előnyös; ha az osztó vezérlését végző állapotgépnek nem lennének kimeneti bufferei, akkor ez a lokális adatút az állapotgép kimeneti logikáját is magában foglalná, tehát a késleltetése a jelenleginél is nagyobb lenne, az időzítések helyrehozása tehát sokkal nehezebb lenne.

e kombinációs hálózatok előálltak. Időzítési szempontból tulajdonképpen nem magukkal a kombinációs hálózatokkal van probléma, inkább azzal, hogy sorba vannak kapcsolva, késleltetéseik tehát összeadódnak. A sorbakapcsoltság oka az, hogy a VHDL modellben `tmp`-t változóként (`variable`) deklaráltuk, a Verilog modellben pedig blokkoló értékadással adtunk neki értéket, ami áramköri szempontból ugyanazt jelenti. Ahogy minden, szinkron folyamatba foglalt értékadás, úgy a `remainder_core` regiszterre vonatkozó értékadás is közvetetten leírja a regisztert meghajtó kombinációs hálózatot. Ha az értékadás jobb oldalán VHDL-ben változó vagy Verilog esetén blokkoló értékadással használt jel van, akkor az annak előállítását végző kombinációs hálózatok szintén részét fogják képezni a `remainder_core`

VHDL

```
L_DIVISION_CORE: process ( clk, rst )
begin
  if ( rst = '1' ) then
    remainder_core <= (others => '0');
    quotient_core <= (others => '0');
    tmp <= (others => '0');
  elsif ( rising_edge(clk) ) then
    if ( reset_core = '1' ) then
      remainder_core <= (others => '0');
      quotient_core <= (others => '0');
    else

      if ( ena_core = '1' ) then
        tmp <= shift_left(remainder_core, 1);
        tmp(0) <= dividend_abs(iterator);
      end if;

      if ( ena_core_2 = '1' ) then
        if ( tmp >= divisor_abs ) then
          remainder_core <= tmp - divisor_abs;
          quotient_core(iterator) <= '1';
        else remainder_core <= tmp;
        end if;
      end if;

    end if;
  end if;
end process;
```

Verilog

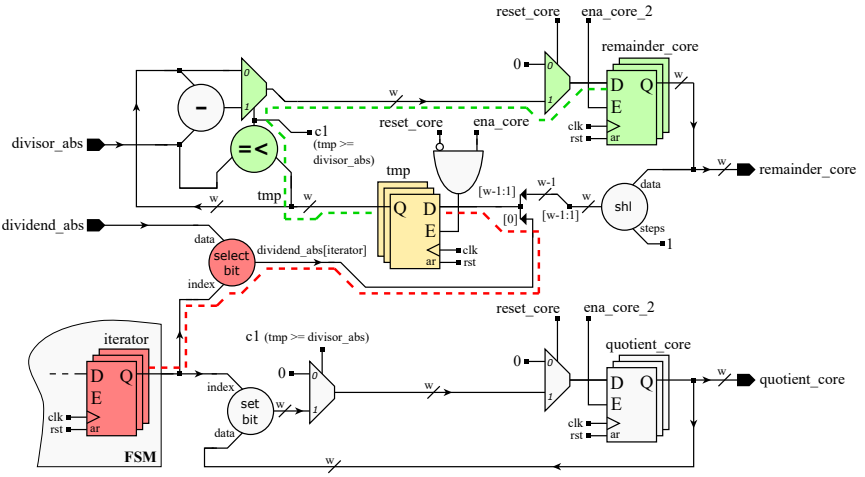
```
// DIVISION_CORE
always @ ( posedge clk, posedge rst )
begin
  if ( rst == 1'b1 ) begin
    remainder_core <= 0; quotient_core <= 0;
    tmp <= 0;
  end else begin
    if ( reset_core == 1'b1 ) begin
      remainder_core <= 0; quotient_core <= 0;
    end else begin

      if ( ena_core == 1'b1 ) begin
        tmp <= remainder_core << 1;
        tmp[0] <= dividend_abs[iterator];
      end

      if ( ena_core_2 == 1'b1 ) begin
        if ( tmp >= divisor_abs ) begin
          remainder_core <= tmp - divisor_abs;
          quotient_core[iterator] <= 1'b1;
        end else begin
          remainder_core <= tmp;
        end
      end

    end
  end
end
```

Áramköri struktúra



9-4. ábra. *Regiszterréteg beiktatása a kritikus útba*

új értékét kiszámító teljes kombinációs hálózatnak (vö. 4-24. és 4-26. ábra). A túl nagy késleltetésű lokális adatutak problémájára egy lehetséges megoldás, ha a kibocsátó és fogadó flip-flopok közötti, nagyméretű kombinációs hálózatot egy regiszterréteggel „kettészeljük", így két, egyenként kisebb késleltetésű lokális adatutat kapunk. Az osztó áramkör műveletvégző egységében a regiszterréteg beiktatását úgy valósítjuk meg, hogy `tmp`-t a VHDL modellben nem változóként (`variable`), hanem jelként (`signal`) deklaráljuk, a Verilog modellben pedig a blokkoló értékadást nem-blokkolóra cseréljük le (lásd 9-4. ábra).

VHDL

```
...
when s2     =>  ena_pre <= '0';
                ena_core_2 <= '0';
                if ( iterator > 0 ) then
                  ena_core <= '1';
                  iterator <= iterator - 1;
                  state <= t1;
                else
                  ena_core <= '0';
                  iterator <= w-1;
                  ena_post <= '1';
                  state <= s3;
                end if;

when t1     =>  ena_core <= '0';
                ena_core_2 <= '1';
                state <= s2;

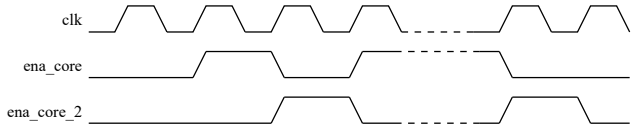
...
```

Verilog

```
...
s2: begin
      ena_pre <= 1'b0;
      ena_core_2 <= 1'b0;
      if ( iterator > 0 ) begin
        ena_core <= 1'b1;
        iterator <= iterator - 1;
        state <= t1;
      end else begin
        ena_core <= 1'b0; iterator <= w-1;
        ena_post <= 1'b1; state <= s3;
      end
    end

t1: begin
      ena_core <= 1'b0; ena_core_2 <= 1'b1;
      state <= s2;
    end
...
```

Áramköri struktúra



9-5. ábra. *Új vezérlőjel és vezérlési állapot hozzáadása az állapotgéphez*

Így tulajdonképpen azt értük el, hogy a `tmp` jel immár maga is egy regiszterként viselkedik. Ez persze azt is jelenti, hogy amit az áramkör eddig egyetlen órajelciklus alatt végzett el, ahhoz most két órajelciklusra lesz szükség; az elsőben kiszámítjuk a `tmp`, a másodikban pedig a `remainder_core` regiszter új értékét. A fokozat működését engedélyező `ena_core` vezérlőjel mellett bevezetünk egy `ena_core_2` jelet is, és ezeket mindig két egymást követő órajelciklusban aktiváljuk. A vezérlést végző állapotgépben ehhez egy új vezérlési állapotot kell létrehoznunk (lásd 9-5. ábra, vö. 4-87. ábra).

A szintézist, majd azt követően a statikus időzítésvizsgálatot újrafuttatva a módosított HDL modellen a 9-3. és 9-4. táblázatban foglalt eredményeket kapjuk.

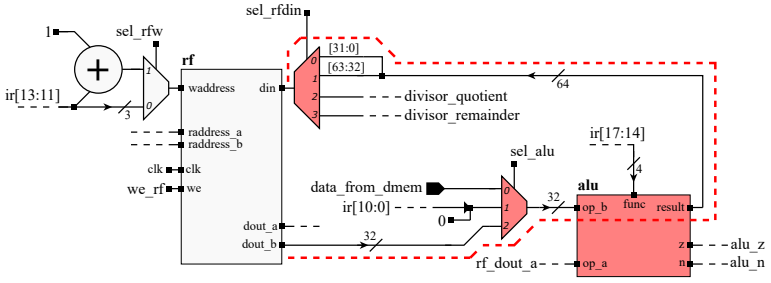
FPGA erőforrás	Felhasznált mennyiség	Rendelkezésre álló mennyiség	Kihasználtság
VersaTile	6434	13824	46,54 %
blokk RAM modul	4	24	16,67 %

9-3. táblázat. *Az áramkör erőforrásigénye #2*

Órajelfeltétel		Kritikus út			
		kibocsátó flip-flop	fogadó flip-flop	t_D	maradékidő (slack)
f_{clk}	t_{Dmax}	rf/doutb[0]	rf/din[31]	52,43 ns	-2,43 ns
20 MHz	50 ns				

9-4. táblázat. *A statikus időzítésvizsgálat eredménye #2*

Az eredményekből az látszik, hogy bár az áramkör továbbra sem felel meg az időzítési feltételeknek, de a kritikus út ezúttal nem az osztó áramkörben alakult ki, tehát a regiszterréteg beiktatása eredményes volt.[85] A kritikus út ezúttal a regisztertömb `b` adatkimenete és az adatbemenete között alakult ki. A 9-6. ábrán látható RTL kapcsolási sémára tekintve könnyen látható, hogy mi ennek az oka.



9-6. ábra. *Kritikus út a regisztertömb adatkimenete és adatbemenete között*

A kritikus út legfőbb összetevője ezúttal egyértelműen az ALU, ami nem meglepő, ha visszatekintünk a 4-96. ábrára, láthatjuk, hogy az ALU-t egy kombinációs folyamattal írtuk le, ami áramköri szempontból egy igencsak méretes kombinációs hálózatot jelent. A megoldás ezúttal is az lehet, hogy a jelútba egy regiszterréteget iktatunk, a kérdés csak az, hogy ezt hová érdemes elhelyezni. Ha a kritikus utat két, nagyjából egyforma késleltetésű jelútra szeretnénk szétbontani, akkor mindenképp az ALU belsejébe kellene elhelyezni a regisztereket, ez azonban magának az ALU-nak a jelentős módosításával oldható csak meg. Megtehetjük azt is, hogy az ALU-nak a 64-bites kimenetére kapcsolunk regiszterréteget, ami a HDL modellben egyszerűbben kivitelezhető. Ekkor a kritikus utat két, nagyon eltérő késleltetésű, kisebb adatútra bontanánk szét, hiszen az egyik tartalmazna egy multiplexert és a teljes ALU logikát, a másik pedig csak egy multiplexert. Ez nem feltétlenül gond, hiszen csupán 2,43 ns-ot kell nyernünk. Ez a megoldás azonban 64 hozzáadott regisztert

[85] Figyeljük meg, hogy a kritikus út késleltetése ezúttal nagyobb, mint a legelső esetben volt. Felmerülhet bennünk, hogy ez hogyan lehetséges, hiszen ha ennek az adatútnak a késleltetése nagyobb, mint az előző szintézis során kapott áramkörben a legnagyobb késleltetés, akkor abban a fordulóban hogyhogy nem ez volt a kritikus út. Nos, a válasz az, hogy a második körben az áramkörben egészen más adatutak alakulnak ki, mivel megváltozott a HDL modell. Ahogy azt a 8.2.2. pontban említettük, az elhelyező és huzalozó algoritmusok véletleneket használó, heurisztikus optimalizációs eljárások, amelyeknek két, egymást követő lefuttatása csak akkor vezet ugyanarra az eredményre, ha sem a HDL kódbázison, sem a peremfeltételeken semmit sem változtattunk. Azzal, hogy módosítottuk a HDL modellt, elhelyezés szempontjából egy teljesen más áramkört kaptunk.

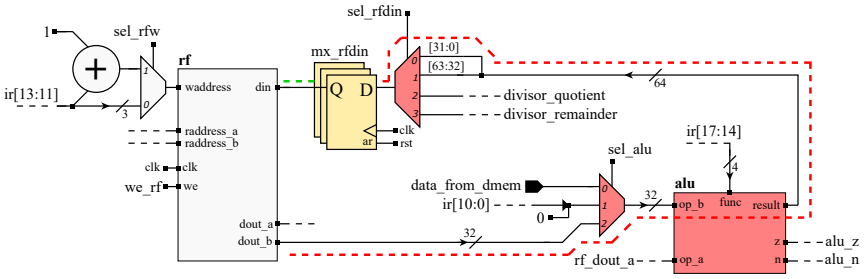
VHDL

```
process ( clk, rst )
begin
  if ( rst = '1' ) then
    mx_rfdin <= (others => '0');
  elsif ( rising_edge(clk) ) then
    mx_rfdin <= alu_result(31 downto 0) when
                sel_rfdin = "00" else
                alu_result(63 downto 32) when
                sel_rfdin = "01" else
                div_quotient when
                sel_rfdin = "10" else
                div_remainder;
  end if;
end process;
```

Verilog

```
always @ ( posedge clk, posedge rst )
begin
  if ( rst == 1'b1 ) mx_rfdin <= 32'd0;
  else
    case ( sel_rfdin )
      2'b00:    mx_rfdin <= alu_result[31:0];
      2'b01:    mx_rfdin <= alu_result[63:32];
      2'b10:    mx_rfdin <= div_quotient;
      default:  mx_rfdin <= div_remainder;
    endcase
end
```

Áramköri struktúra



9-7. ábra. *Regiszterréteg beiktatása a multiplexer és a regisztertömb közé*

jelentene. Ha részletesebben megvizsgáljuk a kritikus út késleltetésének összetevőit, azt látjuk, hogy azt csupán 30,69%-ban adják a logikai kapuk, a maradék 69,31% az vezeték-késleltetés. Azt is észrevehetjük, hogy a regisztertömb adatbemenetét meghajtó multiplexer és a regisztertömb közötti vezetékszakasznak önmagában mintegy 3,2 ns a késleltetése, ami elég jelentős. A HDL modell módosítása szempontjából ezen a ponton beiktatni a regiszterréteget elég egyszerű, nincs más dolgunk, mint hogy szinkronná tegyük magát a multiplexert (lásd 9-7. ábra, vö. 4-95. ábra).

Ezzel a megoldással a 64 hozzáadott regiszter helyett csak 32-t használunk, tehát az erőforrásigény kevésbé fog megnőni. A jelútba iktatott regiszterek természetesen ezúttal is azt eredményezik, hogy a műveletvégzés egy órajelciklussal több időt vesz igénybe, és a műveletvégző egységnek ezt a többletkésleltetését figyelembe kell vennünk a vezérlő egység állapotgépében is; a módosított adatutat használó utasítások vezérlési állapotait az osztó áramkörnél látottakhoz hasonlóan módosítani kell, adott esetben új vezérlési állapotok hozzáadásával. Mindezeket a módosításokat is végrehajtva, majd a szintézist és a statikus időzítésvizsgálatot újrafuttatva a módosított HDL modellen a 9-5. és 9-6. táblázatban foglalt eredményeket kapjuk.

FPGA erőforrás	Felhasznált mennyiség	Rendelkezésre álló mennyiség	Kihasználtság
VersaTile	6460	13824	46,73 %
blokk RAM modul	4	24	16,67 %

9-5. táblázat. *A áramkör erőforrásigénye #3*

Órajelfeltétel		Kritikus út			
		kibocsátó flip-flop	fogadó flip-flop	t_D	maradékidő (slack)
f_{clk}	t_{Dmax}	rf/doutb[7]	mx_rfdin[29]	48,222 ns	**1,778 ns**
20 MHz	50 ns				

9-6. táblázat. *A statikus időzítésvizsgálat eredménye #3*

9.1.4 „Legyen űrös...”

Az eddigiekben arra láthattunk példát, hogy az RTL tervezés során meghozott döntés (például egy aritmetikai egység aszinkron megvalósítása) miként vezethet a logikai és fizikai szintézis elvégzése után olyan áramkörhöz, ami nem felel meg a specifikációnak. A hibásnak bizonyuló döntés eredménye egy iterációs hurok lett a tervezési folyamatban; a HDL modellt módosítani kellett, a funkcionális verifikációt és a szintézist újra el kellett végezni a módosított modellen. Iterációra, részleges újratervezésre természetesen akkor is sor kerülhet, ha a specifikáció menet közben megváltozik. Tegyük fel, hogy a MiniCore processzornak a specifikációban eredetileg is megjelölt FPGA-ban való megvalósításával szemben az az elvárás fogalmazódik meg, hogy űreszközök fedélzetén is használhatónak kell lennie úgy, hogy a várható TID *(Total Ionizing Dose)* 20 krad. A céltechnológiaként megjelölt RT ProASIC3 FPGA család nevében az „RT” annyit tesz: *Rad Tolerant*, a tokozást azonosító CQFP 256 kódban pedig a „C” a kerámiatokra *(ceramic)* utal, ami szintén a sugárzásnak némiképp ellenálló eszközökre jellemző. Maga a céltechnológia tehát alapvetően alkalmas arra, hogy űreszközök fedélzetén használjuk, azonban az űripari szabványoknak, általános tervezési elveknek való megfelelés igénye egy egész sor változtatást tesz szükségessé a processzor RTL HDL modelljét és a szintézis szoftverek beállításait illetően. E változtatások mindegyike egyaránt hatással lesz az előálló áramkörnek az erőforrásigényére és késleltetési viszonyaira.

A világűrben eszközeinket sokféle különböző hatás érheti, amelyek közül a félvezető eszközök rövid- és hosszútávú működésére az ionizáló sugárzások (részecskesugárzás és nagyenergiájú elektromágneses sugárzás) jelentik a legnagyobb veszélyt. E sugárzások többféle forrásból származnak:

- **Napszél**: A Nap atmosztférájának legkülső rétegéből, a koronából származó, elektronokból, protonokból és alfa-részecskékből álló részecskeáram. Ez egy időben többé-kevésbé állandó fluxusú áramlás, amelynek állandóságát időnként a Nap mágneses terének instabilitása okozta, hirtelen bekövetkező, nagy mennyiségű plazmát a bolygóközi térbe juttató ún. koronakidobódások (*Coronal Mass Ejection*, CME) törik meg.
- **Kozmikus sugárzás**: A csillagközi térből származó, minden irányból érkező, protonokból, alfa-részecskékből és nehéz ionokból álló részecskesugárzás, valamint röntgen- és gammasugárzás.

- **Van Allen sugárzási övek**: A Földet körülvevő két, tórusz alakú övezet, ahol a Föld mágneses tere csapdába ejti a napszélből és a kozmikus sugárzásból származó részecskéket. A belső öv a földfelszíntől számított 2000-5000 km-re, a külső pedig 15000-20000 km-re helyezkedik el.

A részecske- és elektromágneses sugárzások hatása típusuktól és energiájuktól függően más és más, ami közös bennük, hogy a félvezető eszközökben töltéshordozó-generációt idéznek elő. A generálódott töltéshordozók a vezető rétegekben szabadon mozognak, így hamar rekombinálódhatnak, itt tehát tranziens hatásról beszélhetünk. A tranziens hatások közül a digitális áramkörök szempontjából a legfontosabbak a következők:

- **SEL** *(Single-Event Latchup)*: A nagyenergiájú részecske becsapódása hatására a félvezető eszköz rétegszerkezetében jelenlévő parazita bipoláris tranzisztor kinyit, rövidzárat hozva létre a táp- és földsín között. A rövidzáron folyó áram nagyságától függően ez akár az eszköz tönkremeneteléhez is vezethet.
- **SET** *(Single-Event Transient)*: A nagyenergiájú részecske becsapódása ideiglenes feszültségváltozást idéz elő az integrált áramkör egy csomópontján. Ez akkor a legveszélyesebb, ha ez a feszültségtüske a digitális áramkör globális jeleinek egyikén, az órajelhálózaton vagy a reset-hálózaton következik be.
- **SEU** *(Single-Event Upset)*: A nagyenergiájú részecske becsapódásának hatására egy tároló elem (flip-flop vagy SRAM cella) értéke megváltozik, az áramkör logikai működésétől teljesen függetlenül, ami értelemszerűen funkcionális hibát okozhat.

A szigetelő rétegekben generálódó többlet töltéshordozók hatása egészen más, ezek ui. nem mozognak szabadon, csapdába eshetnek. Az idővel fokozatosan felhalmozódó töltéshordozók módosítják a belső potenciálviszonyokat, ami az eszköz paramétereinek folyamatos változásában nyilvánul meg. CMOS kapcsolástechnika esetén ennek leglátványosabb esete a MOS tranzisztorok kapuelektródája alatti szigetelőrétegben, a gate-oxidban csapdázódó töltéshordozók okozta küszöbfeszültség-eltolódás. A sugárzásnak való kitettség időtartamának növekedésével a tranzisztor küszöbfeszültsége annyira eltolódik, hogy az eszköz már gyakorlatilag nem működik tranzisztorként. E jelenséget számszerűsíti az adott eszközre jellemző TID érték, ami azt mutatja meg, hogy mekkora az eszköz által elviselhető legnagyobb teljes sugárdózis. Állandó környezeti sugárzást feltételezve ez az érték tulajdonképpen az eszköz várható élettartamáról ad tájékoztatást.

A sugárzások elleni védelemnek több szintje van, az űreszköz megfelelő burkolata, az elektronikus eszközök PCB-szintű árnyékolása és maguknak az integrált áramköröknek a tokozása is mind egy-egy védelmi vonalat jelent. Az integrált áramkörökben alkalmazott kapcsolástechnikai és layout-tervezési megoldások is javítják a sugárzás hatásainak való ellenállás mértékét, ezáltal a megbízhatóságot és az élettartamot, de fontos tudni, hogy e megoldások mind egyúttal az összetevők árát is jelentősen növelik, így – az adott alkalmazás sugárzásnak való várható kitettségének megfelelően – itt is kompromisszumos megoldásokat keresünk.

9.1.4.1 A céltechnológia sugárzásállósága

A specifikációban megadott környezeti körülmények – jelen esetben a teljes várható sugárdózis – ismeretében fel kell mérnünk, hogy RTL tervezőként milyen „trükköket" kell bevetnünk a leendő áramkör megbízhatóságának növelése érdekében. Ehhez először is jól kell ismernünk magát a céleszközt, tudnunk kell, hogy annak milyen képességeire támaszkodhatunk, és mi az, amire maga az eszköz nem kínál megoldást. A példánkban céltechnológiaként megjelölt RT ProASIC3 FPGA egy ún. „real-FLASH" eszköz,[86] ami annyit jelent, hogy a logikai alapelemek közötti konfigurálható összeköttetéseket egy-egy lebegő kapuelektródás FLASH cella valósítja meg (lásd 7-11. ábra: *b*). Ebben a struktúrában az összeköttetés meglétét vagy nem meglétét a FLASH cella lebegő kapuelektródáján lévő töltés mennyisége határozza meg. Ez azt jelenti, hogy bár a konfigurációt tekinthetjük a tranziens jellegű hatásokkal szemben védettnek, de a TID effektusok gondot okozhatnak a konfigurálható összeköttetéseknél. Ahogy a sugárzás hatására az összeköttetéseket megvalósító FLASH cella lebegő kapuelektródáján lévő töltésmennyiség változik, úgy az összeköttetések minősége egyre romlik, a lokális adatutak késleltetése egyre nő. A gyártó az adatlapon megadja, hogy 30 krad teljes dózis esetén legfeljebb 10%-os késleltetésnövekedésre kell számítani, amit az időzítésvizsgálat során figyelembe kell vennünk.

Az RT ProASIC3 FPGA alkalmazása esetén figyelembe kell vennünk azt is, hogy az eszköz flip-flopjai és a blokk-RAM modulok SRAM cellái kapcsolástechnikailag nem TMR megvalósításúak, a nagyenergiájú részecskék becsapódásának hatása (SEU) ellen tehát nem védettek, ez ügyben valamit tennünk kell az RTL tervezés során.

9.1.4.2 SEU-védelem

A SEU-védettség hiánya a flip-flopok esetén könnyen kezelhető a logikai szintézis szintjén a 8-31. ábrán bemutatott soft register-TMR technikával. Ehhez mindössze egyetlen parancsot kell hozzáadnunk a tervezési peremfeltételeket tartalmazó SDC fájlhoz (lásd 9-8. ábra), a többit a szintézis eszköz automatikusan elvégzi.

SDC

```
create_clock -name clk -period 50 [get_ports clk]
define_global_attribute {syn_radhardlevel} {tmr}
```

9-8. ábra. *Soft register-TMR megvalósítást rögzítő SDC parancs*

[86] A „real" jelzőt a gyártó azért használja, hogy megkülönböztesse a saját megoldását a más gyártók (pl. Intel) FLASH-alapúként aposztrofált technológiáitól. Ez utóbbiakban ui. a konfigurációt egy on-chip FLASH memória tárolja, amelynek tartalmával minden bekapcsoláskor valójában egy szokványos SRAM konfigurációs memóriát inicializálnak. Ez a megoldás két fontos dologban különbözik a „real-FLASH" technológiától; egyrészt a bekapcsolás után néhány milliszekundum időtartamig (amíg a konfiguráció átkerül a FLASH memóriából az SRAM-ba) az eszköz még nem működik, másrészt az ilyen eszközök konfigurációja sugárzás hatására könnyen megsérülhet a SEU-jelenségre hajlamos SRAM cellák miatt. Így ezek az eszközök tulajdonképpen a FLASH technológiának csak a nemillékony *(non-volatile)* jellegéből adódó előnyöket nyújtják, míg a „real-FLASH" megoldás a bekapcsolás utáni azonnali működőképességet (*Live At Power-Up*, LAPU), az alacsony fogyasztást és a konfiguráció sugárzás elleni védettségét is biztosítja.

A MiniCore processzor belső regisztertömbjének HDL modellje alkalmas arra, hogy a szintézis szoftver blokk-RAM modulok felhasználásával valósítsa meg azt, és a tárolt információ mennyisége alapján – az erőforrásigény csökkentése céljából – a szintézis eszköz kétségkívül élni is fog ezzel a lehetőséggel. Sajnos a blokk-RAM modulokat felépítő hat tranzisztoros SRAM cellák kifejezetten hajlamosak a SEU-átbillenésre. A blokk-RAM modulok SEU-védelmére a legelterjedtebb megoldás az ún. EDAC *(Error Detection And Correction)*-kódolás és EDAC segédáramkörök alkalmazása, amelyek segítségével a memóriában nemcsak magát az adatot, hanem egy extra biteket tartalmazó kódszót tárolunk el, amelyből az eredeti adatszó akkor is visszaállítható, ha a tárolás során az megsérült.[87] A MiniCore mikroprocesszor esetén az EDAC-logika beépítése a belső regisztertömb időzítési modelljét jócskán átalakítaná, mivel a kódolást és dekódolást tipikusan – különösen ilyen nagy bitszélességek esetén – nem teljesen aszinkron módon, hanem pipeline regiszterek beiktatásával végezzük. Ez jelentős módosítást jelentene magának a regisztertömbnek és a műveletvégző egység vezérlését végző állapotgépnek a modelljén is. Ehelyett a problémát inkább úgy kezeljük, hogy a regisztertömb megvalósítására egyáltalán nem használunk blokk-RAM modulokat, a regisztertömb minden bitjét egy-egy flip-floppal valósítjuk meg. Ez – a soft register-TMR-ből adódóan – fizikailag három flip-flopot fog jelenti bitenként, ami az erőforrásigény jelentős megnövekedését fogja eredményezni, de így az áramkör immár egészen komoly védettséget fog élvezni a nagyenergiájú részecskék okozta SEU-jelenséggel szemben. A 9-9. ábrán az látható, hogy a regisztertömböt megvalósító jelet VHDL-ben egy, a szintézis szoftver által definiált attribútummal látjuk el, Verilogban pedig egy pragmát helyezünk el ugyanannak a jelnek a deklarációja mellé, így kényszerítve ki a blokk-RAM modulok használatának mellőzését.

VHDL

```vhdl
type rf_content_t is array (0 to 7) of std_logic_vector (31 downto 0);
signal rf_content: rf_content_t;
attribute syn_ramstyle: string;
attribute syn_ramstyle of rf_content: signal is "registers";
```

Verilog

```verilog
logic [31:0] rf_content [0:7] /* synthesis syn_ramstyle = "registers" */;
```

9-9. ábra. *Blokk-RAM modulok használatának letiltása a megfelelő szintézis attribútum/pragma segítségével*

[87] Kiolvasáskor az EDAC-áramkör megvizsgálja a kiolvasott kódszót, és megállapítja, hogy keletkezett-e benne bithiba, amíg az SRAM-ban volt. Ha az alkalmazott kódolási módszer képes az adott hibát kijavítani, akkor a felhasználó a már javított adatot kapja meg. Ha az EDAC-áramkör nem tudja kijavítani a hibát, akkor értesíti a felhasználót, hogy a kiolvasott adat hibás. Az FPGA-k blokk-RAM moduljainak védelme érdekében általában egy extra paritásbittel kiegészített Hamming-kódolást, az ún. SECDED *(Single Error Correction, Double Error Detection)* kódolást alkalmazzák, amely képes bármely egybites hiba kijavítására. A kétbites hibákat ez a kódolási módszer képes ugyan detektálni (a felhasználót tehát tudja értesíteni a kétbites hibákról), de javítani nem tudja azokat. Ha az információ sok időt tölt a memóriában, akkor a több-bites hibák gyakorisága megnő, ahogy a nagyenergiájú részecskék becsapódása egyre több SEU-átbillenést idéz elő. A megoldás az ún. *„scrubbing”*; bizonyos időközönként az EDAC-áramkör egy adott memóriaterület minden bejegyzését kiolvassa, ellenőrzi, az esetleges hibákat kijavítja, majd a javított értéket visszaírja a memóriába. Ez a funkció általában az EDAC-moduloknak egy opcionális szolgáltatása, amelynek szintézise a megfelelő generikus paraméterrel letiltható. A scrubbing-logika működése a felhasználó szempontjából transzparens, csak olyankor írja/olvassa a memóriát, amikor a primer interfészeken nincs aktivitás.

A szintézist és a statikus időzítésvizsgálatot újrafuttatva a 9-7. és 9-8. táblázatban látható eredményeket kapjuk.

FPGA erőforrás	Felhasznált mennyiség	Rendelkezésre álló mennyiség	Kihasználtság
VersaTile	11383	13824	82,34 %
blokk RAM modul	0	24	0 %

9-7. táblázat. *A áramkör erőforrásigénye #4*

Órajelfeltétel		Kritikus út			
		kibocsátó flip-flop	fogadó flip-flop	t_D	maradékidő (slack)
f_{clk}	t_{Dmax}	sel_alu/ tmr2[1]	mx_rfdin/ tmr3[23]	50,524 ns	**-0,524 ns**
20 MHz	50 ns				

9-8. táblázat. *A statikus időzítésvizsgálat eredménye #4*

Az áramkör erőforrásigénye jelentősen megnőtt a soft register-TMR-nek köszönhetően, és az is látható, hogy az időzítési feltételeknek sem felel meg. A maradékidő negatív ugyan, de értéke nem túl nagy, ezért ahelyett, hogy azonnal időzítési optimalizációba kezdenénk, végezzük el a többi, megbízhatóságot növelő módosítást is, ezek ui. jelentősen át fogják alakítani a kapuszintű modellt, a lokális adatutak szerkezetét, így az időzítési viszonyokat is.

9.1.4.3 Nagymegbízhatóságú állapotgépek

Nagymegbízhatóságú alkalmazásokban az állapotgépek SEU-védelmének különösen nagy jelentősége van. Ha egy nagyenergiájú részecske az állapotregiszter valamelyik flip-flopját átbillenti,[88] akkor az előálló új bitsorozat jó eséllyel érvénytelen állapotot jelent, az állapotgép tehát holtpontra jut, amit mindenképpen meg szeretnénk akadályozni. A MiniCore processzor esetén az első teendőnk, hogy magát a HDL modellt felkészítjük erre a helyzetre, vagyis az állapotgépeket (a műveletvégző egységet vezérlő állapotgépet, valamint az osztó áramkör vezérlő állapotgépét) megvalósító `case` szerkezetek `others`/`default` ágaiban definiáljuk az ekkor elvárt viselkedést. Erre a 8-27. ábrán láthattunk egy lehetséges megoldást, amelyben az érvénytelen állapotot az állapotgép képes detektálni, jelezni (`fsm_error`), sőt, egy külső vezérlőjel (`recover_fsm`) segítségével az állapotgép vissza is juttatható a kiindulási állapotba anélkül, hogy globális resetet alkalmaznánk. A MiniCore processzor esetén ennél egy kicsit egyszerűbb megoldást valósítunk meg, amely a hibát csak detektálja és jelzi, de az alapállapotba való visszaállításra nem vezet be külön jelet, feltételezve azt, hogy a magasabb hierarchiaszinten működő hibadetektáló rendszer a globális reset jel segítségével indítja újra a processzort. Mindkét állapotgép

[88] A soft register-TMR-nek köszönhetően az állapotregiszter minden bitjét fizikailag három flip-flop képviseli, vagyis az átbillenés csak akkor következhet be, ha legalább két flip-flop egyszerre íródik át. Ennek valószínűsége igen csekély ugyan, de nem nulla, ezért a megbízhatóság növelése céljából az RTL modell szintjén is felkészülünk erre a szituációra.

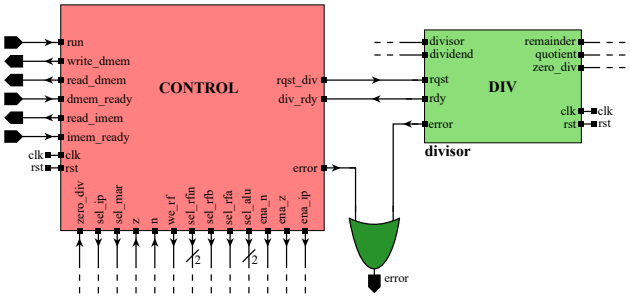
VHDL

```
...
when s_error  =>  error <= '1';
when others   =>  error <= '1';
                  state <= s_error;
```

Verilog

```
...
s_error:  error <= 1'b1;
default:  begin
            error <= 1'b1;
            state <= s_error;
          end
```

Áramköri struktúra



9-10. ábra. *Az állapotgépek érvénytelen állapotait jelző hibajelek és azok közösítése*

HDL modelljét kiegészítjük a 9-10. ábrán látható kódrészlettel, majd az így előálló két hibajelet egy OR kapuval közösítjük.

A 8.2.1. pontban részleteztük, hogy a szintézis szoftverek az állapotgépek viselkedésée nek leírására használt `case` szerkezetek `others/default` ágába foglalt funkciókat megvalósító erőforrásokat kioptimalizálják. A 8-28. ábrán azt is láthattuk, hogy ezt az optimalizációt miként lehet letiltani a `syn_preserve` attribútum/pragma segítségével. Természetesen ezt a MiniCore processzor esetén is meg kell tennünk. A HDL modell megfelelő módosítása és a szükséges attribútumok/pragmák elhelyezése után újrafuttatott szintézis és időzítésvizsgálat eredményeit a 9-9. és 9-10. táblázat foglalja össze.

FPGA erőforrás	Felhasznált mennyiség	Rendelkezésre álló mennyiség	Kihasználtság
VersaTile	11436	13824	82,73 %
blokk RAM modul	0	24	0 %

9-9. táblázat. *Az áramkör erőforrásigénye #5*

Órajelfeltétel		Kritikus út			
		kibocsátó flip-flop	fogadó flip-flop	t_D	maradékidő (slack)
f_{clk}	t_{Dmax}	sel_alu/ tmr2[1]	mx_rfdin/ tmr3[31]	48,454 ns	1,546 ns
20 MHz	50 ns				

9-10. táblázat. *A statikus időzítésvizsgálat eredménye #5*

A MiniCore processzor mindkét állapotgépe esetén one-hot állapotkódolást használtunk, mivel ezzel a megoldással az állapotdekóderek mérete a lehető legkisebb lehetett. Ahogy arra a 8.2.1.2. pontban is kitértünk, one-hot kódolású állapotregiszterek esetén ane nak megállapítása, hogy az állapotregiszter egy bizonyos értéket vett fel, nem igényel kombinációs hálózatot; csak meg kell vizsgálnunk az állapotregiszternek azt a bitjét, amelyik az adott állapothoz tartozó állapotkódban magas értékű. Ha ez a bit magas, akkor az állapotgép a keresett állapotban van. Ez a technika azonban nem érvényes akkor, ha az érvénytelen állapotokat is detektálnunk kell; ha egy SEU folytán az állapotregiszter két bitje is logikai magas értékű, akkor az állapotdekódereket megspóroló megoldásban a kimeneteink úgy viselkedhetnének, mintha az állapotgép egyszerre kétféle állapotban lenne, ami nyilván funkcionális hibákhoz vezethet. Ahhoz, hogy egyértelműen meg tudjuk állapítani, hogy melyik érvényes állapotban vagyunk, vagy esetleg érvénytelen kombináció van az állapotregiszterben, ún. teljes dekódereket kell használnunk, amelyek az állapotregiszter minden egyes bitjének értékét felhasználják egy konkrét állapot beazonosítására. A teljes dekóderek szintézise a szintézis szoftver `syn_safe_case` parancssori argumentumának beállításával lehetséges. A beállítás módosításának hatását a 9-11. és 9-12. táblázat foglalja össze.

FPGA erőforrás	Felhasznált mennyiség	Rendelkezésre álló mennyiség	Kihasználtság
VersaTile	11566	13824	83,67 %
blokk RAM modul	0	24	0 %

9-11. táblázat. *A áramkör erőforrásigénye #6*

Órajelfeltétel		Kritikus út			
		kibocsátó flip-flop	fogadó flip-flop	t_D	maradékidő (slack)
f_{clk}	t_{Dmax}	sel_alu/ tmr2[1]	mx_rfdin/ tmr3[12]	50,027 ns	**-0,027 ns**
20 MHz	50 ns				

9-12. táblázat. *A statikus időzítésvizsgálat eredménye #6*

9.1.4.4 Hamming-2 állapotkódolás

A one-hot állapotkódolás legnagyobb előnyét tehát egyáltalán nem tudjuk kihasználni olyan alkalmazásokban, ahol képesnek kell lennünk az érvénytelen állapotok detektálására. Azt mondhatnánk, hogy ennek ellenére mégiscsak érdemes one-hot kódolást használnunk, hiszen így az állapotregiszter egybites hibái egyértelműen detektálhatók. Ez a képessége azonban a Hamming-2 kódolásnak is megvan, ráadásul az állapotregiszter flip-flopjainak a száma lényegesen kisebb lehet, mint one-hot kódolás esetén. Ezért aztán az állapotkódolást mindenképpen érdemes megváltoztatnunk. A Hamming-2 állapotkódolást a szintézis szoftverek többsége nem ismeri, ezért ezt manuálisan kell megvalósítanunk úgy, hogy az állapotregisztert nem felsorolt típusú jellel, hanem egy egyszerű logikai vektorral írjuk le,

amelynek lehetséges értékeit egy-egy konstanssal magunk definiáljuk. A 9-11. ábra az osztó áramkör példáján mutatja be a manuális, Hamming-2 állapotkódolást.

VHDL

```
constant s0:        std_logic_vector (3 downto 0) := "0001";
constant s1:        std_logic_vector (3 downto 0) := "0010";
constant s2:        std_logic_vector (3 downto 0) := "0100";
constant s3:        std_logic_vector (3 downto 0) := "0111";
constant hold_rdy:  std_logic_vector (3 downto 0) := "1000";
constant t1:        std_logic_vector (3 downto 0) := "1011";
constant s_error:   std_logic_vector (3 downto 0) := "1101";

signal   state:     std_logic_vector (3 downto 0);
```

Verilog

```
parameter   s0:       4'b0001;
parameter   s1:       4'b0010;
parameter   s2:       4'b0100;
parameter   s3:       4'b0111;
parameter   hold_rdy: 4'b1000;
parameter   t1:       4'b1011;
parameter   s_error:  4'b1101;

logic [3:0] state;
```

9-11. ábra. *Hamming-2 állapotkódolás manuális megvalósítása az osztó áramkör HDL modelljében*

A módosított HDL modellen lefuttatott szintézis és időzítésvizsgálat eredményeit a 9-13. és 9-14. táblázat foglalja össze.

FPGA erőforrás	Felhasznált mennyiség	Rendelkezésre álló mennyiség	Kihasználtság
VersaTile	11418	13824	82,6 %
blokk RAM modul	0	24	0 %

9-13. táblázat. *Az áramkör erőforrásigénye #7*

Órajelfeltétel		Kritikus út			
		kibocsátó flip-flop	fogadó flip-flop	t_D	maradékidő (slack)
f_{clk}	t_{Dmax}	rf/doutb/ tmr2[0]	mx_rfdin[25]	49,709 ns	**0,291 ns**
20 MHz	50 ns				

9-14. táblázat. *A statikus időzítésvizsgálat eredménye #7*

Az eredmények azt mutatják, hogy áramkörünk immár megfelel a követelményeknek, az elvárható megbízhatóságnövelő technikák alkalmazása mellett is teljesíti az időzítési feltételeket.[89]



[89] Az FPGA adatlapja szerint a gyártó garantálja, hogy 30 krad TID-ig az eszközben megvalósított lokális adatutak késleltetése legfeljebb 10%-ot nő. Arról nincs pontos információnk, hogy az előírt 20 krad esetén mi várható, ezért az ismert, 30 krad-ra vonatkozó adat alapján úgy számolhatnánk, hogy a sugárzás hatását figyelmen kívül hagyva a maradékidőnek legalább 5 ns-nak kellene lennie ahhoz, hogy 30 krad dózis után is teljesüljenek az időzítési feltételek. Szerencsére ezt a számítást nem ilyen módon kell elvégeznünk, az eszköz sugárzás hatására történő degradációját ui. a statikus időzítésanalízis szoftver már figyelembe vette, a táblázatban látható legkisebb maradékidő tehát a 30 krad dózis okozta jelút-degradáció után érvényes.

Irodalomjegyzék

[1] Ashended, Peter J.: *Digital Design, An Embedded System Approach Using Verilog.* 2008, Elsevier

[2] Ashended, Peter J.: *Digital Design, An Embedded System Approach Using VHDL.* 2008, Elsevier

[3] Scheffer, Louis – Lavagno, Luciano – Martin, Grant (eds.): *EDA for IC Implementation, Circuit Design, and Process Technology.* 2006, CRC Press

[4] Kaeslin, Hubert: *Digital Integrated Circuit Design, From VLSI Architectures to CMOS Fabrication.* 2008, Cambridge University Press

[5] Keating, Michael – Bricaud, Pierre: *Reuse Methodology Manual for System-on-a-Chip Designs.* 2002, Kluwer Academic Publishers

[6] Schaumont, Patrick R.: *A Practical Introduction to Hardware/Software Codesign.* 2010, Springer

[7] Rigo, Sandro – Azevedo, Rodolfo – Santos, Luiz (eds.): *Electronic System Level Design, An Open-Source Approach.* 2011, Springer

[8] Chinnery, David – Keutzer, Kurt: *Closing the Gap Between ASIC & Custom.* 2002, Kluwer Academic Publishers

[9] Hosszú Gábor – Keresztes Péter: *VHDL-alapú rendszertervezés.* 2012, SZAK Kiadó Kft.

[10] Kilts, Steve: *Advanced FPGA Design, Architecture, Implementation, and Optimization.* 2007, John Wiley & Sons.

[11] Harris, David Money – Harris, Sarah L.: *Digital Design and Computer Architecture (2nd ed.).* 2013, Elsevier

[12] Davis, Justin – Reese, Robert: *Finite State Machine Datapath Design, Optimization, and Implementation.* 2008, Morgan & Claypool

[13] Smith, Gina R.: *FPGAs 101, Everything you need to know to get started.* 2010, Elsevier

[14] Smith, Douglas J: *HDL Chip Design, A practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or Verilog.* 1996, Doone Publications

[15] Churiwala, Sanjay – Garg, Sapan: *Principles of VLSI RTL Design, A Practical Guide.* 2011, Springer

[16] Sutherland, Stuart – Davidmann, Simon – Flake, Peter: *SystemVerilog For Design, A Guide to Using SystemVerilog for Hardware Design and Modeling (2nd ed.).* 2006, Springer

[17] Ashended, Peter J. – Lewis, Jim: *The Designer's Guide to VHDL (3rd ed.).* 2008, Elsevier

[18] Kafig, William: *VHDL 101, Everything you need to know to get started.* 2011, Elsevier

[19] Navabi, Zainalabedin: *VHDL, Modular Design and Synthesis of Cores and Systems (3rd ed.).* 2007, McGraw-Hill

[20] Bergeron, Janick: *Writing Testbenches, Functional Verification of HDL Models.* 2002, Kluwer Academic Publishers

[21] Munden, Richard: ASIC and FPGA Verification, A Guide to Component Modeling. 2005, Elsevier

[22] Spear, Chris: *SystemVerilog for Verification, A Guide to Learning the Testbench Language Features (2nd ed.).* 2008, Springer

[23] Ashended, Peter J. – Lewis, Jim: *VHDL-2008, Just the New Stuff.* 2008, Elsevier
[24] Gangadharan, Sridhar – Churiwala, Sanjay: *Constraining Designs for Synthesis and Timing Analysis, A Practical Guide to Synopsys Design Constraints (SDC).* 2013, Springer
[25] Bhasker, J. – Chadna, Rakesh: *Static Timing Analysis for Nanometer Designs, A Practical Approach.* 2009, Springer
[26] Parvez, Husain – Mehrez, Habib: *Application-Specific Mesh-based Heterogeneous FPGA Architectures.* 2011, Springer
[27] Xanthopoulos, Thucydides (ed.): *Clocking in Modern VLSI Systems.* 2009, Springer
[28] Maxfield, Clive: *FPGAs, World Class Designs.* 2009, Elsevier
[29] Farooq, Umer – Marrakchi, Zied – Mehrez, Habib: *Tree-Based Heterogeneous FPGA Architectures, Application Specific Exploration and Optimization.* 2012, Springer
[30] Sherwani, Naveed: *Algorithms for VLSI Physical Design Automation (3rd ed.).* 1999, Kluwer Academic Publishers
[31] Reis, André Inácio – Drechsler, Rolf (ed.): *Advanced Logic Synthesis.* 2018, Springer
[32] Lim, Sung Kyu: *Practical Problems in VLSI Physical Design Automation.* 2008, Springer
[33] Khatri, Sunil P. – Gulati, Kanupriya (ed.): *Advanced Techniques in Logic Synthesis, Optimizations and Applications.* 2011, Springer

Rövidítésjegyzék

AD/DA	Analog-Digital/Digital-Analog (converter)
ALU	Arithmetic/Logic Unit
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
ATPG	Automatic Test Pattern Generation
AXI	Advanced eXtensible Interface
BFM	Bus Functional Model
BFSK	Binary Frequency Shift Keying
BiCMOS	Bipolar CMOS
BLE	Basic Logic Element
CAD	Computer-Aided Design
CDC	Clock Domain Crossing
CDFG	Control/DataFlow Graph
CLB	Configurable Logic Block
CME	Coronal Mass Ejection
CMOS	Complementary MOS
CPLD	Complex Programmable Logic Device
DDR	Double Data Rate
DFT	Design For Testability
DLL	Delay-Locked Loop
DMA	Direct Memory Access
DoD	Department of Defense
DRC	Design Rule Check
DSP	Digital Signal Processing
DUT	Design Under Test
DUV	Design Under Verification
EDAC	Error Detection And Correction
EDIF	Electronic Design Interchange Format
eRM	e Reuse Methodology
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
HK DAQ	Housekeeping Data Acquisition
HKMG	High-K Metal Gate
HLS	High Level Synthesis
HVL	Hardware Verification Language
I²C	Inter-Integrated Circuit vagy Inter-IC Communication
IEEE	International Institute of Electrical and Electronics Engineers
IOB	I/O block

IP	Intellectual Property
LAB	Logic Array Block
LUT	Look-Up Table
LVS	Layout Vs Schematic
MAC	Multiply & Acumulate
MISP	Minimal Instruction Set Processor
MMIO	Memory-Mapped I/O
MOS	Metal-Oxide Semiconductor
OTP	One-Time Programmable
OVM	Open Verification Methodology
PCB	Printed Circuit Board
PDC	Physical Design Constraint
PLA	Programmable Logic Array
PLD	Programmable Logic Device
PLL	Phase-Locked Loop
QPSK	Quadrature Phase-Shift Keying
RF	Radio Frequency
RTL	Register-Transfer Level
SCL	Source-Coupled Logic
SDC	Synopsys Design Constraint
SDF	Standard Delay Format
SDIO	Secure Digital Input Output
SECDED	Single-Error Correction, Double-Error Detection
SEL	Single-Event Latch-up
SET	Single-Event Transient
SEU	Single-Event Upset
SoC	System on Chip
SoPC	System on a Programmable Chip
SPI	Serial Peripheral Interface
STA	Static Timing Analysis
TID	Total Ionizing Dose
TLM	Transaction Level Modeling
TMR	Triple-Modular Redundancy
UART	Universal Asynchronous Receiver Transmitter
UCF	User Constraint File
USART	Universal Synchronous/Asynchronous Receiver Transmitter
UVM	Unified Verification Methodology
VC	Verification Component
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Tárgymutató

A,Á

algoritmusszint 15
állapotátmenet-lefedés 176
állapotgép 104
állapotgép-lefedés 176
állapotkódolás 267
állapotlefedés 176
állapotregiszter 106
állapottér (DUV) 172
always-blokk 72
anti-fuse 240, 244
application-specific instruction-set processsor 49
áramköri szint 15
aszinkron dual-clock FIFO szinkronizáló 221
aszinkron órajeltartományok 210
attribútum (VHDL) 269
Automatic Test Pattern Generation 23
automatizált áramkörszintézis 261

B

beállítási idő 188
beállítási maradékidő 195
beállíthatóság 179
befogó szinkronizáló 215
bemeneti késleltetés 201
BFM modell 167
black-box verifikáció 179
blokkoló értékadás 79
blokk-RAM 245
bondig pad 239
branch coverage 176
bus functional model 163

C

capture synchronizer 215
cellasor 235
check 171
ciklikus példányosítás 116
ciklus 72, 78
clock constraint 201
clock domain crossing 211
clock insertion delay 197
clock skew 197
clock-to-output delay 187
code coverage 176
complex programmable logic device 23
condition coverage 176
constant folding 265
constant propagation 273
constrained random test 175
control unit 126
control/dataflow graph 265
controllability 179
corner case 175
CORNER cella 238
coronal mass ejection 300

D

datapath 126
deaktiválási idő 188, 227
debouncer 119
DECAP cella 235, 282
decision coverage 176
dekompozíció 276
delta-késleltetés 77
design entry 64
design flow 16, 17
design rule check 289

design under verification 156
de-skew áramkör 237
Disjoint (kapcsolómátrix) 243
double-flop szinkronizáló 211, 292
DSP modul 245

E,É

egyed-architektúra pár 66
egymásba vezetés 190
elaboration 261
elágazás-lefedés 176
elcsúszás-kompenzálás 247
éldetektor 90
electronic design interchange format 262
elhelyezés 281, 286
eljárás-alapú BFM 167
előzetes szintézis 21
élváltás-lefedés 176
ENDCAP cella 235, 282
erőforrás-allokáció 58
erőforrás-megosztás 265
error detection and correction 303
érzékenységi lista 72
expansion 265
expression coverage 176

F

factorization 276
false path 206
feléledési idő 188, 226
felhasználói követelmények 18
feltételes példányosítás 114
feltétel-lefedés 176
field-programmable gate array 23, 239, 263
FILLER cella 235, 282
fizikai órajelforrás 197
fizikai szintézis 22, 262, 279, 285
flattening 274
floor-plan 280
folyamat 72
folytonos értékadás 67
földgyűrű 236, 281
frekvenciaszintézis 247
FSM coverage 176
FSM dead-lock 271
full-custom ASIC 232
functional coverage 179
funkcionális lefedés 179
funkcionális verifikáció 21, 155
futásidejű paraméter 111

G

Gajski–Kuhn-diagram 14
generic folding 112, 265
generikus szintézis 22, 266
generikusság 111
gerjesztési minta 156, 172
globális huzalozási hálózat 246
globális jel 87, 237
glue-logic 240
gray (állapotkódolás) 268
gray szinkronizáló 216
Gray-enkóder/dekóder 85, 217
grey-box verifikáció 180

H

Hamming-2 (állapotkódolás) 268, 306
Hamming-3 (állapotkódolás) 268
hand-shake pulse synchronizer 215
hardver/szoftver szétválasztás 30
hardverleíró nyelv 57
hardver-verifikációs nyelv 182
HDL szimulátor 156
hold slack 196
hold-time 188
hold-time violation 197, 200
holtpont (FSM) 271
hullámforma-megjelenítő 158
huzalozás 282, 287
huzalozási csatorna 233

I,Í

időzítési kényszer....193, 293
időzítési kivétel....204
időzítési peremfeltétel....193
időzítési szimuláció....24, 288
időzítésvezérelt szintézis....264
impedanciaillesztett órajelfa....238
informális specifikáció....17
initial-blokk....157
input delay constraint....201
integrációs tesztelés....181
IP-alapú tervezés....16
irányított teszt....173

J

jitter-kompenzálás....247

K

kapcsolómátrix....243
kapuszint....15
kaszkád előosztó....124
kimeneti buffer (állapotgép)....108
kimeneti késleltetés....187, 202
kimeneti logika....108
kódlefedés....176
kombinációs folyamat....72
konfigurálható I/O elem....242
konfigurálható logikai elem....241
konfigurálható összeköttetéshálózat....242
konkurens utasításblokk....76
következőállapot-logika....106
kritikus út....195
kulcsállapot....175

L

latch....91, 188
layout vs schematic....290
lineáris tesztkörnyezet....158
local datapath....194
logikai szimuláció....64
logikai szint....15
logikai szintézis....22, 64, 262, 263
lokális adatút....194
look-up table....241
loop unrolling....265

M

master-slave D flip-flop....188
megfigyelhetőség....179
memória....99
metastabilitás....185, 292
modul-alapú BFM....167
moduldeklaráció....66
multicycle path....206
multipass placement....287
multiplexer....91
multiply and accumulate....98
műveletvégző egység....126

N

nem-blokkoló értékadás....77
netlista....262
név szerinti porthozzárendelés....70

O,Ó

observability....179
one-hot (állapotkódolás)....268
one-shot (állapotgép)....107
órajel....180
órajel-becsatolási késleltetés....197
órajel-domén....210
órajelelcsúszás....89, 197
órajel-előosztó....122
órajelfa....89, 237
órajelfeltétel....194
órajeltartomány....209
órajeltartományok közötti adatátvitel....210
output delay constraint....202

Ö,Ő

önellenőrző verifikációs környezet....47, 171

P

PAD cella....235, 284
PAD-gyűrű....239, 284
példányosítás....70
pergésmentesítés....119
placement....281
portlista....67, 156
post-layout időzítési szimuláció....289
post-mapping időzítési szimuláció....289
post-place&route időzítési szimuláció....289
post-szintézis időzítési szimuláció....288
pozíció szerinti porthozzárendelés....70
pragma (Verilog)....269
próbapad....156
process....72
programmable logic device....231, 263
propagation delay....187

R

random teszt....175
RC terhelés....185
referenciamodell....163, 170
register duplication....271
register pruning....270
regiszter....87
regiszter-optimalizáció....269
regisztertömb....99
regiszter-transzfer szint....15
regressziós tesztelés....171
rendszerszint....15
rendszertervezés....20, 29
reset delay....185
reset fan-out kiegyenlítés....229
reset késleltetés....187
reset recovery time....188, 226
reset removal time....188, 227
reset-szinkronizáció....226
resolution time....190
resource-sharing....265
router....237
routing....282
RTL kódolási szabályok....65
RTL szintézis....22, 64
RTL tervezés....21, 57

S

scan-path....23, 278
scan-path insertion....278
SDF....288
sea-of-gates....239
sea-of-transistors....239
semi-custom ASIC....232
setup slack....195
setup-time....188
setup-time violation....196, 199
single error correction, double error detection....303
single-event latchup....301
single-event transient....301
single-event upset....301
soft register-TMR....272, 302
SPARE cella....235, 282
specifikáció....17, 155
stabilizálódási idő....190
standard cella....232
standard cellás ASIC....232
state coverage....176
statement coverage....176
static timing analysis....289
statikus időzítésvizsgálat....23, 289
statikus jel....212
stripe....236, 281
synopsys design constraint....207
system-on-a-programmable chip....246
system-on-chip....246

Sz

szekvenciális (állapotkódolás) 267
szekvenciális utasításblokk 76
szimulációs szemantika 63
szinkron folyamat 74
szinkron órajeltartományok 210
szintaxisfa 265
szintézis 21, 22, 249
szintézis szemantika 63
szintézis-paraméter 111
szűkített véletlen teszt 175

T

tápgyűrű 236, 281
tartási idő 188
tartási maradékidő 196
technológiafüggetlenség 64, 89, 111
technológiai leképezés 22, 277
technology mapping 277
terjedési késleltetés 187
testbench 156
teszteset 156, 159
tesztkörnyezet 45, 55, 156
tesztvezérlő 164
TIE cella 235
timing closure 193, 294
timing constraint 193
timing exception 204
timing-driven synthesis 263
toggle coverage 176
toggle pulse synchronizer 213
toplevel integráció 22, 181
toplevel verifikáció 181
total ionizing dose 300
többciklusú adatút 206
transition coverage 176
translation 261
transmission-gate 188
tranzisztorszint 15
triple modular redundancy 272

U,Ú

újrafelhasználhatóság 39, 111
újrakonfigurálható hardver 23, 233
utasítás-lefedés 176

Ü,Ű

ütemezés 58

V

validációs eljárás 18
változó-értékadás 80
véletlenszerű teszt 173
verifikációs komponens 164
verifikációs környezet 156, 164
vezérlési/adatfolyam gráf 265
vezérlő egység 126
virtuális órajelforrás 197
viselkedési modellezés 20, 37, 163
visszacsatolatlan impulzusszinkronizáló 213
visszacsatolatlan vektor-szinkronizáló 219
visszacsatolt impulzus-szinkronizáló 215
visszacsatolt vektorszinkronizáló 221
vízesésmodell 26

W

watchdog 116
white-box verifikáció 180
Wilton (kapcsolómátrix) 243
wire-load modell 264

Z

zero-delay szimuláció 185

A kiadásért felel *Gyenes Ádám,* a kiadó vezetője.

A kiadó kötetei megrendelhetők,
illetve kedvezménnyel megvásárolhatók:
L'Harmattan Könyvesbolt
1053 Budapest, Kossuth L. u. 14–16.
Tel.: (36) 1-267-5979
www.harmattan.hu

Olvasószerkesztő: *Vigyázó Ágnes*
Nyomdai előkészítés: *Gembela Zsolt*