1 Hardver leíró nyelvek – VHDL

Az eddigiekben bemutatott digitális rendszerek tervezését ebben a fejezetben egy más nézőpontból világítjuk meg. A fejezet bemutatja a digitális rendszerek tervezését hardver leíró nyelvek segítségével.

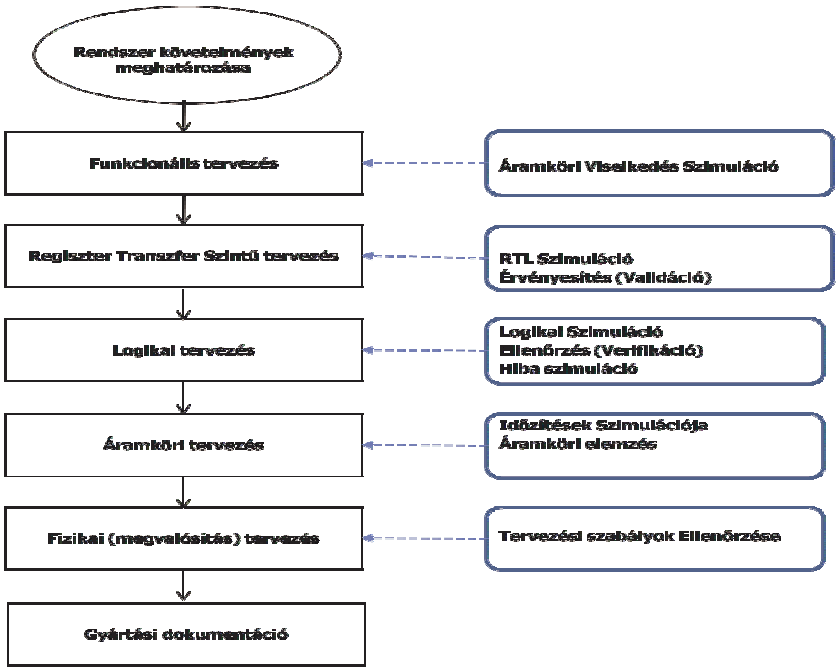
1.1 Digitális rendszerek tervezése

A digitális rendszerek tervezése az a folyamat, amely kezdődik a rendszerrel szembeni követelmények megállapításával és több lépésen keresztül eljut a rendszer fizikai megvalósításáig. Az integrált áramköri technológia fejlődése révén egyre bonyolultabb digitális áramkörök tervezése valósítható mg. A bonyolult tervek kezelése két irányzatot eredményezett a tervezési módszerekben:

- az áramkörök leírása viselkedésük alapján;
- a számítógép alapú tervezésautomatizálás.

A tervezési folyamatban a szimuláció, mint a szintézis egymást kiegészítő tevékenységek. Egy berendezésorientált áramkör tervezésé esetében például a szimuláció segítségével kiküszöbölhetők a tervezési hibák és így a gyártási költségek (integrált áramköri maszkok előállítása) jelentősen csökkenthetők. Az 1. ábra a tervezés és szimuláció közötti összehangolást mutatja.

Az első lépés a rendszerrel szembeni követelmények meghatározása. Ezek a követelmények meghatározzák a működési sebességet, a késleltetési időket, a csatlakozási pontokat (interfészeket), a disszipált teljesítményt, és egyéb fizikai paramétereket.



1. ábra Digitális rendszerek tervezésének folyamata

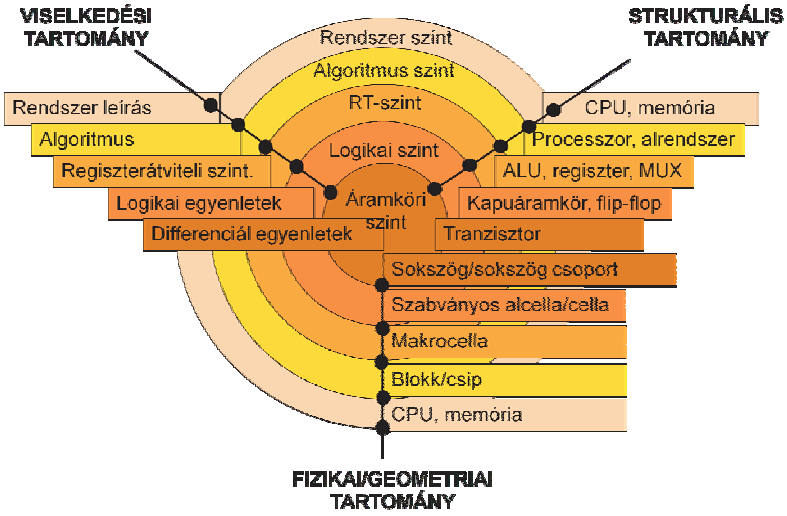
A fent említett követelmények alapján a funkcionális tervezés során megtervezhető a rendszer és ezen a szinten használt szimulációval ellenőrizni lehet a funkcionális követelményeket. Ezt a tervet a regiszter átviteli szinten finomítani lehet. Ezen a szintem a feladatokat átveszik a regiszterek, memóriák, aritmetikai egységek, állapotgépek.

A logikai tervezés megvalósítja a regiszter-átviteli szinten meghatározott elemek megvalósítását. A hibák szimulációja modellezi a gyártáskor várható hibákat és a környezeti

hatások által gerjesztett hibákat is. Végül a fizikai megvalósítással jön létre a digitális áramkör, ami lehet akár egy a gyártásra kész integrált áramkör.

1.2 Hardver absztrakciós szintek

A tervezési hierarchia minden szintjén áramköri elemek használatával írjuk le a digitális rendszer tervet. A komplex digitális rendszerek különböző módszerekkel írhatók le. Azonban ezen leírási módszerek egymással kompatibilisek. Az áramköröket általában három tartományban tudjuk leírni: viselkedési strukturális és fizikai tartomány. Ezen leírási módszereket a Y diagram fejezi ki [1.]



2. ábra Az Y diagram

Az Y diagram három tengelye (az „Y" ágai), különböző leírási módszerek. Az egyes ágakon a kívülről befelé történő leírást finomításnak nevezzük, míg a bentről kifele történő leírást absztrakciónak nevezzük.

1.3 Digitális áramkörök szöveges leírása

Az integrál áramkörök bonyolultságának növekedésével és a technológia fejlődésével egyre inkább látszott az, hogy a megszokott kapcsolási rajz típusú tervezés egyre áttekinthetetlenebb és nehezen kezelhető áramköri terveket eredményez.
Az áramköri technológia pedig lehetővé tette az úgynevezett programozható logikai áramkörök gyártását. A programozható logikai áramkörök a felhasználó szempontjából szabványos integrált áramkörök, amelyek programozható funkciókkal rendelkeznek. Azaz a felhasználó nem tudja módosítani belső felépítésüket, de a digitális áramköri függvényt a felhasználó alakítja ki.
Ezen programozható logikai áramkörökkel való tervezéshez azonban már szükség volt egy olyan tervezési eszközre, amelynek segítségével a felhasználó könnyen leírja a célfüggvényt. Először ez a leírási módszer az áramköri technológiából adódó biztosítéktérkép állomány leírásában merült ki. A biztosítéktérkép pedig az áramkör belsejében lévő kapcsolatokat határozta meg, hasonlatosan a tervezésben megszokott huzalozási listához. Ez a módszer nehézkes és bonyolult volt ezért aztán szükségessé vált egy olyan általános hardvermodellező nyelv kidolgozása, amely a digitális áramkörök különböző elvonatkoztatási szinteken történő, egységes leírására alkalmas a felhasználható legkülönbözőbb elektronikai rendszerek tervezésére, különböző gyártóktól származó és különböző technológiákkal megvalósított

programozható áramkörök használatával. Ilyen programozható áramköröket jelenleg az Altera, a Xilinx és a Lattice, stb. cégek gyártanak.
A technológia fejlődésével több hardver leíró nyelv közül választhatott a felhasználó: PALASM, ABEL, VHDL, VERILOG, System C, stb. Napjainkban a legelterjedtebb a VHDL, a System Verilog és a SystemC. A továbbiakban a legjelentősebbeket ismertetjük röviden és bővebben a VHDL nyelv leírásával foglalkozunk.

1.3.1 VHDL Hardverleíró nyelv

Mit jelent a VHDL kettős betűszó? A rövidítés jelentése: VHSIC-HDL– (**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit **H**ardware **D**escription **L**anguage) - azaz nagyon (V) nagy (H) sebességű (S) integrált (I) áramkörök (C) hardver (H) leíró (D) nyelve (L).
A VHDL nyelvet eredetileg az amerikai Védelmi Minisztérium (DoD – Department of Defence) megbízásából fejlesztették ki. A fejlesztés egy olyan hardverleíró programozási nyelvet eredményezett, amely messzemenően megfelelt a tervezés szempontjainak. Egy olyan programozási nyelv jött létre a VHDL nyelvvel, amely önmagát dokumentó, strukturált és érthető. A forráskód egyben egyfajta specifikációs dokumentáció is. A hardverleíró nyelvek legfontosabb eleme pedig a párhuzamosságok, konkurens folyamatok kezelése. Ugyanakkor képesek sorrendi hálózatok komplex és kompakt modellezésére.
A VHDL szabványosítását az IEEE (Institute of Electrical and Electronics Engineers – Villamosmérnökök Nemzetközi Szervezete) amerikai szervezete 1987-ben végezte el. A szabvány első hivatalos frissítése 1993-ban történt meg. A VHDL nyelv kiterjesztése az analóg és vegyes jelű rendszerek modellezhetőségével új fejezetet nyitott az elektronikus rendszerek modellezésében VHDL-AMS (**a**nalogue **m**ixed **s**ignal), amely a VHDL szabvány egyik kiterjesztése. A kiterjesztés csupán a rendszer szimulációra vonatkozik, hiszen az analóg áramkörök szintézise egy igen komplex és sokparaméteres probléma megoldását jelenti, amely probléma jelenleg nem megoldott.

1.3.2 Verilog hardverleíró nyelv

A Verilog nyelv tervezői egy olyan hardverleíró nyelvet akartak létrehozni, amely azonos felépítésű, mint a C programozási nyelv, amely már széleskörűen elterjedt az mikroprocesszoros rendszerek fejlesztésében. A Verilog érzékeny a kisbetű-nagybetű típusra, hasonlóan az ANSI C/C++ nyelvhez rendelkezik egy előfeldolgozó egységgel (preprocessor), vezérlés átadó kulcsszavakkal (if/else, for, while, case, etc.), kompatibilis operátor előzménnyel (precedence). Szintaktikai különbségek a változók típusmegadásában (declaration), a folyamattömbök elválasztásában és egyebekben mutatkoznak.
A Verilog terv több hierarchikus modulból áll. A modulok tartalmazzák a tervezési hierarchiát. A modulok egymásközti kapcsolatát a kimenetek, bemenetek és kétirányú kapcsolatok (port) határozzák meg. Egy modul belső felépítésében szintén tartalmazhat belső változókat/jeleket (wire, reg, integer, etc.), sorrendi állapottömböket, modulokat. A Verilog nyelv egy egyidejű és sorrendi adatfolyam feldolgozó nyelv.

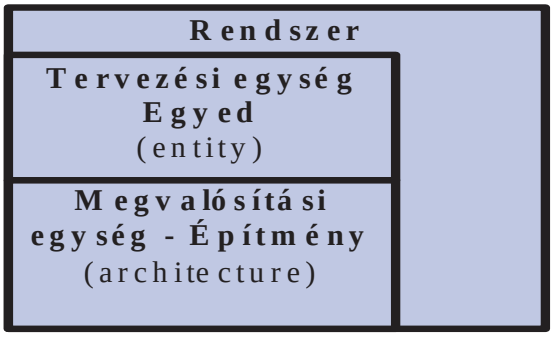
1.3.3 System C hadverleíró nyelv

A SystemC® a szabványos ANSI C++ osztály (class) könyvtár kiterjesztése a hardvertervezés számára. A SystemC célja egy C++ alapú tervezési szolgáltatás olyan hibrid rendszerek tervezéséhez, amelyek mind hardver mind szoftver alkotóelemekkel rendelkeznek.
A SystemC nyelvet az IEEE 1666-2005-ös szabványa írja le. A SystemC alkalmazás során felhasználhatjuk a C++ lehetőségeit, azzal a megkötéssel, hogy ne térjünk el a szabványban meghatározottaktól.

1.4 VHDL alapok

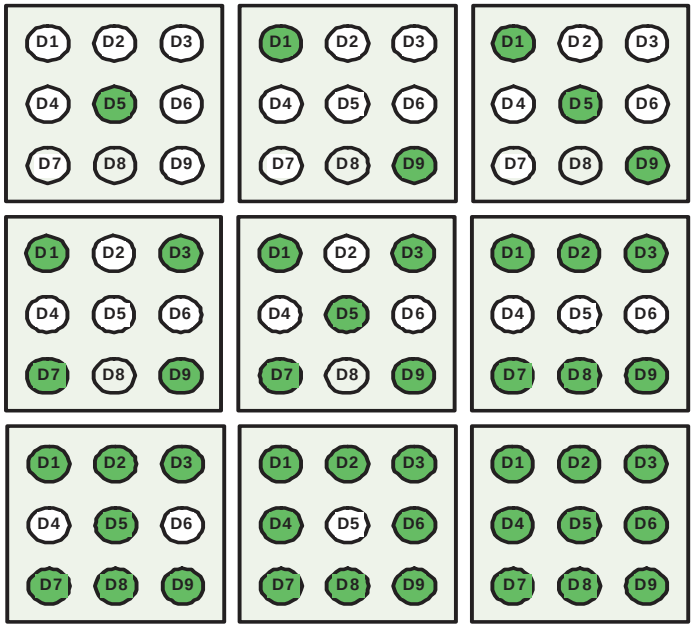
Elemezzük közelebbről a strukturális és viselkedési rendszer leírást. Egy digitális rendszer alapvetően ***jelek (signal)*** feldolgozásával foglalkozik. A jelek bináris értékeket vehetnek fel (0 vagy 1). A digitális rendszerek alkotóelemei az alkatrészek (component), úgy, mint logikai kapuk, flip-flopok, számlálók és processzorok, stb. Az alkatrészek közötti összeköttetéseket a vezetékek (wire) valósítják meg. A bemenő jeleket (input signal) ugyancsak az alkatrészek alakítják át ***kimenő jelekké (output signal)***, ugyanakkor vannak ***kétirányú*** vagy ***ki-bemeneti jelek (input/output signal)***.

A VHDL program felépítését tekintve ***tervezési egységekből*** áll. A megvalósítható (szintetizálható) VHDL programnak két tervezési egységet kell tartalmazna: egy ***tervezési egységet (egyedet - entity)*** és egy az egyedhez hozzárendelt ***megvalósítási egységet (építményt - architecture)*** (1. ábra).



1. ábra VHDL modell

A VHDL program felépítését legjobban egy példán keresztül tudjuk szemléltetni. Ehhez vegyük a ?? fejezetben említett „dominó” kód megvalósítást (2. ábra- 3. ábra és 4. ábra és 1-2 egyenletek).



2. ábra „Dominó” kód kijelzése

mi	D	C	B	A	D9	D8	D7	D6	D5	D4	D3	D2	D1
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	1	0	0	0	0
2	0	0	1	0	1	0	0	0	0	0	0	0	1
3	0	0	1	1	1	0	0	0	1	0	0	0	1
4	0	1	0	0	1	0	1	0	0	0	1	0	1
5	0	1	0	1	1	0	1	0	1	0	1	0	1
6	0	1	1	0	1	1	1	0	0	0	1	1	1
7	0	1	1	1	1	1	1	0	1	0	1	1	1
8	1	0	0	0	1	1	1	1	0	1	1	1	1
9	1	0	0	1	1	1	1	1	1	1	1	1	1
A	1	0	1	0	X	X	X	X	X	X	X	X	X
B	1	0	1	1	X	X	X	X	X	X	X	X	X
C	1	1	0	0	X	X	X	X	X	X	X	X	X
D	1	1	0	1	X	X	X	X	X	X	X	X	X
E	1	1	0	1	X	X	X	X	X	X	X	X	X
F	1	1	1	1	X	X	X	X	X	X	X	X	X

3. ábra „Dominó” kód kombinációs táblája

D1 = D+C+B; D2 = D+C*B; D3 = D+C; D4 = D; (1)

D5 = A; D6 = D; D7 = D+C; D8 = D+C*B; D9 = D+C+B; (2)

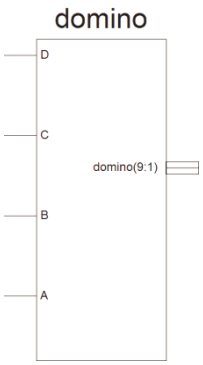
```
|-------------------------------------------------------------------------------

library IEEE;                                          -- könyvtár használat engedélyezése
use IEEE.STD_LOGIC_1164.ALL;                           -- csomag engedélyezés

entity domino is                                       -- egyed meghatározás kezdete
   Port (D, C, B, A : in STD_LOGIC;                    -- portok bemeneti jelek maghatározása
         Domino: out STD_LOGIC_VECTOR (9 downto 1));   -- kimeneti jelek meghatározása
end entity domino;

architecture Behavioral of domino is                   -- építmény leírás kezdete

begin                                                  -- kimeneti függvények lerírása
        Domino(1) <= D OR C OR B;                      -- Domino1
        Domino(2) <= D OR (C AND B);                   -- Domino2
        Domino(3) <= D OR C;                           -- Domino3
        Domino(4) <= D;                                -- Domino4
        Domino(5) <= A;                                -- Domino5
        Domino(6) <= D;                                -- Domino6
        Domino(7) <= D OR C;                           -- Domino7
        Domino(8) <= D OR (C AND B);                   -- Domino8
        Domino(9) <= D OR C OR B;                      -- Domino9

end Behavioral;                                        -- leírás vége
-------------------------------------------------------------------------------
```

4. ábra A „dominó” kód megvalósítása VHDL nyelvű programmal

1.4.1 Az tervezési egység/egyed (Entity) mint modell interfész

A tervezési egység meghatározza a terv nevét, azt, ahogyan a leírt VHDL elem csatlakozik más elemekhez, az áramköri egység alapvető jellemzőit, bemeneti és kimeneti lábait. Azaz milyen statikus paraméterekkel (konstansokkal) és dinamikus csatornákon (jelekkel) történik az információcsere, az egység (egyed) és környezete között.

Példában a „dominó” vezérlő áramköre rendelkezik egy 4 jelből meghatározott bemenettel (D, C, B, A) és egy kilencbites sín (busz) kimenettel (D[9:1]) (1. ábra). Megjegyezzük, hogy jelen esetben e bemenetek és kimenetek egyvezetékes jelek.



1. ábra A „dominó” kódot megvalósító vezérlő áramköre (VHDL programból létrehozott rajz szimbólum)

A tervezési egyed általános felépítése a következő (5. ábra):

```
-------------------------------------------------------------------------------------------------
entity <entity_nev> is
        -- paraméterek vagy statikus információátadási csatornák
        generic (<konstans_nev> : <tipusa> := <érték>;
                  <egyéb_konstansok>...          );
        -- csatlakozások, jelek vagy dinamikus információátadási csatornák
        port (    <port_nev> : <irany> <tipus>;
                  <port_nev> : <irany> <tipus>;
                  . . .
                  <port_nev> : <irany> <tipus>
        -- megjegyzés az utolsó port leírási sorát nem zárjuk le „;” karakterrel
        );
end entity <entity_nev>;
```

5. ábra Egyed meghatározás

A tervezési fontos része a ***port,*** amely tekinthető a tervezési egység dinamikus információcserét lebonyolító csatornájának (azaz a rendszer és környezetét összekötő jelek összességét meghatározó egység). A ***port*** meghatározása magába foglalja a ***port*** nevét (<portnév>), irányát (in - bemenet, out - kimenet, inout – két irányú port). A port iránya megadja a jel terjedési irányát. A kétirányú port esetén (inout) a jelterjedés mindkét irányban történhet. Más port típusok is léteznek, de ezekkel jelen fejezetben nem foglalkozunk. A példában a (D, C, B, A) bemeneti jelek ezeket a „dominó” vezérlő csak olvashatja, míg a sínrendszerként meghatározott domino(9:1) kimeneti jeleket csak írhatja a rendszer. A jel típusa harmadik jellemzője egy dinamikus csatornának. A típus azt határozza meg, hogy az adott jelünk egy- (STD_LOGIC) vagy több vezetéken, azaz sínrendszerben (STD_LOGIC_VECTOR) továbbítja az információt.

Példánkban egyvezetékes típusú jelek a bemeneteink, míg sín vagy vektor (VHDL meghatározás szerint) típusú a kimenetünk.

1.4.2 Megvalósítási egység (építmény - architecture)

Az építmény, a rendszer viselkedését/struktúráját leíró egység. VHDL-ben egy tervezési egységhez egy vagy több megvalósítási egység tartozik. Amennyiben több építmény írja le a rendszer viselkedését, úgy a feldolgozás során (szintézis, szimuláció) egyetlen megvalósítást rendelünk az egyedhez. A megvalósítási egység egyszerűsített felépítése a **Hiba! A hivatkozási forrás nem található.**. ábra mutatja.

```
-------------------------------------------------------------------------
architecture arch_nev of entity-nev is
-- belso_jelek_felsorolasa
signal
-- alkatreszek_felsorolasa
begin

egyideju_egyenletek;        --concurrentstatement
egyideju_egyenletek;        --concurrentstatement
...
end arch_nev ;
-------------------------------------------------------------------------
```

6. ábra Megvalósítási egység

Az architektúra első sora hozzákapcsolja a megvalósítást az előzőekben meghatározott egyedhez. Az architektúrában használt belső állandók és jelek meghatározása (amennyiben használunk ilyeneket) a következő lépés. Itt a kulcsszó a ***constant*** és a ***signal***. Például:

```
-------------------------------------------------------------------------------
-- belso_jelek_meghatározása
signal  ez_jel: std_logic;
signal ez_is_az: std_logic_vector (0 to 7);
-- pelda vege
-------------------------------------------------------------------------------
```

7. ábra Jel meghatározása

Az építmény leírása a ***begin – end arch_név*** páros között valósul meg. A konkurens (egyidejű) leírások határozzák meg a megvalósítást. A mi példánk esetében a „dominó vezérlő” a következő:

```
-------------------------------------------------------------------------------------------------------------
begin                                                   -- kimeneti fügvények lerírása
        Domino(1) <= D OR C OR B;                       -- Domino1 D1 = D+C+B;
        Domino(2) <= D OR (C AND B);                    -- Domino2 D2 = D+C*B;
        Domino(3) <= D OR C;                            -- Domino3 D3 = D+C;
        Domino(4) <= D;                                 -- Domino4 D4 = D;
        Domino(5) <= A;                                 -- Domino5 D5 = A;
        Domino(6) <= D;                                 -- Domino6 D6 = D;
        Domino(7) <= D OR C;                            -- Domino7 D7 = D+C;
        Domino(8) <= D OR (C AND B);                    -- Domino8 D8 = D+C*B;
        Domino(9) <= D OR C OR B;                       -- Domino9 D9 = D+C+B;

end Behavioral;                                         -- leírás vége
-------------------------------------------------------------------------------------------------------------
```

8. ábra Jel meghatározása

1.4.3 Tervezési egység, könyvtár

A tervezési egységek a VHDL program építőkövei. Egy VHDL program fordítása során a feldolgozó program öt önálló programegységet különböztet meg. Mindenik egységet elemzése külön – külön történik. Az öt önálló egység a következő:

- Tervezési egység meghatározása (Entity)
- Megvalósítási egység meghatározása (Architecture)
- Csomagok meghatározása (Package)
- Csomag leírása (Package body)
- Konfiguráció (Configuration)

Az előzőekben bemutattuk a tervezési és megvalósítási egységet. A VHDL csomag (***package)*** tartalmazza a gyakran használt adattípusokat, függvényeket, alkatrészeket, amelyeket a legtöbb VHDL programban használunk. A csomag meghatározási egységben leírjuk a csomagban megvalósított és használható elemeket, míg a csomag leírása tartalmazza a függvények, alkatrészek megvalósítását leíró egyenleteket, VHDL programkódokat, stb.

A VHDL jellegzetessége, hogy egy entitáshoz több architektúra is rendelhető. a konfiguráció meghatározza, hogy azt az építményt, melyet a tervezési egyedhez rendelünk az általunk meghatározott feltételek mellett.

A VHDL könyvtár (***library)*** tárolja a tervezési egységeket, amelyek a leggyakrabban használatos alkatrészeket, függvényeket tartalmazzák.

A gyors szintézis megvalósítása érdekében az IEEE létrehozott különböző szabványosított VHDL csomagokat, amelyeket az 1164, 1076.3 –as szabványokban határozott meg. A szabványos könyvtárakat ***library*** kulcsszóval nyitjuk meg, míg a használni kívánt könyvtári csomagot ***use*** kulcsszóval adjuk meg.

```
-------------------------------------------------------------------------------

-- szabványos konyvtar megnyitasa
library IEEE;
-- szabványos csomagok hasznalata
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
 -- pelda vege
-------------------------------------------------------------------------------
```

2. ábra Jel meghatározása

Azért kell meghívnunk a könyvtárat és azon belül a csomagokat, mert az adott VHDL programban használunk különböző a csomagokban meghatározott adat típusokat. Például: szabványos logika std_logic, szabványos sínrendszer std_logic_vector, amely a std_logic_1164 csomag része).

1.4.4 A VHDL program feldolgozása

A VHDL programot három lépcsőben dolgozza fel a keretrendszer: ***elemzés, előkészítés, végrehajtás***. Az ***elemzés*** során a fejlesztői keretrendszer ellenőrzi a VHDL program nyelvtani helyességét és statikus jelentéstani felépítését. Az elemzést a rendszer egy tervezési egységre végzi el. Mivel egy VHDL állományban egy entitáshoz több architektúra is tartozik a fordítás mindig az aktuális építményre történik. A szintaktikai hibák javítása után a fordító lefordítja a programot és eltárolja egy ideiglenes állományba. A komplex rendszerek esetében a hierarchikus leírás a jellemző. Ebben az esetben a legfelső szint az alrendszereket, mint alkatrészeket tartalmazza ezért a rendszer ezeket a VHDL (al)-programokat is ellenőrzi.

Az ***előkészítési*** művelet során a fordító kiindulva a legfelső szintű tervezési egységből összeköti az építményben lévő elemeket a konfigurációs beállítások alapján. Az egyes alkatrészekhez hozzáköti a megfelelő építményeket. A folyamat rekurziv módon addig ismétlődik, míg a fordító rendszer az egész hierarchiában minden alkatrészhez hozzárendeli a megfelelő architektúrát. A ***végrehajtási*** fázisban az előzőekben elemzett és összekötött leírást előkészíti a szoftver a szimuláció vagy a szintézis számára.

1.5 A VHDL szintaktika és elemek

1.5.1 Lexikális elemek

A lexikális elemek a VHDL nyelv alapját képező szinaktikai egységek. Ide tartoznak a megjegyzések, az azonósítók, a fentartott szavak, számok, karakterek, és a karakterláncok (string).

A megjegyzés – comment „- -„ két mínuszjel után írt szöveg. A megjegyzést a rendszer fordításkor figyelmen kívül hagyja. A megjegyzéseket legfőképpen a program dokumentálására használjuk.

```
-- megjegyzés példa
library IEEE;
-- szabványos csomagok hasznalata
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- megjegyzés bárhol állhat, de legjobb ha a programsor végén van

architecture megjegyez of comment is
begin
signal jel: std-logic; -- a „jel” egy belső jel
jel <= be1 and be2; -- a jel értéke az be1 ÉS be2
ki  <= jel; -- a jel-et egy kimenthez rendeljük
egy_masik_ki <= not jel;
end megjegyez;

-- pelda vege
```

Figure 3. ábra Megjegyzésre példa

Azonosító – identifier egy VHDL objektum neve**.** Az azonosítókra a következő szabályok érvényesek:

- az azonosító megnevezésére csupán a latin abc betűit, decimális számokat és aláhúzás karaktereket használhatjuk;
- az azonosító első karaktere mindig betű;
- az utolsó karakter nem lehet aláhúzás („_”)

Helyes példák azonosítókra: A10, next_state, KovetkezoAllapot, mem_cim_eng

Mivel a VHDL nem különbözteti meg a kis és nagybetűket ezért a következő azonosító nevek mind ugyanarra az objektumra hivatkoznak: azonosito, azonoSITO, AZONOSITO, AzOnoSiTo. Ezért a helyes programírási mód az, ha mindig ugyanúgy hivatkozunk egy - egy objektumra. Az azonosítók egyik helyes használatára jó ha az azonosítókat csupa nagybetűkkel írjuk és a kisbetűk használata az objektumok nevében valamely tulajdonságra utal Hiba: A hivatkozás forrása nem található. Például az „n” utótag jelentheti azt is, hogy a jel alacsonyszinten aktív. A gyakorlat az, hogy az objektumok névadásában minél kifejezőbb neveket használunk, azonban fenntartott szavak használata tilos!

A fenntartott szavak* (*Reserved words) a VHDL alapvető nyelvi felépítésében vesznek részt. Ezek a szavak a következők a teljesség igénye nélkül:

abs, access, after, alias, and, architecture, array, assert, attribute, begin, block, body, buffer, bus, case, component, configuration, constant, disconnect, downto, else, elsif, end, entity, exit, file, for, function, generate, generic, guarded, if, impure, in, inertial, inout, is, label, library, linkage, literal, loop, map, mod, nand, new, next, nor, not, null, of, on, open, or, others, package, port, postponed, procedure, process, pure, range, record, register, eject, rem, report, return, rol, ror, select,severity, shared, signals, sra, srl, subtype, then, to, transport, type unaffected, units, until, use, variable, wait, when, while, with, xnor, xor Hiba: A hivatkozás forrása nem található**.**

Számok – Numbers, karakterek - characters és ***karakter sorok - string-ek***. VHDL-ben a **számok** típusa lehet egész, lebegőpontos, valós. A számok ábrázolásában a tízes számrendszer (pl. 23), a kettős számrendszer (pl. 23 => 2#10111#), vagy a tizenhatos /hexadecimális/ pl. (23 =>16#17#)

használatos. Például 123456 szám azonos az 123_456 számmal, mint ahogyan 2#101001011010# bináris szám is azonos a 2#1010_0101_1010# számmal.

A ***karakter – character*** típusokat aposztrófok között jelöljük, például 'V', 'H', 'D', 'L'. Megjegyezzük, hogy a 23 és '23' közül az első egy szám, míg a második egy karakter típus. A ***karakter sorok*** vagy ***string***-ek jelölése kettős idézőjelek között történik, úgymint „VHDL". Itt is megjegyezzük, hogy a számok (2#0101_1010#) és a karakter sorok („0101_1010") jelölés és tartalom szempontjából is különbözik, mi több a következő két karakter sor sem azonos: „0101_1010", („01011010").

1.5.2 Objektumok

Az objektum (object) azonosítóval ellátott, adott értéket felvevő/tartalmazó adat típus. Négy ilyen objektum típust ismerünk: *a **jel**et (signal), a **változó**t (variable), az **állandó**t (konstans - constant) és az **állomány**t (file).* Az állomány típust ebben a fejezetben nem tárgyaljuk, mivel nem szintetizálható elem.

A jeleket az architektúra elején („architecture" után, „begin" előtt) kell meghatároznunk. A jelek csak az abban az architektúrában látszanak, amelyekben meghatároztuk őket.

A jel meghatározása a következő:

signal jel_nev: jel_típus;

A jel értékadása a következő értékadó művelettel történik: **<=**

Például:

jel <= be1 and be2;

Az entitás kimeneti és bemeneti portjai is jeleknek tekinthetők.

A változók hasonlóan a hagyományos értelemben vett programozási nyelvek változóihoz, ugyanazt a szerepet töltik be.. A változókat használata csak a folyamatokban (process) engedélyesett. A változó felfogható úgy is, mint „helyi szimbolikus tároló elem" Hiba: A hivatkozás forrása nem található A változók használatával leírható a rendszer absztrakt viselkedése. A változó meghatározása a következő:

variable valtozo_nev: valtozó_típus;

A jel értékadása a következő értékadó művelettel történik: **:=**

Például:

valtozo := be1 and be2;

Megjegyzés: Mivel a változóhoz a fordító rendszer semmiféle időzítési értéket nem rendel, ezért a változó értékadása azonnali hatással van a változóra.

Az állandó értéke nem változtatható. Az állandó meghatározása a következő módon történik:

constant allando_nev: adat_tipus := ertek_vagy_kifejezés;

Például:

constant sin_rendszer_merete: integer := 32;

constant sin_rendszer_bytejainak_a_szama: integer := **sin_rendszer_merete** / **8;**

Az állandók olvashatóbbá teszik a VHDL programunkat, ugyanakkor a program rugalmasabban módosítható.

1.5.3 Adat típusok, műveletek

A VHDL nyelvben minden változóhoz (azaz objektumhoz vagy jelhez) hozzárendelünk egy adat típust. Az adat típus meghatározza a változók által felvehető értékek halmazát és a változókkal elvégezhető műveletek halmazát. Mivel a VHDL nyelvben a jelekhez rendelhető értékek és a jelekkel végezhető műveletek erősen típusfüggőek, ezért ha egy más adat típusú értéket akarunk hozzárendelni vagy más adat típusú műveletet akarunk elvégezni az adott objektummal, akkor az adott jelet típuskonverzióval át kell alakítanunk megfelelő típusúra.

1.5.4 Szabványos VHDL adattípusok

A szintézis szempontjából a leggyakrabban előforduló adattípusok a következők:

- **egész – integer típus**: a VHDL nyelvben a [-2^{31}-1, 2^{31}-1] intervallumban határozzuk meg az egész típust, ami megfelel a 32 bites ábrázolási módnak. Két altípust is meghatározunk a természetes (**N***) és a pozitív (**N**) típust;
- **boole – boolean típus: {**igaz (true), hamis (false)} értékekkel;
- **bit típus:** {'0', '1'} értékekkel;
- **bit_vector típus:** amelyet egydimenziós bit típusú elemekből álló tömbként határozunk meg.

Valós digitális tervezés esetén egy jelhez nem csupán kétállapotú jelszinteket rendelhetünk, hiszen a jel lehet nagyimpedanciás állapotú vagy akár lehet jelkonfliktus is huzalozott logika esetén. Ezért a VHDL nyelvben a probléma megoldására bevezették a STD_LOGIC és az STD_LOGIC_VECTOR típust az IEEE std_logic_1164 csomagjában. Mivel ez egy rugalmasabb jelkezelést megengedő adat típus ezért a példák során ezt fogjuk használni.

Table 1. táblázat A VHDL-93 szabvány szerinti műveletek

Művelet	**leírás**	a **operandus adat típusa**	b **operandus adat típusa**	**eredmény adat típusa**
	számtani műveletek			
a **+** b	összeadás	integer	integer	integer
a **-** b	kivonás	integer	integer	integer
a ***** b	szorzás	integer	integer	integer
a **/** b	osztás	integer	integer	integer

a **mod** b	modulo	integer	integer	integer
a **rem** b	osztási maradék	integer	integer	integer
a ****** b	exponenciális	integer	integer	integer
abs a	abszolút érték	integer		integer
	tömb művelet			
a **&** b	összeláncolás	egydimenziós tömb	egydimenziós tömb	egydimenziós tömb
a **sll** b	balra tolás b bittel	bit_vector	integer	bit_vector
a **srl** b	jobbra tolás b bittel	bit_vector	integer	bit_vector
a **sla** b	balra shift balra tolás	bit_vector	integer	bit_vector
a **sra** b	jobbra shift balra tolás	bit_vector	integer	bit_vector
a **rol** b	balra forgatás	bit_vector	integer	bit_vector
a **ror** b	jobbra forgatás	bit_vector	integer	bit_vector
	boole algebrai műveletek			
not a	negáció	boolean, bit, bit_vector		boolean, bit, bit_vector
a **and** b	and	boolean, bit, bit_vector	„a” típussal azonos	boolean, bit, bit_vector
a **or** b	or	boolean, bit, bit_vector	„a” típussal azonos	boolean, bit, bit_vector
a **xor** b	xor	boolean, bit, bit_vector	„a” típussal azonos	boolean, bit, bit_vector
a **nand** b	nand	boolean, bit,	„a” típussal azonos	boolean, bit,

		bit_vector		bit_vector
a **nor** b	nor	boolean, bit, bit_vector	„a” típussal azonos	boolean, bit, bit_vector
a **xnor** b	and	boolean, bit, bit_vector	„a” típussal azonos	boolean, bit, bit_vector
	reláció műveletek			
a **=** b	a egyenlő b	bármely típus	„a” típussal azonos	boolean
a **/=** b	a nem egyenlő b	bármely típus	„a” típussal azonos	boolean
a **<** b	a kisebb b	skaláris, egydimenziós tömb	„a” típussal azonos	boolean
a **<=** b	a kisebb egyenlő b	skaláris, egydimenziós tömb	„a” típussal azonos	boolean
a **>** b	a nagyobb b	skaláris, egydimenziós tömb	„a” típussal azonos	boolean
a **>=** b	a nagyobb egyenlő b	skaláris, egydimenziós tömb	„a” típussal azonos	boolean
		skaláris, egydimenziós tömb	„a” típussal azonos	boolean

A szintézis során a VHDL nyelvű program fizikai alkatrészekkel valósul meg. A műveletek által igényelt erőforrások eltérőek.

STD_LOGIC, STD_LOGIC_VECTOR adattípus

A két leggyakrabban használt adattípus amelyet a std_logic_1164 –es csomag tartalmaz. A std_logic típus az előjelnélküli std_ulogic altípusa amelyet a std_ulogic_1164 –es csomag tartalmaz. Néhány esetben ezt a típust használjuk, amikor használatát meg is magyarázzuk.

Ahhoz, hogy valamely adattípust használjunk ahhoz a megfelelő csomagot meg kell nyitnunk:

library ieee;

use ieee.std_logic_1164.all;

A szabványos logikai (std_logic) típusn kilenc lehetséges értéket vehet fel:

{’U’, ’X’, ’0’, ’1’, ’Z’, ’W’, ’L’, ’H’, ’-’}

A fenti értékek értelmezése a következő:

- A ’0’ és ’1’ - a „határozott logikai 0” illetve a „határozott logikai 1” értékeket jelölik, amelyeket akkor vesz fel egy jel ha egy szabványos meghajtó áramkör vezérli;
- A ’Z’ nagyimpedanciás értéket a jel akkor veheti fel ha a meghajtó áramköre egy háromállapotú meghajtó áramkör (tri-state buffer);
- Az ’L’ és ’H’ – „gyenge logikai 0” illetve „gyenge logikai 1” értékeket egy jel huzalozott logikai áramkörök esetében veheti fel, amikor a meghajtó vezérlő áramkör gyenge meghajtó áramot szolgáltat.
- Az ’X’ és a ’W’ – „határozatlan logikai állapot” illetve „gyenge határozatlan logikai állapot” a jel egy olyan értéket vesz fel amit nem lehet sem logikai 0-nak sem logikai 1-nek értelmezni. Ez az állapot akkor következik be amikor két vezérlő egymásnak ellentmondó értékű (logikai 0 és 1) logikai állapotba vezérli az jelet. Az állapotok főleg áramköri szimulációnál jelölik a hibás áramköri működést.
- ’U’ – Szimulációban használt állapot. Azt mutatja, hogy a jelhez vagy a változóhoz még nem rendeltünk értéket.
- ’-’ a jel redunáns voltát jelöli.

1.5.5 Általános VHDL kódolási ajánlások

A VHDL egy komplex nyelv, melyet az IEEE 1076-os szabvány rögzít. A legfrissebb szabvány az IEEE 1076-2008. A szabvány leírása az IEEE Standard VHDL Language Reference Manual – VHDL referencia kézikönyv vagy rövidítve LRM kézikönyvként ismert.

Jelen VHDL leírás főleg a szintézisre vonatkozóan ad útmutatást, ezért legfőképpen a szintézist elősegítő programírási ajánlásokat soroljuk fel az alábbiakban, a teljesség igénye nélkül Hiba: A hivatkozás forrása nem található:

- az std_logic és a std_logic_vector adattípus használata ajánlott a bit és a bit_vector helyett;
- aritmetikai műveletek alkalmazásakor a numeric_std csomag és a csomagban lévő unsigned és signed adattípusok használata ajánlott. Például:

```
library IEEE;

use IEEE.STD_LOGIC_ARITH.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

- unisegned típus leírásakor a csökkentő sorrend (**downto**) használata ajánlott. Például:

```
port (domino: out std_logic_vector (8 downto 0));
```

- a kívánt műveletvégzési sorrend zárójelek használatával érhető el;
- Ha csak nem indokolt, akkor a felhasználó átlal meghatározott adattípusok használata kerülendő;
- ne használjuk az azonnali értékadás jelet („:=”) egy jel alapállapotba állításához;
- a relációs műveletek esetében az operandusok legyenek azonos méretűek.

1.6 Egyidejű értékadási azonosságok a VHDL nyelvben

Az értékadási azonosság és a hardvere elemek között szoros kapcsolat van, ezért a VHDL nyelv segítségével hatékony hardver megvalósítást lehet elérni. A VHDL szabvány szerint az egyidejű/párhuzamos értékadás három módon tehető meg a VHDL nyelvben:

- egyszerű értékadási azonosság ***(simple signal assignment statement)*** – ami megfelel a Boole algebrai egyenleteknek.

- feltételes értékadás ***(conditional signal assignment statement)***;
- jelkombinációk szerinti értékadás ***(selected signal assignment statement)***;

1.6.1 Kombinációs/sorrendi hálózatok

A digitális áramkörök kombinációs és sorrendi hálózatokba csoportosíthatók. A kombinációs hálózatok nem rendelkeznek belső tároló elemekkel (memóriával) és belső köztes állapotokkal. A kombinációs hálózatok esetében a bemeneti változók adott értékeire mindig ugyanazt a kimeneti függvény értékeket adják. A valóságban rövid átmeneti tranziensek jelenhetnek meg az áramkörök késleltetéséből adódóan. Azonban a bementi változók állandósult állapotában mindig ugyanazt a kimeneti értéket veszik fel. A kombinációs hálózat a megvalósítás szempontjából (programozható logikai áramkörökkel) egy memória vagy belső tároló nélküli (latch vagy flip-flop) vagy visszacsatolás nélküli áramkör.
A sorrendi hálózatok kimenetei a bemeneti változók állapotának és a belső változók állapotának függvényei. A sorrendi hálózatok leírása megtehető egyidejű értékadással, de az általánosan elterjedt módszer nem ez. A folyamat az a szekvenciális leírási mód, aminek segítségével a párhuzamos/egyidejű eseményeket leíró VHDL program kezeli a sorrendi hálózatokat.

1.6.2 Egyszerű értékadás

Az egyszerű értékadás feltétel nélkül azonnal hozzárendel egy jelhez egy kifejezést. A kifejezés kiértékelése megtörténik, ha a kifejezésben szereplő objektumok bármelyike változik. A kifejezés kétféle leírást tartalmaz. Az egyik leírás konstans illetve logikai vagy algebrai kifejezés kiértékeléséből adódik, míg a másik megadja, hogy az új érték hozzárendelése mikor történik meg.

jel_nev <= kifejezes;

Például vegyük a következő értékadást:

F **<=** A **xor** B **after 10ns**;

A kifejezés jelzi, hogy a bemeneti jelek (A, B) bármelyikének változására az „A **xor** B” kifejezés kiértékelődik és az F kimeneti jel **10ns**-os késleltetéssel rendeljük hozzá a változást.

Az **„after”** kulcsszó megfelel az áramkörök belső késleltetésének jelölésére és kiegészíti a logikai vagy algebrai egyenletet a megadott áramköri késleltetéssel.

Megjegyzés: Mivel a tervezőrendszer szintetizáló motorja és az áramkörök sokfélesége következtében nehezen megbecsülhető a késleltetési idő, ezért ezt a paramétert inkább szimulációs elemzések kapcsán használjuk, és szintézisnél inkább mellőzzük.

Íme néhány példa szintetizálható kifejezésekre:

allapot <= ’0’;

kimenet <= „0101”;

led <= (a **and** b) **or** (c **and** (**not** d));

osszeg <= a **+** b **+** cy **-**1;

1.6.3 Feltételes értékadás

Feltételes értékadáskor Boole algebrai kifejezések sorát elemzi a rendszer és az első igaznak (true) kiértékelt azonosságnak megfelelően megtörténik az értékadás. (A Boole algebrai kifejezés értéke lehet hamis – false, vagy igaz – true.). A feltételes értékadás általános kifejezése:

jel_nev <= *ertek_vagy_kifejezes_0* when *boole_algerbai_kifejezes_0* else

***ertek_vagy_kifejezes_1* when *boole_algerbai_kifejezes_1* else**

***ertek_vagy_kifejezes_2* when *boole_algerbai_kifejezes_2* else**

…

***ertek_vagy_kifejezes_n* when *boole_algerbai_kifejezes_n*;**

A *boole_algebrai_kifejezes_i (i = 0, 1, . . .,n)* kiértékelése igaz vagy hamis értéket ad, amelynek függvényében megtörténik az értékadás vagy sem. Amennyiben a j-eidk kifejezés értéke igaz abban az esetben a *ertek_vagy_kifejezes_j* értéket veszi fel a kimeneti jel (jel_nev).

Az alábbiakban bemutatunk néhány példát a feltételes értékadás bemutatásához.

1.6.4 Multiplexer megvalósítása

Feladat 4 bemenetű multiplexer áramkör megvalósítása. A multiplexer bemenetei A, B, C, D egyenként 4 bites vektorok. a multiplexer kimenete F, címvezetéke pedig Cím. A megvalósított példaprogram a Figure 4. ábra látható. A program első két sora megnyitja a szabványos logikai csomagot (IEEE.std_logic_1164), melynek eredménye képpen használhatjuk a szabványos adat típust. A következő programrészben rész a tervezési egységben a bemeneti jelek (A, B, C, D, CIM) és a kimeneti jel (MUX_KI) meghatározása történik. Mivel a multiplexer bemenetei, kimenete négy bites ezért a jelek szabványos logikai vektorok. A négy bemenet kiválasztását a CÍM jel segítségével valósítjuk meg.

```
----------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
        -- Felteteles ertekadas pelda
        -- bemenetek: A, B, C, D,
        -- kimenet: MUX_KI
        -- cim: CIM
entity multiplexer is
   Port( A    : in  STD_LOGIC_VECTOR (3 downto 0);
         B    : in  STD_LOGIC_VECTOR (3 downto 0);
         C    : in  STD_LOGIC_VECTOR (3 downto 0);
         D    : in  STD_LOGIC_VECTOR (3 downto 0);
         CIM  : in  STD_LOGIC_VECTOR (1 downto 0);
         MUX_KI: out STD_LOGIC_VECTOR (3 downto 0));
end multiplexer;

architecture Behavioral of multiplexer is

begin
        MUX_KI <= A when CIM = "00" else
                  B when CIM = "01" else
                  C when CIM = "10" else
                  D;
end Behavioral;
----------------------------------------------------------------------------------------------------
```

Figure 4. ábra Feltételes értékadás példa: multiplexer

A megvalósítási egység vagy architektúra részben először a CIM = „00” Boole algebrai feltétel kerül kiértékelésre és amennyiben „igaz” értékű, úgy a kimenethez (MUX_KI) az A sin rendszert értékét veszi fel. Amennyiben a Boole algebrai feltétel „hamis” úgy a következő feltétel kerül kiértékelésre (CÍM = „01”), majd amennyiben ez is „hamis” értékű a következő feltételt értékeli ki a rendszer. Míg ha egyik feltétel sem igaz úgy a kimenet a D jel értékét veszi fel. A feladat szimulációjában (Figure 5. ábra) a bemeneti jelekhez a következő értékeket rendeltük:

$$A = 0001; B = 0010; C = 0100; D = 1000; \qquad (3)$$

A címvezeték pedig rendre az m0, m1, m2, m3 mintermek értékét veszi fel. A kimeneten pedig jól látható, hogy a megfelelő CÍM-hez hozzárendelt bemenet értéke kerül a kimenetre.

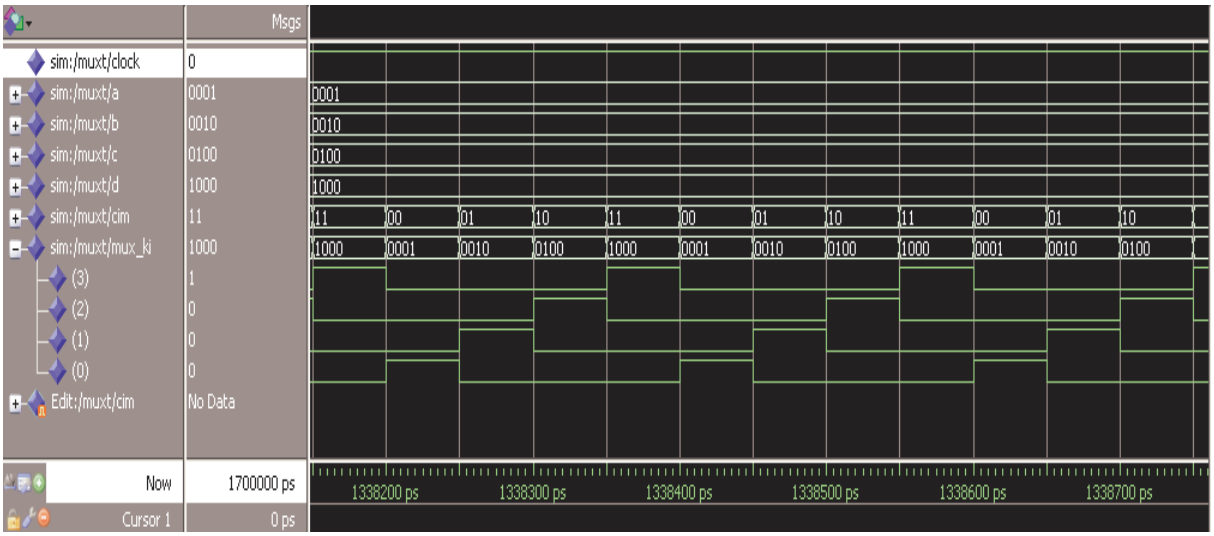


Figure 5. ábra. Multiplexer példa szimulációs eredménye

Megjegyzés: Mint ahogyan azt a Hiba: A hivatkozás forrása nem található fejezetben ismertettük a szabványos logikai adattípus 9 különböző állapotot vehet fel. Ily módón a „00”, „01”, „10”, „11” kombinációkkal együtt összesen 81 (9*9) lehetséges CIM értékünk lenne. Ezért az utolsó értékadás (MUX_KI=D;) nemcsupán az „11” bemeneti kombinációt hanem az összes többi nem említett kombinációt is kezeli. A valóságban előfordulhat ugyan a „Z0” vagy a „XU” kombináció, de értelmetlen. Ezért a szintézis során az áramkör úgy kerül megvalósításra, mint ahogyan azt elvárjuk.

1.6.5 Négydigites hétszegmenses kijelző anódjainak vezérlése

Második példánkban a Figure 8. ábra látható négydigites hétszegmenses kijelző anódjainak a vezérlését mutatjuk be. Az anódok egyenkénti vezérlése akkor szükséges amikor az egyes digitekre különböző adatokat szeretnénk kijelezni például egy 16 bites (4x4) számláló aktuális értékét.

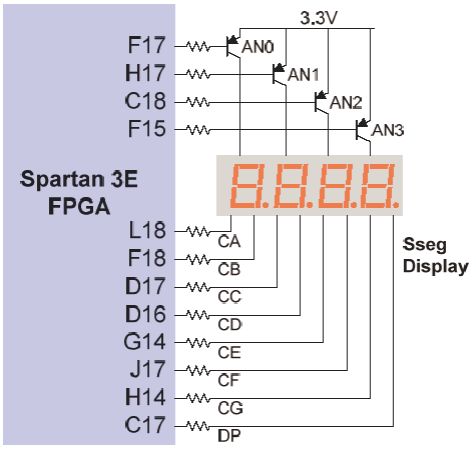


Figure 6. ábra Négydigites hétszegmenses kijelző vezérlése FPGA áramkörrel Hiba: A hivatkozás forrása nem található

Egy-egy digit szegmensei csak akkor fognak világítani, ha az egyes anódokat (AN0-AN3) vezérlő tranzisztorok bázisát nyitó feszültséggel vezéreljük, azaz „0” logikai szinttel.

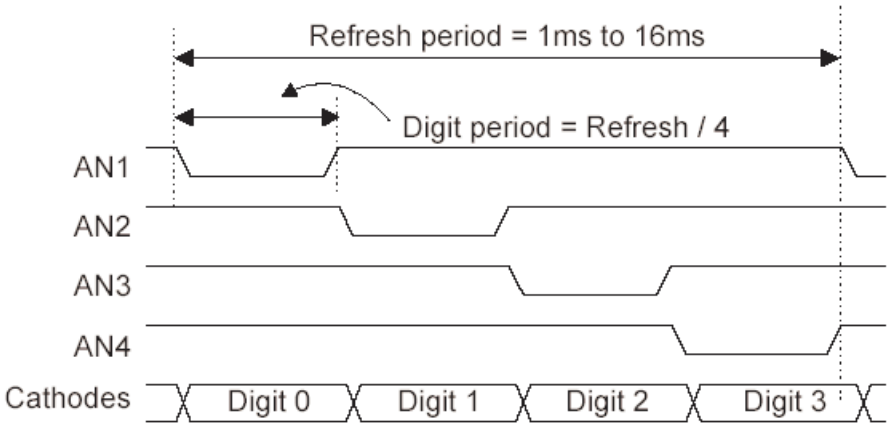


Figure 7. ábra A hétszegmenses kijelző anódjainak vezérlési diagramja

Az anódvezérlő áramkört tekinthetjük egy 2-2^2 bináris dekóder áramkörnek, amelynek a kombinációs tábláját a Table 2. táblázat mutatja:

Table 2. táblázat Anódvezérlő kombinációs táblája

CIM1	CIM0	AN3	AN2	AN1	AN0
0	0	1	1	1	**0**
0	1	1	1	**0**	1
1	0	1	**0**	1	1
1	1	**0**	1	1	1

A táblázatból látható, hogy AN[3:0] azon bitje lesz nulla, amelyre a CIM[1:0] által képzett minterm mutat. A fentiekben ismertetett feladat VHDL megvalósítását a **Hiba! A hivatkozási forrás nem található.**. ábrán látható.

A program ugyanúgy, mint az eddigiekben és ezután is a szabványos logikai adattípust használja (IEEE.STD_LOGIC_1164.ALL csomag használata). Az építményben két jelet határozunk meg a bemeneti 2 bites sínt (CIM) és a kimeneti 4 bites sínt (AN).

A megvalósítási részben Az AN(0) = ’0’ amikor CÍM = „00”, AN(1) = ’0’ amikor CÍM = „01”, AN(2) = ’0’ amikor CÍM = „10” és végül AN(3) = ’0’ amikor CÍM = „11”.

A megtervezett áramkör szimulációjának eredménye a Figure 9. ábran látható.

```
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--
-- hetszegmeses kijelző anodjainak dekodolasa
-- bemenet: cim vezetek 2 bites
-- kimenet: anod 4 bites
-- az aktiv anod 0 erteket vesz fel
--
entity anod_dekoder is
   Port( CIM : in STD_LOGIC_VECTOR (1 downto 0);
         AN : out STD_LOGIC_VECTOR (3 downto 0));
end anod_dekoder;

architecture Behavioral of anod_dekoder is

begin
        AN <=  "1110" when CIM = "00" else -- AN0 tranzisztor bázisán 0V
               "1101" when CIM = "01" else -- AN1 tranzisztor bázisán 0V
               "1011" when CIM = "10" else -- AN2 tranzisztor bázisán 0V
               "0111";                     -- AN3 tranzisztor bázisán 0V

end Behavioral;
----------------------------------------------------------------------------------
```

Figure 8. ábra Bináris dekóder: 4 digites hétszegmenses kijelző digit dekódoló

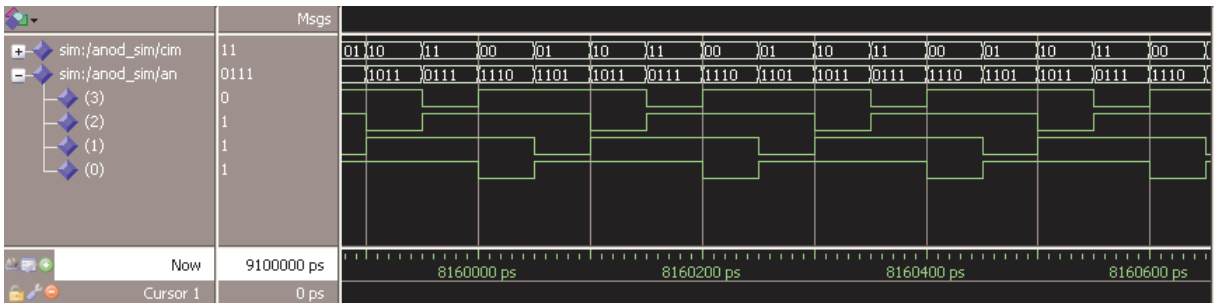


Figure 9. ábra Az anódvezérlő áramkör szimulációs eredménye

Az áramköri szimuláció eredménye mutatja, hogy az anódok vezérlése a bemeneti jel függvényében történik.

1.6.6 Jelkombinációk szerinti értékadás

A jelkombinációk szerinti értékadás is egy értéket vagy kifejezést rendel az adott jelhez, azonban ez akkor történik meg, amikor egy vezérlő-jel vagy kifejezés a kiértékelés eredményeképpen felvesz egy adott értéket a lehetséges értékek közül. A vezérlő-jel/kifejezés nem felsorolt értékei szerinti értékadás az „***others***” kulcsszóval történik.

with *vezérlő_jel_vagy_kifejezes* SELect

jel_nev <= *ertek_vagy_kifejezes_0* **when** *ertek_0*,

ertek_vagy_kifejezes_1 **when** *ertek _1,*

ertek_vagy_kifejezes_2 **when** *ertek _2,*

…

ertek_vagy_kifejezes_n **when** *kiretekelesi_eredmeny _n-1*,

ertek_vagy_kifejezes_n **when** ***others***;

A vezérló jel vagy kifejezés kiértékelésének eredménye diszkrét vagy edgydimenziós tömb lehet. Azaz a kiértékelés véges számú értékeket vehet fel. Ha az előző fejezetben ismertetett anódvezérlő áramkört vesszük példának akkor annak megvalósítási egysége a következő:

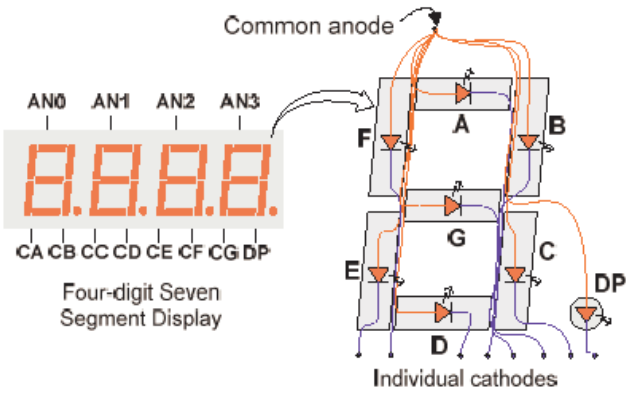
with CIM SELect

```
AN <=  "1101" when "01", -- 1
       "1011" when "10", -- 2
       "0111" when "11", -- 3
       "1110" when others , -- 0 és minden egyeb erteknel
```

Ebben az esetben a CÍM egy egydimenziós tömb (STD_LOGIC_VECTOR) egy kétbites sín. melynek állapotai rendre ”01”, ”10”, ”11” és ”00”. A felvehető értékek exkluzívak (egy érték sem használható fel többször) és inkluzívak is (az összes értéket fel kell használni) egyben. Az „***others”*** kulcsszó jelenti az összes fel nem használt értéket, példánkban a ”00” és az összes jelállapotból adódó (Z, X, H, L, stb.) kombinációt is. Azaz az „others” jelenti a „ZZ”, „UX”, stb. értékeket is.

1.6.7 Hétszegmenses kijelző vezérlése

A hétszegmenses kijelző szegmenseinek vezérlését bemutató példával tovább folytatjuk a **Hiba! A hivatkozási forrás nem található.**. ábrán már bemutatott négydigites hétszegmenses kijelző vezérlésének tervezését. A **Figure 10** ábrán látható a közös anódú négydigites hétszegmenses kijelző.



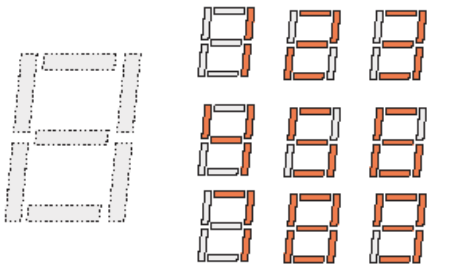


Figure 10. ábra Közös anódú négydigites hétszegmenses kijelző Hiba: A hivatkozás forrása nem található

Az ábrán jól látható, hogy egy-egy szegmens akkor világít, ha a megfelelő LED dióda katódját (A, …, DP) kisebb feszültséggel vezéreljük, mint az anód feszültsége, ez esetünkben a logikai „0” szintnek megfelelő feszültség szint (0.2V). Tehát a kombinációs táblánkban (Table 3. táblázat) a vezérelni kívánt kimeneti függvények (összesen 8) ott logikai „0” értékűek ahol a megfelelő szegmenst (LED-et) vezérlejük. Példánkban a DP – decimális pont, akkor vezérelt amikor a kijelzett számjegy nagyobb mint 9.

Table 3. táblázat Hétszegmenses kijelző vezérlése

mi	HEX 3	HEX 2	HEX 1	HEX 0	SLED7 (DP)	SLED 6 (G)	SLED 5 (F)	SLED 4 (E)	SLED 3 (D)	SLED 2 (C)	SLED 1 (B)	SLED 0 (A)
0	0	0	0	0	1	1	0	0	0	0	0	0
1	0	0	0	1	1	1	1	1	1	0	0	
2	0	0	1	0	1	0	1	0	0	1	0	0
3	0	0	1	1	1	0	1	1	0	0	0	0
4	0	1	0	0	1	0	0	1	1	0	0	1
5	0	1	0	1	1	0	0	1	0	0	1	0
6	0	1	1	0	1	0	0	0	0	0	1	1
7	0	1	1	1	1	1	1	1	1	0	0	0
8	1	0	0	0	1	0	0	0	0	0	0	0
9	1	0	0	1	1	0	0	1	0	0	0	0
A	1	0	1	0	0	0	0	0	1	0	0	0
B	1	0	1	1	0	0	0	0	0	0	1	1
C	1	1	0	0	0	1	0	0	0	0	1	1
D	1	1	0	1	0	0	1	0	0	0	0	1
E	1	1	1	0	0	0	0	0	0	1	1	0
F	1	1	1	1	0	0	0	0	1	1	1	0

A bemeneti változók, amelyek a jelkombinációk szerinti értékadást vezérlik HEX[3:0], míg a vezérlet szegmensek az SLED[7:0] vektorok. Az elkészült VHDL program a Figure 11. ábran látható. Hasonló módon a szabványos logikai csomagot használjuk (IEEE:STD_LOGIC_1164.ALL). A bemeneti és kimeneti portok elnevezését a Table 3. táblázat szerint választottuk.

```
----------------------------------------------------------------------------------------------------------
-- Design Name:   hétszegmenses kijelző szegmenseinek vezérlése
-- Module Name:  hetossg - Behavioral
-- Project Name:   hetszegmeses vezerlo
-- Target Devices: sp3e
----------------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity hetosled is
    Port ( HEX : in  STD_LOGIC_VECTOR (3 downto 0);
           SLED : out  STD_LOGIC_VECTOR (7 downto 0));
end hetosled;

architecture Behavioral of hetosled is

begin

          with HEX SELect
           SLED(6 downto 0)<=    "1111001" when "0001",    --1
                                 "0100100" when "0010",   --2
                                 "0110000" when "0011",   --3
                                 "0011001" when "0100",   --4
                                 "0010010" when "0101",   --5
                                 "0000010" when "0110",   --6
                                 "1111000" when "0111",   --7
                                 "0000000" when "1000",   --8
                                 "0010000" when "1001",   --9
                                 "0001000" when "1010",   --A
                                 "0000011" when "1011",   --b
                                 "1000110" when "1100",   --C
                                 "0100001" when "1101",   --d
                                 "0000110" when "1110",   --E
                                 "0001110" when "1111",   --F
                                 "1000000" when others;   --0
          SLED(7) <= '0' when HEX < "1001" else '1' ; -- a DP akkor 1 ha HEX > 9;

end Behavioral;
----------------------------------------------------------------------------------------------------------
```

Figure 11. ábra Hétszegmenses kijelző vezérlő.

Mivel az SLED(7) – DP – nem része a feltételes értékadásnak, ezért külön kell vezérelnünk. A fenti megoldás (**Hiba! A hivatkozási forrás nem található.**. ábra) az összes hexadecimális számnak kijelzéséhez szükséges függvényt valósítja meg. Azért hogy jobban megérthessük a „minden egyéb” (***others***) kulcsszót, irjuk át a feladatot úgy hogy a hétszegmenses kijelző csupán a BCD számokat ábrázolja. A **Hiba! A hivatkozási forrás nem található.**. ábra ezt a megoldást mutatja. Ha megfigyeljük akkor a számunkra felesleges programsorok megjegyzésként vannak beillesztve a programba.

```
----------------------------------------------------------------------------------------------------------
-- Design Name:   hétszegmenses kijelző szegmenseinek vezérlése
-- Module Name:  hetossg - Behavioral
-- Project Name:   hetszegmeses vezerlo
-- Target Devices: sp3e
----------------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity hetosled is
    Port ( HEX : in  STD_LOGIC_VECTOR (3 downto 0);
           SLED : out  STD_LOGIC_VECTOR (7 downto 0));
end hetosled;

architecture Behavioral of hetosled is

begin

          with HEX SELect
           SLED(6 downto 0)<=    "1111001" when "0001",    --1
                                  "0100100" when "0010",   --2
                                  "0110000" when "0011",   --3
                                  "0011001" when "0100",   --4
                                  "0010010" when "0101",   --5
                                  "0000010" when "0110",   --6
                                  "1111000" when "0111",   --7
                                  "0000000" when "1000",   --8
                                  "0010000" when "1001",   --9
                                  -- -- ez nem BCD: "0001000" when "1010",   --A
                                  -- -- ez nem BCD:"0000011" when "1011",   --b
                                  -- -- ez nem BCD:"1000110" when "1100",   --C
                                  -- -- ez nem BCD:"0100001" when "1101",   --d
                                  -- -- ez nem BCD:"0000110" when "1110",   --E
                                  -- -- ez nem BCD:"0001110" when "1111",   --F
                                  "1000000" when „0000”,
                                 ”1111111”  when others;   -- A, B, C, D, E, F
          SLED(7) <= '0' when HEX < "1001" else '1' ; -- a DP akkor 1 ha HEX > 9;

end Behavioral;
----------------------------------------------------------------------------------------------------------
```

Figure 12. ábra BCD Számábrázolás - hétszegmenses kijelző vezérlésével.

Az is megfigyelhető, hogy a módosított programunkban a ”0” vezérlése kikerült az „***others***” kulcsszó hatálya alól és a kulcsszó most a nem BCD számjegyeket kezeli. Ha ezt egy kombinációs táblán kellene ábrázolnunk, akkor ott a kimeneti függvény értékek redundánsak lennének.

1.7 Szekvenciális értékadás

Mint említettük a VHDL programozásban, az építmény mindig azonos időben végrehajtott értékadásokból áll. Azonban a párhuzamos események mellett a digitális technikában fontos szerepet töltenek be az események sorozatából álló hálózatok. Ebben az esetben nem lehet azonos időben történő eseményekről beszélni, hiszen egy esemény bekövetkezése függ egy másik esemény bekövetkezésétől és/vagy a rendszer jelenlegi állapotától. Ezért fontos, hogy az azonos időben történő végrehajtási struktúra mellet a VHDL lehetővé teszi a programsorok (értékadási egyenletek) szekvenciális végrehajtását. Ebben az esetben a szekvenciális programrész végrehajtását az állapotváltozást előidéző jelek változása indítja el. Ezt a fajta programstruktúrát a „***process***” (a továbbiakban folyamat) fogja össze. A folyamat végrehajtása a környezetében lévő egyidejű azonosságokkal egyszerre történik, azonban a folyamatban szereplő programsorok az élesítést (az elindító jelek változása) követően, szekvenciálisan történik. A folyamatok leírásában fontos szerepet játszanak az értékadási azonosságok, a feltételes értékadások és a ciklusok kezelése. Ebben a fejezetben ezen VHDL elemeket írjuk le.

Fontos különbséget kell tennünk a „sorrendi értékadási azonosságok” és „a sorrendi hálózatok” között. Az előbbi a folyamat belső állapotait írják le, míg az utóbbi digitális hálózat belső állapotokkal.

1.7.1 Folyamatok

A ***folyamat*** belső állapotaival együtt kombinációs és sorrendi hálózatok leírását valósítja meg [2.]. Egy folyamat általános leírása a következő:

```
-----------------------------------------------------------------------------------------------------------------
process(elesito_jelek)
belső_valtozok_meghatarozasa;
begin
        szekvencialis_allapot_azonossag;
        szekvencialis_allapot_azonossag;
        szekvencialis_allapot_azonossag;
        ....
end process;
-----------------------------------------------------------------------------------------------------------------
```

Figure 13. ábra Process - folyamat

Az élesítő jelek azok a jelek, amelyek elindítják a folyamatot. Az élesítő jelek közül ha legalább egyik jel állapota változik, akkor a folyamat elindul.
A belső változók meghatározása magába foglalja mindazokat a helyi jeleket, objektumokat, amelyeket a folyamat használ és láthatóságuk csak a folyamaton belül biztosított. A „szekvenciális állapot azonosságok” feltételes kiértékelő és értékadó utasítások. Ezek egymásután következő végrehajtásával megtörténik a folyamat belső állapotátmenete. A folyamat, egy digitális áramkör részeként vagy aktív, vagy felfüggesztett állapotban van az élesítő változásától függően.
Egy egyszerű folyamatot mutat be a következő példa ahol a folyamatot élesítő jeleket az építményben határozzuk meg (Figure 14. ábra):

```
-----------------------------------------------------------------------------------------------------------------
entity folyamat is
    Port ( BE1 : in  STD_LOGIC; -- bemeneti jelek BE1, BE2
           BE2 : in  STD_LOGIC;
              FKI : out  STD_LOGIC -- kimeneti jel FKI );

end folyamat;

architecture Behavioral of folyamat is
-- a folyamat bemutatása
signal A,B : STD_LOGIC;
begin
 -- valtoztatjuk a folyamatot elesito jeleket A, B -t.
 A <= BE1 NOR BE2;
 B <= BE1 NAND BE2;
 -- Ha valtozik A vagy B akkor a folyamat kiertekeli
 -- a kimeneti fuggvenyt
 process (A, B) --
 begin
        FKI <= A or B;
 end process;
---- -- pelda vege --
end Behavioral;
 -----------------------------------------------------------------------------------------------------------------
```

Figure 14. ábra Egyszerű példa a folyamat bemutatására

Ha A, vagy B jelek valamelyikének értéke változik (A és B is a BE1, BE2 jelek függvénye), akkor a folyamatban a kiértékelés megtörténik FKI jel értéket kap a folyamat végén.
A folyamatok leírásánál nagyon fontos az érzékenységi jelek megállapítása. Ajánlott az összes olyan jel felvétele az érzékenységi listába, mely valamilyen módon befolyásolja a folyamatot. Például az alábbi folyamatban kihagytuk a „B” és „C” jeleket az érzékenységi listából.

```
---------------------------------------------------------------------------------------------------------------
entity folyamat is
    Port ( A, B, C : in  STD_LOGIC; -- bemeneti jelek A, B, C;
           FKI : out  STD_LOGIC -- kimeneti jel FKI );

end folyamat;

architecture Behavioral of folyamat is
-- Nem teljes az ezrekenysegi lista folyamat bemutatása

begin
 --

 process (A) --
 begin
        FKI <= (A nor B) nand (B nor C);
 end process;
---- -- pelda vege --
end Behavioral;
 ---------------------------------------------------------------------------------------------------------------
```

Figure 15. ábra Hiányos érzékenységi lista

Ebben az esetben a folyamat csak akkor kerül végrehajtásra, amikor az „A” jel változik. Másrészt pedig ha „B” vagy „C” jelek változnak a folyamat felfüggesztett állapotban marad és az „FKI” kimeneti függvény értéke tartja az előző értékét. Úgy is mondhatjuk, hogy a folyamat úgy viselkedik, mint egy tároló elem, amelyet az „A” jel minden változása vezérel. Ez a fajta működés inkább a sorrendi hálózatok működésére jellemző, ezért ha kombinációs hálózatok megvalósítását folyamatokkal szeretnénk megvalósítani akkor ez gondos tervezést igényel.

1.7.2 Folyamatok felfüggesztése „wait” állapotokkal

Az elektronikus rendszerek működését az jellemzi, hogy a feladat elvégzésében mindig ugyanazt a végtelen ciklust hajtják végre. A feladat végrehajtása során a rendszer várakozik bizonyos feltételek teljesülésére ilyenkor felfüggeszti a ciklus végrehajtását, tehát valamilyen feltétel utáni várakozás történik a „folyamat” felfüggesztésekor. Ugyanígy a VHDL nyelvben is a folyamatot felfüggeszti valamilyen feltétel teljesülése utáni várakozás. Ezt a VHDL programban a következő módon fejezzük ki (**Hiba! A hivatkozási forrás nem található.**. ábra)

```
---------------------------------------------------------------------------------------------------------------
architecture Behavioral of folyamat is
-- Folyamat feltételes felfüggesztése

begin
 --

 process () --
 begin
        allapotegyenlet1;
        allapotegyenlet2;
        allapotegyenlet3;
        ...;
        wait <FELTETEL>; -- folymat felfüggesztése
        allapotegyenletk;
        ...;
        allapotegyenletn;

 end process;
---- folyamat vége --
end Behavioral;
 ---------------------------------------------------------------------------------------------------------------
```

16. ábra Folyamat felfüggesztése várakozási feltétellel

A wait feltételes felfüggesztés esetében a folyamatnak nincsenek élesítő jelei.
A következő feltételes felfüggesztési struktúrákat ismerjük:

```
---------------------------------------------------------------------------------------------------------------
wait on jel ;
wait until boole_algebrai_kifejezes;
wait for idozito_kifejezes
---------------------------------------------------------------------------------------------------------------
```

17. ábra „wait” típusok

A wait utasítás használatát legjobban példákon keresztül érthetjük meg. Az Figure 15. ábran lévő példaprogram segítségével szemléltetjük a wait használatát:

```
process
begin
        FKI <= A and B or C;
        wait on A, B , C ;
end proces;
```

18. ábra Wait használatának szemléltesése

Megjegyezzük, hogy a 18. ábran látható folyamat nem tartalmaz élesítő jeleket. A folyamat automatikusan elindul a rendszer tápfeszültségének bekapcsolása után, majd folytatódik a wait programsor végrehajtásával. Azaz a rendszer addig várakozik, míg az A, B, C jelek valamelyike megváltozik. Mihelyt ez megtörténik, a folyamat elindul, megtörténik a folyamat végrehajtása majd visszatér addig míg eléri újból a folyamatot felfüggesztő wait programsor. A wait további két formálya is hasonló módon működik, a különbség csupán a feltételek meghatározásában van. A folyamat felfüggesztése megtörténhet akár a folyamat elején, akár a folyamat végén, agy bárhol a folyamat leírásában.
A „wait until boole_algebrai_kifejezes” addig függeszti fel a folyamat végrehajtását, míg a Boole algebrai kifejezés teljesül. Míg a „wait for idozito_kifejezes” az időzítés lejártáig függeszti fel a folyamatot. A
Következtetés: Mivel a VHDL megengedi többszörös és bonyolult wait feltételek használatát ezért komplex időzítési és sorrendi események modellezésében használhatjuk.

1.7.3 Változók

A VHDL alapvető célja leírni a rendszer viselkedését, azaz a kimenetek hogyan válaszolnak a bemeneti változásokra. Mind a bemeneteket mind a kimeneteket jelekként határozzuk meg. A kimenetek változása az értékadás (<=) során történik. A folyamatokban az értékadás a folyamatok soronkénti végrehajtása során történik. A jelek folyamatban történő használata három fontos szempont figyelembevételével történhet meg:

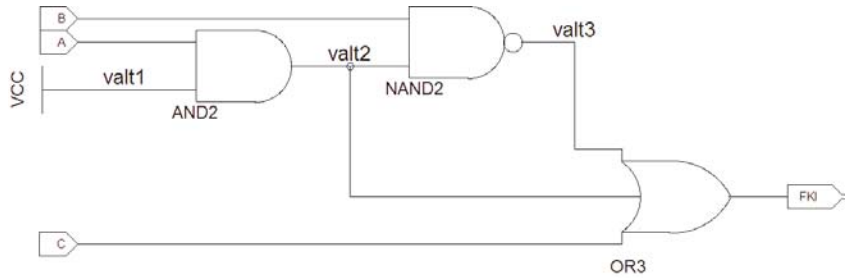
1. A folyamatban nem határozhatunk meg jeleket.
2. A folyamatokban történt értékadások hatására a jelek az új értéket a folyamat befejezésével veszik fel. A folyamat végéig a jelek megőrzik előző értéküket.
3. Többszörös értékadás esetében a jel az utolsó értékadás által adott értéket veszi fel a folyamat végén.

A fenti korlátozásokat figyelembe kell vennünk a gyakorlatban. Az a tény, hogy a folyamat belsejében nem határozhatunk, meg újabb jeleket nem okoz különösebb gondot. Mivel a jelek csupán az utolsó értékadást tárolják, ezért nem használhatunk jeleket köztes értékek tárolására. Másrészt az értékadás nem azonnal történik, hanem a folyamat végén vagy annak felfüggesztésekor. Ez megnehezítheti a rendszer elemzését. Tehát szükséges bevezetnünk egy olyan VHDL elemet, amely az előzőekben elmondottakat feloldja.
A „változó” (variable) meghatározható a folyamat belsejében, értékadáskor azonnal felveszi az új értéket és azt tárolja. A változók hasonlóak a jelekhez, azonban alkalmazásuk kiküszöbölik a jelek használatának korlátait. A jelek nem használhatók a folyamaton kívül, meghatározásuk a **variable** kulcsszó segítségével történik, az értékadás pedig a **:=** szimbólummal. A következő példa a változók használatát mutatja be:

```
process (A, B, C)
variable valt1, valt2, valt3: std_logic;
begin
        valt1 := '1';
        valt2 := valt1 and A;
        valt3 := valt2 nand B;
        FKI <= valt2 or valt3 or C;
end proces;
```

19. ábra Változók használata

A szintézis szempontjából a változókat tekinthetjük úgy is mint belső jeleket. A hardware erőforrások feltérképezésének szempontjából általános használat céljából használjunk jeleket, míg változókat csak olyankor használjunk, amikor a feladat nem megoldható jelekkel.
A 20. ábrán lévő példa kapukkal történő megvalósítása a 19. ábrán lévő példának.



20. ábra Változók, mint belső jelek

1.7.4 IF Feltételes elágazás

Az IF feltételes elágazás formátuma a következő:

```
-------------------------------------------------------------------------------------------------------------
if boole_algebrai_kifejezes0 then
        szekvenciális azonosság1;
        . . .
        szekvenciális azonosság2;
elsif boole_algebrai_kifejezes1 then
        szekvenciális azonosság1;
        . . .
        szekvenciális azonosság2;
elsif boole_algebrai_kifejezes3 then
        szekvenciális azonosság1;
        . . .
        szekvenciális azonosság2;
else

end if;
sequential-statements;

-------------------------------------------------------------------------------------------------------------
```

21. ábra IF feltételes elágazás formátuma

Az **IF** feltételes elágazás - „ha a feltétel igaz, akkor … egyébként pedig … – rendelkezik egy ***„then ággal”*** és esetlegesen egy vagy több ***„elsif ággal”.*** Az egyes ágakban szereplő feltételek kiértékelése Boole igaz/hamis értékkel tér vissza. Az igaz feltételhez tartozó azonosságok szekvenciálisan értékelik ki az azonosságokat. Ha minden kiértékelt ág hamis, és ha létezik „***else***” akkor ezt a részt hajtja végre a hardver megvalósítás.
Következő példánkban a „2-ről 4-re” dekóder áramkört mutatjuk be, amelyet az 1.6.3. fejezetben mutattunk be. Most az „if-then-else” megoldással.

```
-------------------------------------------------------------------------------------------------------------
architecture Behavioral of anod_dekoder is

begin
  d2_4: process (cim)
        begin
                if (cim = "00") then
                        an <= "1110";    -- ha a cím=”00” akkor az AN0 aktív
                elsif cim ="01" then
                        an <= "1101";    -- ha a cím=”01” akkor az AN1 aktív
                elsif cim ="10" then
                        an <= "1011";    -- ha a cím=”10” akkor az AN2 aktív
                else
                        an <= "0111";    -- ha a cím=”11” akkor az AN3 aktív
                end if;
        end process d2_4;

end Behavioral;

-------------------------------------------------------------------------------------------------------------
```

22. ábra „If” struktúra példa: dekóder 2ről 4-re

1.7.5 CASE feltételes elágazás

A CASE feltételes elágazás formátuma a következő:

```
case case_kifejezés is
when lehetoseg_1 =>
        szekvencialis_azonossagok_1;
when lehetoseg_2 =>
        szekvencialis-azonossagok_2;
…
when lehetoseg_n =>
        szekvencialis_azonossagok_n;
when others =>
        szekvencialis_azonossagok_others;
end case ;
```

23. ábra „Case” struktúra

A ***case*** a feltételben szereplő ***case_kifejezes*** értéke alapján kiválasztja a ***szekvencialis_lehetoseg***-nek megfelelő szekvenciális ***azonosságok***-at. A ***case_kifejezes*** értéke a kifejezés értékeinek halmazából azt az értéket választja ki, amely a kiértékelés pillanatában igaznak bizonyul. Az kiértékelés kizárólagos és teljes körű. A ***szekvencialis_lehetoseg_i*** kiértékelés pillanatában érvényesés és hatása a következő kiértékelésig tart. Az ***others*** kulcsszó a lehetőségek halmazából a case feltételben fel nem sorolt értékekhez rendelt ***szekvencialis_azonossagok_others*** kifejezések végrehajtását engedélyezi, amennyiben a ***case_kifejezes*** olyan értéket vesz fel ami az előzően kiértékelt feltételek között nem található meg.

Példaként bemutatjuk a **Hiba! A hivatkozási forrás nem található.**. ábrán bemutatott dekóder áramkör case feltétel kiértékeléssel történő megvalósítást (lásd)

```
begin
    d2_4: process (cim)
        begin
                case (cim) is
                        when "00" =>
                                an <= "1110";    -- ha a cím="00" akkor az AN0 aktív;
                        when "01" =>
                                an <= "1101";    -- ha a cím="01" akkor az AN1 aktív;
                        when "10" =>
                                an <= "1011";    -- ha a cím="10" akkor az AN2 aktív;
                        when "11" =>
                                an <= "0111";    -- ha a cím="11" akkor az AN3 aktív;
                        when others =>
                        an <= "1111";    -- ha a cím= egyéb érték;
                end case;
        end process;
 end Behavioral;
```

24. ábra „Case” struktúrával megvalósított dekóder 2ről 4-re

A dekóder áramkör kimenete „an”, a „cím” négy lehetséges értékére vesz fel kimeneti értékeket. A „cím” változó értékkészlete 81 lehetséges kombinációt vehet fel. Ne feledjük, hogy az előzőekben említett „00”, „01”, „10”, „11” értékeken kívül még 77 másik meta értéket vehet fel a „cím” változó, ezért az „others” kezeli ezeket az állapotokat.

1.8 Kombinációs hálózatok leírása

A továbbiakban olyan tervezési példákat mutatunk be, amelyeket a kombinációs hálózatok témaköréből választottunk. A példák a következők: Összeadó kivonó áramkör, komparátor áramkör, abszolút érték kiszámítása és Barell shifter áramkör. A példák leírása a [2.] szerint történik.

1.8.1 Összeadó- kivonó áramkör

Tervezzük meg azt az aritmetikai áramkört, amely összeadást vagy kivonást egy vezérlő bemenet állapotának függvényében. Amennyiben a vezérlő bemenet „0” logikai értékű úgy az áramkör összeadást végez, míg ha logikai „1” értékű úgy kivonást végez.
Az áramkör kombinációs táblázatát az alábbi táblázat tartalmazza:

Table 4. táblázat Összeadó-kivonó kombinációs táblázata

Vezérlő	A	B	Kimenet
0	A	B	A+B
1	A	B	A-B

Az alábbi egyenlet szemlélteti az áramkör működését:

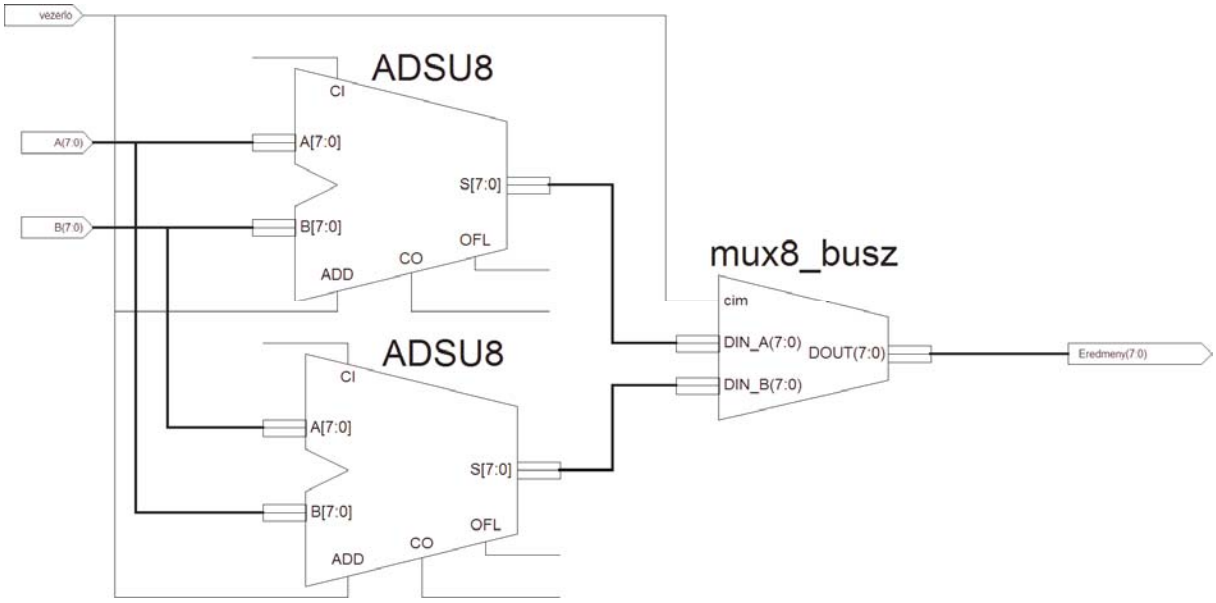
$$eredmey = \begin{cases} A + B\ ha\ vezerlo = 0; \\ A - B\ ha\ vezerlo = 1; \end{cases} \qquad (1)$$

A műveletek elvégzéséhez típuskonverziót kell végrehajtanunk a két bemeneten érkező jelkombinációkon. A bemeneti jelek „*vezerlo*”, amelynek értéke szerint az áramkör elvégzi az összeadást vagy a kivonást; „*A*” és „*B*” a két operandus; kimeneti jel: „*eredmeny”*. A példa VHDL programja az

```
----------------------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity add_sub is
   Port ( vezerlo : in  STD_LOGIC; -- vezérlő bemenet
          A : in  STD_LOGIC_VECTOR (7 downto 0);
          B : in  STD_LOGIC_VECTOR (7 downto 0);
          eredmeny : out  STD_LOGIC_VECTOR (7 downto 0) );
                              -- eredmény a vezerlo fuggvenyeben eredmeny=A+B vagy
eredmeny=A-B
end add_sub;

architecture Behavioral of add_sub is
signal op0, opl, sum: signed(7 downto 0);
begin
          -- típuskonverzió vektor => előjellel rendelkező szám
          op0 <= signed(A);
          op1 <= signed(B);
          -- ha vezerlo = 0 akor osszeadas
          sum <= op0 + op1 when vezerlo='0' else
          -- ha vezerlo = 0 akor osszeadas
                    op0 - op1;
          -- az eredmény átalakítáasa vektorra
          -- típuskonverzió
  eredmeny <= std_logic_vector(sum);
end Behavioral;
----------------------------------------------------------------------------------------------------------------
```

25. ábra Összeadó-kivonó áramkör VHDL nyelvű programja

Az összeadás elvégzéséhez típuskonverziót kell végeznünk ahhoz, hogy a matematikai műveletek helyesen (előjelhelyesen) értelmezzék az összeadást/kivonást. Az áramkör kapcsolási rajza a következő:



26. ábra Összeadó- Kivonó Áramkör kapcsolási rajza

A fenti példa nem tartalmazza a továbbvitel bitkezelését, hiszen a típuskonverzió révén ezt a VHDL nyelv megoldja, és szintéziskor megfelelően kezeli. Azonban az alábbiakban kiegészítjük a példánkat úgy, a kivonáshoz is és az összeadáshoz is ugyanazt az egyenletet használjuk (*sum <= op0+op1+cin*) végeztetjük el, azonban ebben az esetben a kivonáshoz szükséges kettős komplemens előállítását is meg kell valósítanunk. A művelet elvégzéséhez kiegészítjük mindkét bemeneti vektort egy-egy bittel (összefűzés) [52]. Az A vektort összefűzzük egy ’1’ értékű bittel. Ily módon a *A(0)<=1* értékű lesz. Az alulról jövő továbbvitel bitet pedig a B vektorhoz fűzzük (27. Ábra).

```
---------------------------------------------------------------------------------------------------------------

architecture Behavioral of add_sub_cy is
signal op0, opl, sum: signed(8 downto 0);
Signal b_koz: signed(7 downto 0);
signal cin: signed(0 downto 0);

begin
        -- op1 8 bites előjelse szám = "a7_a6_a5_a4_a3_a2_a1_a0_1"
        op0 <= signed(A & '1');
        -- b_koz vagy 7 bites előjeles szám B vagy
        -- kivonás esetén B 1-es komplemense (not B)
        b_koz <= signed(B) when ctrl='O' else
                        signed (not B) ;

        cin <= "0" when ctrl='0' else
                        "1";
        op1 <= signed(b_koz & cin);
        sum <= op0 + op1 + cin;
        Eredmeny <= std_logic_vector(sum (8 downto 1));
end Behavioral;
---------------------------------------------------------------------------------------------------------------
```

27. ábra Összeadó-kivonó áramkör VHDL nyelvű programja

Ebben az esetben az összeadó-kivonó áramkör bemenő jeleit (operandusait op1 és op2) előjeles számként kezeljük, ezért kivonás művelet végzése esetében a kettős komplemens képzése b_koz jel segítségével és a az alulról jövő túlcsordulás jel segítségével képezzük.

1.8.2 Komparátor áramkör

A komparátor áramkör példáját előjel nélküli, nyolc bites vektorokra mutatjuk be. Az áramkörnek két 8 bites bemenete van: A(7 donwto 0) és B(7 downto 0). A áramkör kimenetei AnagyobbB, AegyenloB, AkisebbB. A komparátor VHDL programját az ábrán mutatjuk be.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity kompartor is
   Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
          B : in  STD_LOGIC_VECTOR (7 downto 0);
          AkisebbB : out  STD_LOGIC;
          AegyenloB : out  STD_LOGIC;
          AnagyobbB : out  STD_LOGIC);
end kompartor;

architecture Behavioral of kompartor is

begin
  AkisebbB <=     '1' when  A < B else
                  '0'; -- A kisebb mint B
  AegyenloB <=   '1' when  A = B else
                  '0'; -- A egyenlő B
  AnagyobbB <= '1' when  A > B else
                  '0'; -- A nagyobb B

end Behavioral;
```

28. ábra Komparátor áramkör VHDL nyelvű programja

A komparátor három kimenete AkisebbB, AegyenloB, AnagyobbB jelzik a két bemenet közötti relációt (kisebb, egyenlő, nagyobb). Az komparátor megvalósítás három relációt megvalósító áramkörből jön létre. Az első két relációból következik a harmadik, így egyszerűsíthető a VHDL program. Az egyszerűsített programot a következő ábra szemlélteti.

A komparátor három kimenete AkisebbB, AegyenloB, AnagyobbB jelzik a két bemenet közötti relációt (kisebb, egyenlő, nagyobb). Az komparátor megvalósítás három relációt megvalósító áramkörből jön létre. Az első két relációból következik a harmadik, így egyszerűsíthető a VHDL program. Az egyszerűsített programot a következő ábra szemlélteti.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity kompartor_uj is
   Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
          B : in  STD_LOGIC_VECTOR (7 downto 0);
          AkisebbB : out  STD_LOGIC;
          AegyenloB : out  STD_LOGIC;
          AnagyobbB : out  STD_LOGIC);
end kompartor;

architecture Behavioral of kompartor_uj is
signal nagyobb, kisebb: std_logic;
begin
  kisebb    <=      '1' when  A < B else
                    '0'; -- A kisebb mint B
  nagyobb <= '1' when  A > B else
                    '0'; -- A nagyobb B
AkisebbB  <= kisebb;
AnagyobbB <= nagyobb;
AegyenloB <= not (kisebb or nagyobb);
---- Az első két feltétel kiértékeléséből következik az egyenlőség
---- Ha nem A<B vagy nem A>B => A=B;

end Behavioral;
```

29. ábra Komparátor áramkör VHDL nyelvű programja

Bár számunkra az egyszerűsítés nyilvánvaló, a legtöbb fordító program számára ez nem egyértelmű és az egyszerűsítés előnyét nem tudja kihasználni.

1.8.3 Barrel shifter (eltoló) áramkör

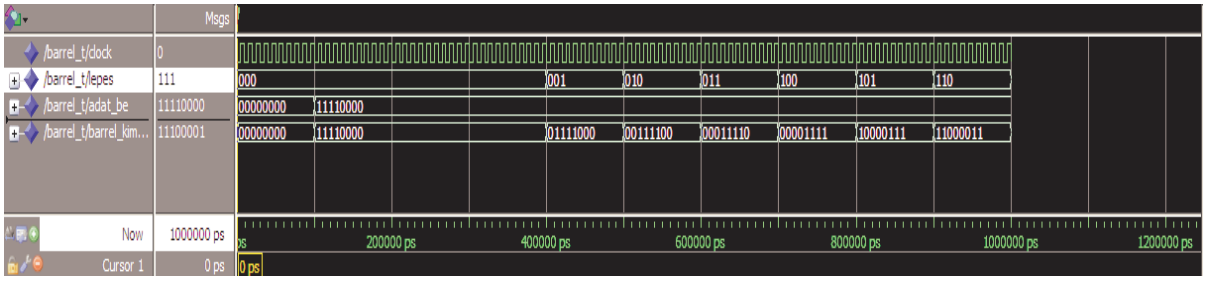
Igaz ugyan, hogy a regiszter áramkörök a sorrendi hálózatokhoz tartoznak, azonban a barrel (hordó) eltolást végző áramkör jellegzetessége miatt inkább egy kombinációs áramköri megvalósítási példát adunk meg.

A barrel shifter áramkör jellegzetessége, hogy a bemeneti adatokat, a kimeneten eltolja adott számú bittel (jobbra vagy balra), az eltolás/forgatás értékét a címző bemenet adja. A VHDL nyelv szabványos könyvtára (IEEE_std_logic_1164) lehetővé teszi különböző shift és forgató operátorok használatát. A „shift" műveletek logikai és aritmetikai eltolást illetve jobbra/balra forgatást tesznek lehetővé. Ebben a példában egy nyolcbites forgató áramkört ismertetünk, amely logikai/aritmetikai jobbra tolást tesz lehetővé. A vezérlő bemenetek a jobbra vagy balra történő forgatást (vezerlo) és az eltolás nagyságát (azaz hány bittel történik az eltolás -- lepes) teszik lehetővé [2.].

A példa úgy valósítja meg a barrel forgatást, hogy összefűzi a bemenet megfelelő súlyozású bitjeit. A VHDL program az 30 ábrán látható. A feladat szimulációja pedig a 31. ábrán látható.

```
----------------------------------------------------------------------------------------------------------------
	library IEEE;
	use IEEE.STD_LOGIC_1164.ALL;

	entity barrel is
	   Port ( adat_be : in  STD_LOGIC_VECTOR (7 downto 0);
	                  -- nyolbites adatbemenet;
	          -- irany : in  STD_LOGIC;
	                  -- a forgatás iránya jobbra ha irany = 0;
	                  -- illetve                    balra  ha irany = 1;
	          lepes : in  STD_LOGIC_VECTOR (2 downto 0);
	                  -- fogratás mennyisége
	          barrel_kimenet : out  STD_LOGIC_VECTOR (7 downto 0));
	                  -- kimenet
	end barrel;

	architecture Behavioral of barrel is

	begin
	with lepes select
	  barrel_kimenet <=     adat_be                                  when "000",     -- nulla bites forgatás
	                        adat_be(0)& adat_be(7 downto 1)          when "001",     -- egy bites forgatás
	                        adat_be(1 downto 0) & adat_be(7 downto 2) when "010",    -- kettő bites forgatás
	                        adat_be(2 downto 0) & adat_be(7 downto 3) when "011",    -- három bites forgatás
	                        adat_be(3 downto 0) & adat_be(7 downto 4) when "100",    -- négy bites forgatás
	                        adat_be(4 downto 0) & adat_be(7 downto 5) when "101",    -- öt bites forgatás
	                        adat_be(5 downto 0) & adat_be(7 downto 6) when "110",    -- hat bites forgatás
	                        adat_be(6 downto 0) & adat_be(7) when others ;           -- hét bites forgatás

	end Behavioral;end Behavioral;
----------------------------------------------------------------------------------------------------------------
```

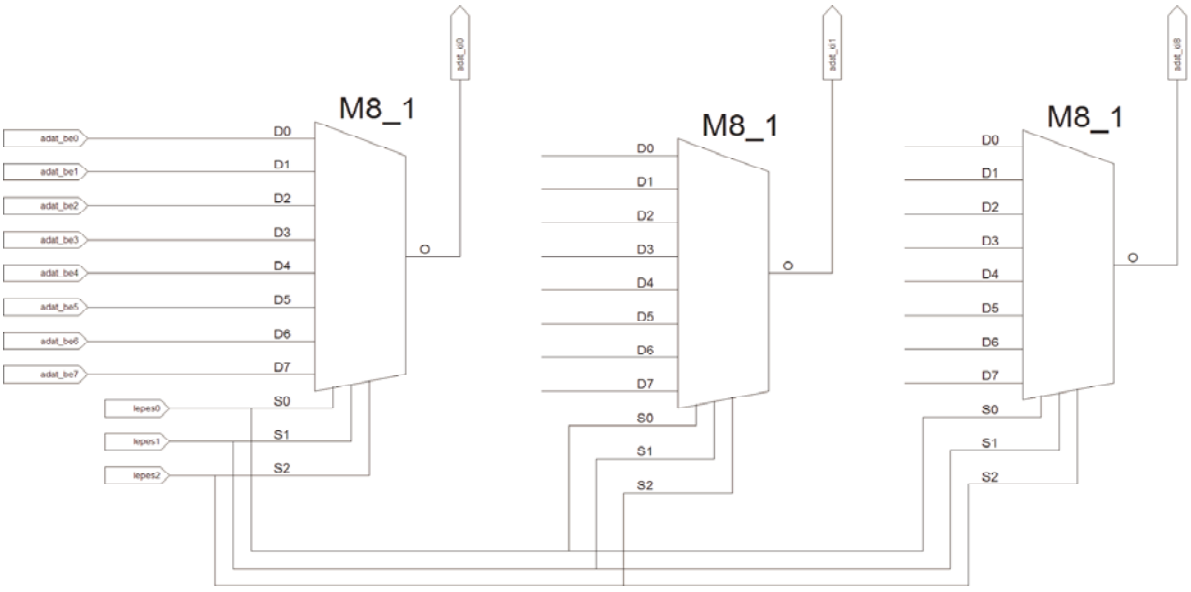
30. ábra Barrel shifter VHDL nyelvű programja



31. ábra Barrel shifter szimulációja

A szimuláción jól látható, hogy kimenet (barrel_kimenet) értéke a lepes változó értéke szerint veszi fel a bemenet értékeit. Azaz a kimeneteket annyi bittel forgatja a bemenetekhez képest, amennyi a lepes értéke. Amennyiben kapcsolási rajzot kell létrehoznunk akkor egy másik

megvalósítását kapjuk a Barrel shifter áramkörnek, amelynek tervezését nem mutatjuk be, de szemléltetjük a



32. ábra Barrel shifter megvalósítása multiplexer áramkörökkel.

An alternative design is to do the rotating in levels, as shown in Figure 7.12(a). In each level, a bit of the **amt** signal indicates whether the input is passed directly to the output or rotated by a fixed amount.

1.8.4 Prioritás kódoló áramkör 4to2

A prioritás kódoló áramkör felépítését és működését a ?? fejezetben ismertettük. Csupán az ismétlés kedvéért: az áramkör a bemeneten megjelenő legnagyobb prioritású kérést továbbítja a kimenetre. A feltételes jelkiértékelés a legmegfelelőbb az áramkör függvényének leírására. Lévén, az áramkör egy kaszkád rendszerben megvalósuló huzalozás (when-else-when-else… lánc) ezért a bemenetek növekedésével az áramkör működési sebessége csökkenhet. Az áramkör megvalósítása a 33. ábrán látható.

```
----------------------------------------------------------------------------------------------------------------
entity prioritas is
   Port ( negy_be : in  STD_LOGIC_VECTOR (3 downto 0);
          kod_4_2_ki : out  STD_LOGIC_VECTOR (1 downto 0);
          jelzes : out  STD_LOGIC);
end prioritas;

architecture Behavioral of prioritas is

begin
          kod_4_2_ki <= "11" when negy_be (3)='1' else
                        "10" when negy_be (2)='1' else
                        "01" when negy_be (1)='1' else
                        "00";
          jelzes <= negy_be (3) or negy_be (2) or negy_be (1) or negy_be (0);

end Behavioral;
----------------------------------------------------------------------------------------------------------------
```

33. ábra 4-2 prioritást kódoló áramkör

Az áramkör szimulációjától eltekintünk.

1.8.5 Bináris-Gray kódátalakító áramkör

A Gray kód esetében két egymást követő kódszó között egyetlen bit változik (a Hamming távolság egy). Ezáltal a bitátmenetek két egymásután következő kódszónál minimálisra

csökkennek. A Bináris-Gray kód megfeleltetést az 5. Táblázat tartalmazza. Amennyiben a táblázat szerint valósítjuk meg a Bináris-Gray kódátalakító áramkört, úgy a VHDL programunk egyszerű, bár nem túl elegáns. Ezt a megoldást tartalmazza a

5. táblázat Bináris-Gray kódátalakító kombinációs táblázata

mi	Bináris kód	B3	B2	B1	B0	Gray kód	G3	G2	G1	G0
0	**0**	0	0	0	0	**0**	0	0	0	0
1	**1**	0	0	0	1	**1**	0	0	0	1
2	**2**	0	0	1	0	**2**	0	0	1	1
3	**3**	0	0	1	1	**3**	0	0	1	0
4	**4**	0	1	0	0	**4**	0	1	1	0
5	**5**	0	1	0	1	**5**	0	1	1	1
6	**6**	0	1	1	0	**6**	0	1	0	1
7	**7**	0	1	1	1	**7**	0	1	0	0
8	**8**	1	0	0	0	**8**	1	1	0	0
9	**9**	1	0	0	1	**9**	1	1	0	1
10	**10**	1	0	1	0	**10**	1	1	1	1
11	**11**	1	0	1	1	**11**	1	1	1	0
12	**12**	1	1	0	0	**12**	1	0	1	0
13	**13**	1	1	0	1	**13**	1	0	1	1
14	**14**	1	1	1	0	**14**	1	0	0	1
15	**15**	1	1	1	1	**15**	1	0	0	0

```
----------------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity bin_gray is
    Port ( -- binaris - bemenet 4 bites
           binaris : in  STD_LOGIC_VECTOR (3 downto 0);
           -- gray kimenet 4 bites
           gray : out  STD_LOGIC_VECTOR (3 downto 0));
end bin_gray;

architecture Behavioral of bin_gray is

begin
        with binaris SELect
                gray <= "0001"  when "0001",    --          gray 1-es kimenet
                        "0011" when "0010",     --          gray 2-es kimenet
                        "0010" when "0011",     --          gray 3-es kimenet
                        "0110" when "0100",     --          gray 4-es kimenet
                        "0111" when "0101",     --          gray 5-es kimenet
                        "0101" when "0110",     --          gray 6-es kimenet
                        "0100" when "0111",     --          gray 7-es kimenet
                        "1100" when "1000",     --          gray 8-es kimenet
                        "1101" when "1001",     --          gray 9-es kimenet
                        "1111" when "1010",     --         gray 10-es kimenet
                        "1110" when "1011",     --         gray 11-es kimenet
                        "1010" when "1100",     --         gray 12-es kimenet
                        "1011" when "1101",     --         gray 13-es kimenet
                        "1001" when "1110",     --         gray 14-es kimenet
                        "1000" when "1111",     --         gray 15-es kimenet
                        "0000" when others;     --          gray 0-ás kimenet

end Behavioral;
----------------------------------------------------------------------------------------------------------
```

34. ábra Bináris-Gray kódátalakító áramkör

Egy rövidebb és egyszerűbb VHDL programot eredményez a bináris-Gray kódátalakító áramkör megvalósítása, amennyiben a Gray- bináris kód közötti egyszerű összefügéseket használjuk. Amit az alábbi egyenletekkel adhatunk meg:

$$G_3 = B_3 \oplus 0; \quad (4)$$

$$G_2 = B_3 \oplus B_2; \quad (5)$$

$$G_1 = B_2 \oplus B_1; \quad (6)$$

$$G_0 = B_1 \oplus B_0; \quad (7)$$

A fenti egyenletek alapján létrehozott VHDL program a 35. ábra szemléltet.

```
----------------------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity bin_gray is
   Port ( -- binaris - bemenet 4 bites
          binaris : in  STD_LOGIC_VECTOR (3 downto 0);
          -- gray kimenet 4 bites
          gray : out  STD_LOGIC_VECTOR (3 downto 0));
end bin_gray;

architecture Behavioral of bin_gray is

begin
        ---  egyszerübb megoldás az egyeneletek felhasználásával
        gray(3) <= binaris(3) xor '0';          -- harmadik bit
        gray(2) <= binaris(3) xor binaris(2);   -- masodik bi
        gray(1) <= binaris(2) xor binaris(1);   -- elso bit
        gray(0) <= binaris(1) xor binaris(0);   -- nulladik bit
end Behavioral;
----------------------------------------------------------------------------------------------------------------
```

35. ábra Bináris-Gray kódátalakító áramkör

A bináris-gray kódátalakító áramkör két VHDL példán keresztül történő megvalósításával azt szeretnénk szemléltetni, hogy míg az első példa szerinti (34. ábra) megvalósítás nem általánosítható, addig a második megvalósítás (35. ábra) kiterjeszthető akár több bites kódátalakító áramkör megvalósítására, hiszen a G_{i-1} bitje kiszámítható a B_{i-1} és a B_i bitek antivalencia kapcsolatából:

$$G_{i-1} = B_i \oplus B_{i-1}; \quad (8)$$

A fenti néhány példán keresztül szemléltettük a kombinációs hálózatok megvalósítását VHDL hardverleíró nyelv segítségével. Összegezésként elmondható, hogy a kombinációs hálózatok megvalósításában leggyakrabban használt VHDL struktúra a „with – SELect – when" feltételes vizsgálat, vagy a másik gyakran használt struktúra a logikai egyenletekkel történő megvalósítás.

1.9 Sorrendi hálózatok leírása VHDL

Ebben a fejezetben a sorrendi (szekvenciális) hálózatok VHDL nyelven történő megvalósítására adunk néhány példát. Először egy flip-flopot modellezünk, majd számláló egységet, ily módon az egyszerű áramköröktől a bonyolultabbak felé.

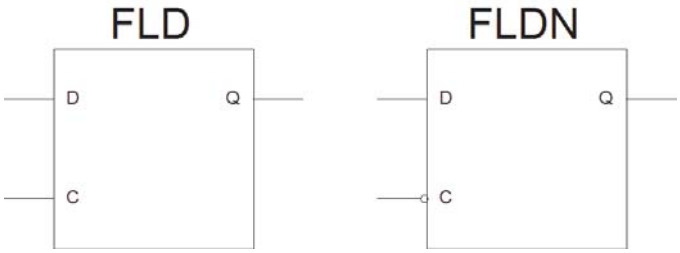
A sorrendi hálózatok elméletével megismerkedtünk az előző fejezetekben ezért most csak az ide vonatkozó elméleti elemeket említjük. Egy sorrendi hálózat rendelkezik belső állapotokkal (átmeneti vagy állandósult) és/vagy belső memóriával. A programozható logikai áramkörök előnyben részesítik a szinkron sorrendi hálózatokat, azaz a rendszer órajel ellenőrzi a belső állapotok/memória vezérlését. A kombinációs áramkörökkel ellentétben a sorrendi hálózatok kimeneti állapota függ a bemenetek jelenlegi állapotától és a belső állapotoktól. Ily módon a rendszer kimeneti jelei a bemeneti jelek jelenlegi és előző állapotának függvényei. Ezért nevezünk egy áramkört, amely belső állapotokkal rendelkezik sorrendi hálózatnak.

A sorrendi hálózatok alapvető építőelemei a flip-flopok és a tárolok. Az alábbiakban a D típusú tároló és flip-flopok közötti különbséget tisztázzuk.

1.9.1 D tároló

Digitális áramkörök esetében két módon tárolhatjuk az információt. Az egyik tárolási mód memória áramkör alkalmazása (kombinációs hálózat), a másik tárolók/flip-flop alkalmazása. A tároló elem kapcsolási rajz szimbólumai a 36. ábra mutatja. Az FLD szimbólum logikai 1 szintre írja be a q

kimenetre a D bemeneten lévő jel állapotát, míg az FLDN szimbólum logikai 0 szintre írja be a jel állapotát. Mivel az állapotváltozás az órajel szintjétől függ ezért azt mondjuk, hogy a D tároló szintézékeny.



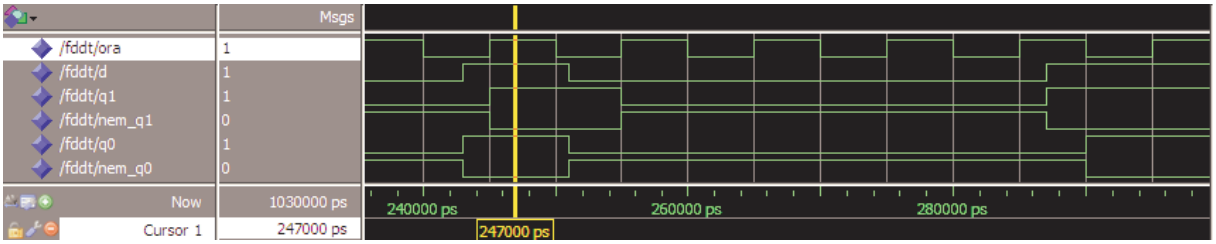
37. ábra Egyre (FLD) és nullára (FLDN) beíró tároló (latch)

A D tárolót VHDL nyelven modellezzük. Mindkét tárolót egyetlen VHDL programmal valósítjuk meg, oly módon, hogy egy-egy folyamat valósítja meg az FLD illetve FLDN tárolót. Az órajel „óra”, a bemenet d, a kimenet pedig q. A „0” és az „1” jelőlik az FLD és az FLDN-hez tartozó bemeneteket és kimeneteket (38. ábra). A D tároló szimulációjánál (39. ábra) jól látható, hogy az állapotváltozások valóban csakis akkor következnek be amikor az „ora” jel 0/1 szintet vált. A szimuláció megvalósításában a D bemeneti jel állapotváltozásait a következő VHDL programsorral adtuk meg:

d <= '0', '1' after 17 ns, '0' after 22 ns, '1' after 43 ns, d <= '0', '1' after 17 ns, '1' after 43 ns,'0' after 22 ns, '1' after 51 ns, '0' after 74 ns, '1' after 87 ns; (2.)

```
----------------------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity fdl is
    Port ( d : in  STD_LOGIC;                -- d tároló bemenet
           ora : in  STD_LOGIC;              -- órajel
                                  q1 : out  STD_LOGIC;                    -- 1-re billenő kimenet
                                  nem_q1 : out  STD_LOGIC; -- 1-re billenő "nem" kimenet
                                  q0 : out  STD_LOGIC;             -- 0-ra billenő kimenet
                                  nem_q0 : out  STD_LOGIC);-- 1-re billenő "nem" kimenet
end fdl;

architecture Behavioral of fdl is

begin
        fddl: process (ora,d)
        -- 1 logikai szintre érzékeny tároló
        begin
                -- ha a ora=1 akkor beírás
                if (ora = '1') then
                        q1 <= d;   --after 5 ns;
                        nem_q1 <= not d;--after 5 ns;
                end if;
        end process;
        -- 0 logikai szintre érzékeny tároló
        fdln: process (ora,d)
        begin
                -- ha a ora=1 akkor beírás
                if (ora = '0') then
                        q0 <= d;  --after 5 ns;
                        nem_q0 <= not d; --after 5 ns;
                end if;
        end process;
end Behavioral;
----------------------------------------------------------------------------------------------------------------
```

38. ábra D tároló VHDL megvalósítása

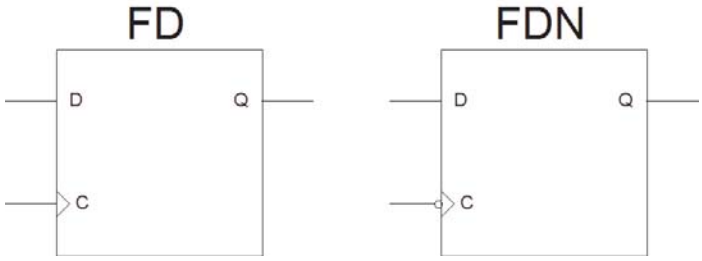


39. ábra D tároló szimulációja

A 39. ábrán lévő idődiagramon jól látható, hogy a tárolók a vezérlő jel hatására jelenitik meg a kimeneten a bemeneti adat állapotát. Például a 247ns időpillanatban mutatott állapot esetében a q0 jel már az ora ’0’ szintjénél azonnal megjeleníti a d=’1’ állapotot, míg a q1 csakis az ora=’1’ jelszint megjelenésekor.

1.9.2 D flip-flop

A 40. ábrán ábrázolt flip-flop áramkörök ellentétben a D-tárolóval nem az órajel szintjére vezérlik a flip-flop-ot, hanem a jelváltozásokra azaz a D-flip-flop élvezérelt áramkör. A pozitív élvezérelt flip-flop-ot a $0 \rightarrow 1$ átmenet, míg a negatív élvezérlelt flip-flop-ot az $1 \rightarrow 0$ átmenet vezérli. A vezérlő átmenet kivételével a flip-flop kimenete állandó marad azaz tartja az információt a következő vezérlő átmenetig. Úgy is mondhatjuk, hogy a vezérlő jel hatására a flip-flop mintavételezi a bemeneti jelet. A kapcsolási rajz ábrázolásában az órajelnél „>” jellel jelöljük azt, hogy az áramkör élvezérelt.

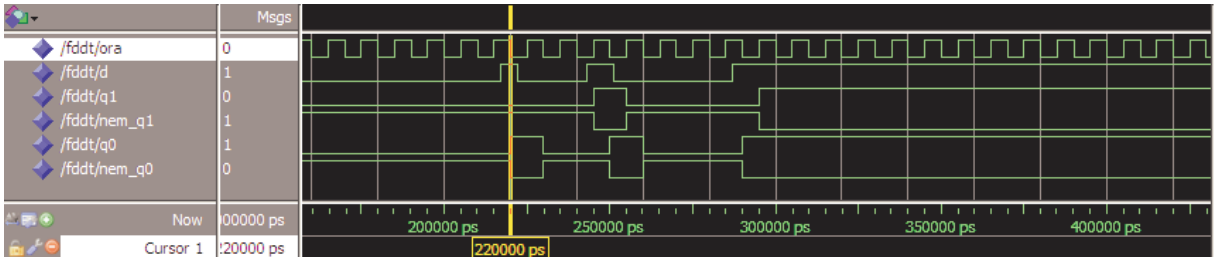


40. ábra D flip-flop

A D-flip-flop VHDL kodú modelezése a 41. ábrán látható. Az építmény leírásában megfigyelhető, hogy mind a felfutó, mind a lefutó óraél megvalósítására két módszert adtunk meg. Mind a kettő helyes.
Azaz a felfutó/pozitív óraél érzékelése leírható a következő módon:
(ora'event and ora = '1') vagy
(rising_edge(ora)).
Míg a lefutó/negatív óraél leírása pedig a következő:
(ora'event and ora= '0') vagy
(falling_edge(ora)).
Mindkét leírási mód használatos.
A D flip-flop-ok működésének szemléltetését a 42. ábra mutatja be. Az ábrán megfigyelhető, hogy a kimenetek (q0, q1) változása csakis órajel élváltozásra (pozitív, negativ) történik.

```
----------------------------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity fdl is
    Port ( d : in  STD_LOGIC;                -- d tároló bemenet
           ora : in  STD_LOGIC;              -- órajel
           q1 : out  STD_LOGIC;              -- felfutó élre-re billenő kimenet
           nem_q1 : out  STD_LOGIC; -- felfutó élre billenő "nem" kimenet
           q0 : out  STD_LOGIC;   -- lefutó élre -re billenő kimenet
           nem_q0 : out  STD_LOGIC);-- lefutó élre -re billenő "nem" kimenet
end fdl;

architecture Behavioral of fdl is

begin
           fddflip: process (ora,d)
           -- pozítiv élvezérelt flip-flop
           begin
                      -- ha a felfutó oraél van  => akkor beírás
                      if (ora'event and ora = '1') then
                      -- masik lehetőség:
                      -- felfutó óraél érzékelésére
                      -- if (rising_edge(ora)) then
                                 q1 <= d;   --after 5 ns;
                                 nem_q1 <= not d;--after 5 ns;
                      end if;
           end process;
           -- negativ élvezérelt flip-flop
           fddflop: process (ora,d)
           begin
                      -- ha a lefutó oraél => akkor beírás
                      if (falling_edge(ora)) then
                      -- másik lehetőség:
                      -- lefutó óraél érzékelésére
                      -- if ora'event and ora= '0' then
                                 q0 <= d;  --after 5 ns;
                                 nem_q0 <= not d; --after 5 ns;
                      end if;
           end process;
```

41. ábra D flip-flop VHDL megvalósítása



42. ábra D flip-flop VHDL Szimulációja

A flip-flop áramkörök előnyét a tárolókkal szemben jól szemlélteti a fenti ábra. Azaz a flip-flop áramkör kiküszöböli a bemeneten megjelenő zavaró jeleket, ezáltal csakis a bemenet állandósult állapotai kerülnek a kimenetre. Egy másik előny pedig a versenyhelyzetek kiküszöbölése egy esetleges visszacsatolás esetében.

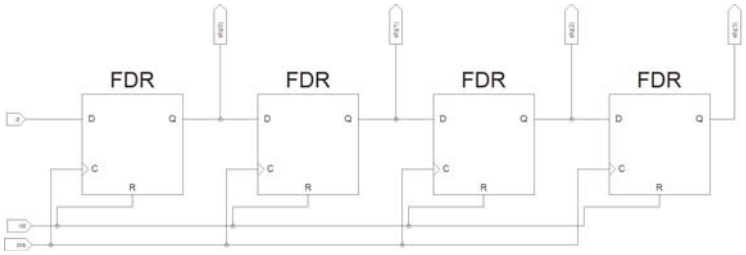
1.9.3 Shift regiszter

Bár a VHDL lehetővé teszi a digitális áramkörök felépítését alapelemekből (logikai kapukból és flip-flop-okból), mégsem ez a megszokott tervezési eljárás. Példánk azonban szemléltetni kiván egy jobbra shiftelő 4 bites regiszter megvalósítását FD áramkörből felépítve. A példa megvalósításában felhasználjuk az előző fejeztben leírt flip-flop-ot és kiegészítjük oly módon, hogy tudjuk az áramkört alaphelyzetbe állítani, azaz a bemeneti jeleket kiegészítjük a törlő (reset) vezélrő jellel. Ily módon a D flip-flop VHDL leírása a következő lesz:

```
----------------------------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity fdd is
    Port ( d :      in  STD_LOGIC;                  -- d tároló bemenet
           ora :    in  STD_LOGIC;                  -- órajel
           rst:     in STD_LOGIC;                   -- törlő bemenet
           q :      out  STD_LOGIC                  -- felfutó élre-re billenő kimenet)
           );
end fdd;

architecture Behavioral of fdl is

begin
        fddflip: process (ora, rst)
        -- pozítiv élvezérelt flip-flop
        begin
                --ha rst = 1 => a kimenet alapállapotba
                if rst='1' then
                        q <= '0';
                elsif (ora'event and ora='0') then
                        q <= d;
                end if;
        end process fdd;
end Behavioral;
----------------------------------------------------------------------------------------------------------------------
```

43. ábra FDR flip-flop Törlő bemenettel

A shift regisztert az négy darab FDR flip-flop-ból valósítjuk meg, ahogyan azt az alábbi kapcsolási rajz mutatja. Az egyes kimeneteket összekötjük a következő FDR bemenettel, órajel és a reset jel közös. A shift regisztert megvalósító VHDL program a 45. ábrán látható.



44. ábra 4 bites shift regiszter kapcsolási rajza

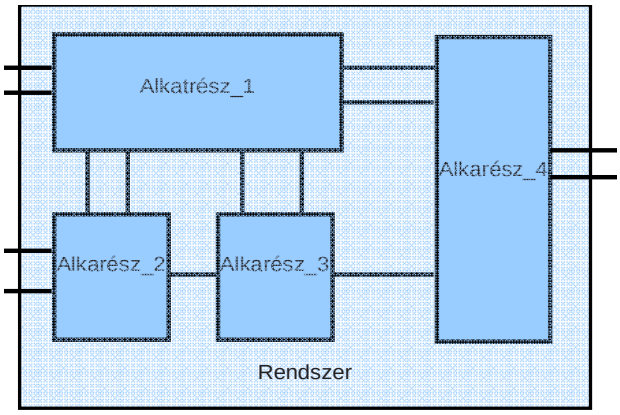
```
----------------------------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity shiftreg is
    Port ( d : in  STD_LOGIC;
           ora : in  STD_LOGIC;
           rst : in  STD_LOGIC;
           sfq : out  STD_LOGIC_VECTOR (3 downto 0));
end shiftreg;

architecture Behavioral of shiftreg is
-- az FD mint alkatrész meghatározása
                component fdd
                port (d : in  STD_LOGIC;
                                ora : in  STD_LOGIC;
                                rst : in  STD_LOGIC;
                                q : out  STD_LOGIC);
                end component;
signal fqd : STD_LOGIC_VECTOR (3 downto 0);
begin

fd0 : fdd
        port map (
                d => d,
                ora => ora,
                rst => rst,
                q => fqd(0)
        );
fd1 : fdd
        port map (
                d => fqd(0),
                ora => ora,
                rst => rst,
                q => fqd(1)
        );
fd2 : fdd
        port map (
                d => fqd(1),
                ora => ora,
                rst => rst,
                q => fqd(2)
        );
fd3 : fdd
        port map (
                d => fqd(2),
                ora => ora,
                rst => rst,
                q => fqd(3)
        );
sfq <=fqd;
end Behavioral;
----------------------------------------------------------------------------------------------------------------------
```

45. ábra 4 bites shift regiszter megvalósítása D-flip-flop-okkal

A négy bites shift regiszter megvalósítása esetében látható, hogy az FD elemet mint VHDL alkatrészt illesztettük be. Ezáltal növeltük a VHDL kód olvashatóságát és a programban az FD elemhez tartozó csatlakozásokat kell megadnunk az adott helyen. Ezzel egy új VHDL fogalmat vezettünk be az alkatrész fogalmát, amely a digitális rendszert felépítése (struktúrája) és az elemek egymás közötti kapcsolatai szempontjából írja le.

Strukturális leírás: Alkatrész (Component)

A digitális rendszer szerkezetének leírása azt jelenti, hogy a VHDL programban leírjuk azt, hogy a rendszer milyen alkatrészekből áll és az alkatrészek milyen kapcsolatban vannak egymással. A szerkezeti leírás lehetővé teszi az alkatrész használatával a többszintű (hierarchikus) tervezés alkalmazását a VHDL programban. Az fő program alrendszereiben is használhatunk alkatrészeket amelyek még egyszerűbb függvényeket valósítanak meg. A legalsó szinten a VHDL program leírja az alkatrészt viselkedése szempontjából. Az alkatrészeket a VHDL programban jelekkel kötjük össze (46. ábra).



46. ábra Szerkezeti leírás

A szerkezeti leíráshoz két VHDL elemre van szükségünk. Az egyik elem az alkatrész mint önálló egyed (VHDL program), amely rendelkezik önálló entitással és architektúrával. A másik elem az alkatrész beillesztése az architektúrába a ***component*** kulcsszóval. Mint könyvtári elemet, az alkatrészt meghívása után bele kell illesztenünk a rendszer felépítésébe, ezért le kell írnunk a többi az architektúra jeleival való viszonyát. Azaz meg kell adnunk az alkatrész ***port***jának jelei és a felsőbb szintű rendszer jelei közötti kapcsolatokat (component instatiation). Az alkatrész beillesztésének lépései:

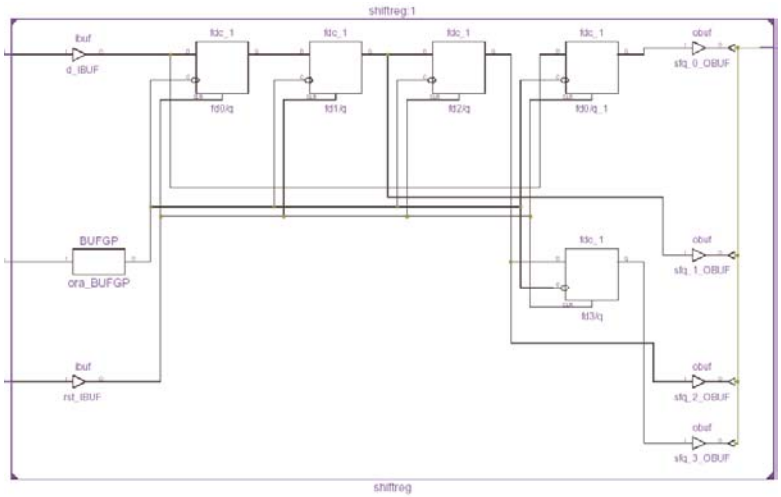
1. Az alkatrész entitásának és architektúrájának leírása (VHDL program).
2. Az alkatrész beillesztése component kulcsszóval az architektúrába:

```
alkatrszesz: component <alkatresz_nev>
        generic (
                <generic_neve> : <tipus> := <erteke>;
                <tovabbi_generic_tipusok>...
            );
        port (
                <port_nev> : <port_irany> <tipus>;
                <tovabbi_portok>...
              );
    end component;
```

3. Az alkatrész és a jelek kapcsolatának meghatározása:

```
<alkatresz_hivatokozas_cimke> : <alakatresz_nev>
        generic map (
                <generic_nev> => <ertek>,
                < tovabbi_generic>...
                    )
        port map (
                    <port_nev> => <jel_nev>,
                    <tovabbi_portok>...
                );
```

Visszatérve a shiftregiszterre a VHDL program szemlélteti a fent leírt lépéseket. Létrehoztuk az fdd: D flip-flop alkatrészt, beillesztettük a shift regiszterbe component kulcsszóval, es mind a négy esetben megadtuk azoknak a jeleknek a nevét melyekhez csatlakozik az alkatrész. A szintézis után pedig a rendszer az alábbi shift regiszter kapcsolást hozta létre (47. ábra):



47. ábra 4 bites shift regiszter kapcsolási rajza szintézis után

Az ábrán megfigyelhető, hogy az fd0 elemet kétszer illesztette be fd0/q és fd0/q_1 néven. Mivel a szintézis stratégia beállítása egy kiegyensúlyozott rendszert hoz létre (otpimális hardver, optimális működési sebesség és optimális disszipált teljesítmény), ezért a kétszeres beillesztés.

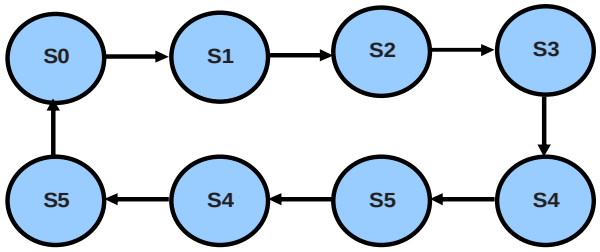
1.9.4 Számláló áramkörök

A sorrendi hálózatok egyik fontos elemét a számláló áramkörök alkotják. A számláló áramkör előre megadott sorrendben megy át a meghatározott állapotokon. A következő állapot logikai függvénye a számláló kimenetén lévő szekvenciát is meghatározza. Például ha szükségünk van egy három bites Gray kódú számlálóra, akkor a kimeneten a következő szekvenciának megfelelő kombinációs függvényt kell létrehoznunk: „000”, „001”, „011”, „010”, „110”, „111”, „101”, „100”. Az alábbiakban néhány példát mutatunk be.

Bináris számláló

A bináris számláló állapotai a bináris kódnak megfelelően ismétlődnek. A három bites bináris számláló például a 48 ábrán szemléltetett gráfon, nyolc állapoton megy át és a következő állapotkodólást alkalmaztuk:

S0 → „000”; S1 → „001”; S2 → „010”; S3 → „011”;
S4 → „100”; S5 → „101”; S6 → „110”; S7 → „111”



48. ábra 3 bites bináris számláló állapotgráfja

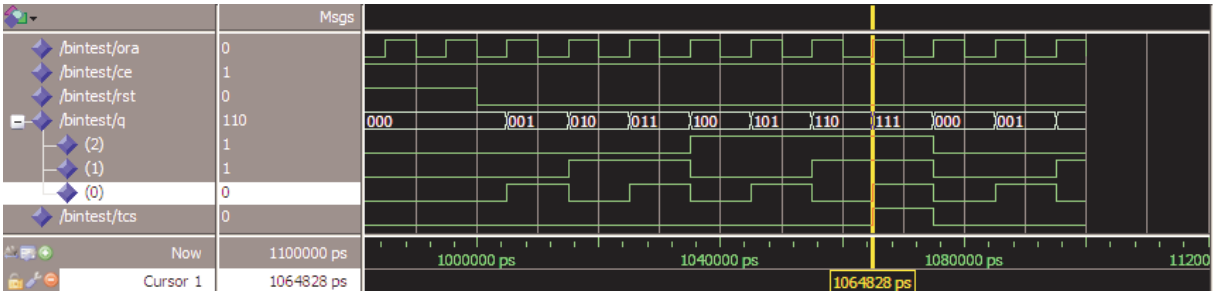
A bináris számláló megvalósítására két VHDL megvalósítási példát adunk. Az első példa a kimenetet előjel nélküli számként kezeli, amelyet minden órajel felfutó élének hatására növel eggyel. A második megoldás Köztes jeleket használ (reg_all és reg_kov_all) a jelenlegi és következő állapotok meghatározására és a kimenetet pedig a jelenlegi állapot regiszter (reg_all) alapján frissíti. Az első megvalósítás szimulációján (51. ábra) láthajuk, hogy a túlcsordulás mindig jelzi a számláló maximális értékének az elérését.

```
----------------------------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

entity binaris is
    Port ( ora : in  STD_LOGIC; -- orajel
                                ce  : in  STD_LOGIC; -- orejel engedelyezes ha = 1 akkor
szamol
           rst : in  STD_LOGIC; -- alpaallapot ha rst = 1 akkor q ="000" egyébként szamol
           q : inout  STD_LOGIC_VECTOR (2 downto 0); -- binaris szamlalo kimenet
           tcs : out  STD_LOGIC); -- tulcsordulas ha q="111" akkor tcs='1' egyébként '0'
end binaris;

architecture Behavioral of binaris is

begin

process (ora, rst)
 begin
   if rst='1' then
      q <= (others => '0');
   elsif ora='1' and ora'event then
      if ce='1' then
         q <= q + 1;
      end if;
   end if;
 end process;
 tcs <= '1' when q = "111" else
                              '0';
end Behavioral;
----------------------------------------------------------------------------------------------------------------------
```

49. ábra 3 bites bináris számláló első változat

```
----------------------------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

entity binaris is
    Port ( ora : in  STD_LOGIC; -- orajel
                                ce  : in  STD_LOGIC; -- orejel engedelyezes ha = 1 akkor
szamol
           rst : in  STD_LOGIC; -- alpaallapot ha rst = 1 akkor q ="000" egyébként szamol
           q : inout  STD_LOGIC_VECTOR (2 downto 0); -- binaris szamlalo kimenet
           tcs : out  STD_LOGIC); -- tulcsordulas ha q="111" akkor tcs='1' egyébként '0'
end binaris;

architecture Behavioral of binar2 is
           signal reg_all : STD_LOGIC_VECTOR (2 downto 0) ; -- jelenlegi allapot
regisztere
           signal reg_kov_all : STD_LOGIC_VECTOR (2 downto 0) ; -- kovetkezo allapot
regisztere
begin
           -- regiszter vezerlo
           process (ora, rst)
                      begin
                      if rst='1' then
                                 reg_all <= (others => '0');
                      elsif ora='1' and ora'event then
                                 reg_all <= reg_kov_all;
                      end if;
           end process;
           reg_kov_all <= reg_kov_all + 1;
           -- kimeneti függvény eloallitasa
           q <= std_logic_vector(reg_all);
--         tcs <= '1' when reg_all = "lll" else
           --                          '0'; -- tulcsordulas
end Behavioral;
----------------------------------------------------------------------------------------------------------------------
```

50. ábra 3 bites bináris számláló második változat



51. ábra 3 bites bináris számláló szimulációs eredmény

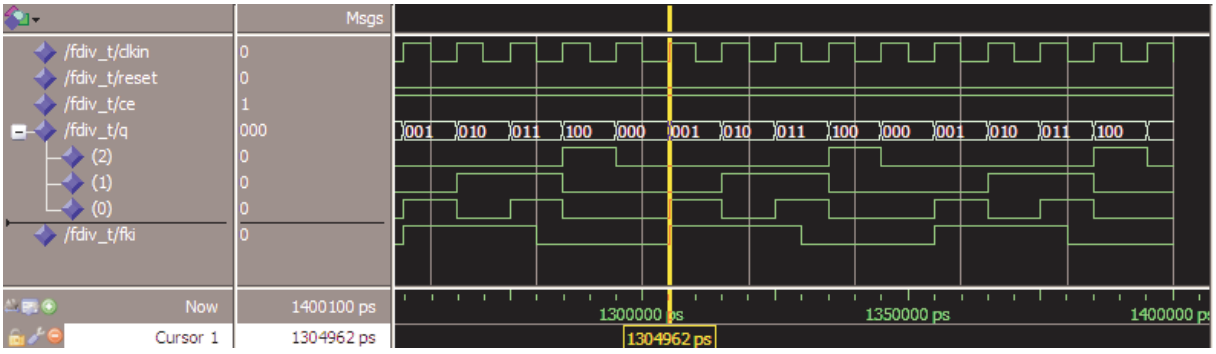
A fenti példákból megállapítható, hogy általában egy *n* bites számláló rendelkezik egy *n* bites regiszterrel és a kimenetet pedig előjel nélküli egészként kezeli a VHDL program. A szabadon futó bináris számláló minden órajel hatására növeli a számláló értékét egyel (0-tól égészen 2^n-1-ig).

Ötven százalékos kitöltési tényezőjű frekvenciaosztó

A bináris számlálók egyúttal frekvenciaosztást is végeznek. A kettő hatványai szerinti frekvenciaosztók kimeneti frekvenciája ötven százalékos kitöltési tényezőjű. A kettő hatványaitól különböző frekvenciaosztók esetében azonban a kimeneti frekvencia már nem ötven százalékos kitöltési tényezővel rendelkezik.
Példánkban egy olyan frekvenciaosztó áramkört mutatunk be amely átalakítható tetszőleges páratlan számú frekvencia osztás megvalósítására. A példában azonban az kimeneti frekvencia a bemeneti frekvencia ötöd része:

$$\boldsymbol{fki} = \frac{\boldsymbol{fbe}}{\mathbf{5}}; \qquad \textbf{(3.)}$$

A VHDL megvalósítás előtt vizsgáljuk meg a frekvencia osztó szimulációs eredményét (52 ábra). Megfigyelhető, hogy az fdiv_t/q(2) jele végzi az 1:5 – höz frekvencia osztást, de kitöltési tényezője húszszázalékos, míg az fdiv_t/fki jel kitöltési tényezője ötvenszázalékos.
Ezt úgy lehet elérni, hogy az idődiagram elemzése alapján megállapítjuk azt, hogy milyen feltételeknek kell teljesülni ahhoz, hogy a kívánt kimeneti jelalakot megvalósítsuk.



52. ábra 1:5 frekvenciaosztó szimulációs eredménye

Az idődiagram alapján megállapítható, hogy ahhoz, hogy fki jelátmenetei megvalósuljanak a következő feltételeknek kell teljesülniük:

1. fki felfutó él van ha:

$$(\boldsymbol{clkin}\uparrow) * (\boldsymbol{q}(\mathbf{2}) = '\mathbf{0}') * (\boldsymbol{q}(\mathbf{1}) = '\mathbf{0}') * (\boldsymbol{q}(\mathbf{0})\uparrow) \qquad \textbf{(4.)}$$

2. fki lefutó él van ha:

$$(\boldsymbol{clkin}\downarrow) * (\boldsymbol{q}(\mathbf{2}) = '\mathbf{0}') * (\boldsymbol{q}(\mathbf{1}) = '\mathbf{1}') * (\boldsymbol{q}(\mathbf{0}) = '\mathbf{1}') \textbf{(5.)}$$

ahol „↓” és „↑” jelek a felfutó illetve a lefutó éleket jelölik.

A grafikonon már nem látszik, de az (53. ábra), amely a feladat megvalósítását mutatja be megfigyelhető, hogy az fki jelet két belső jel (div1 és div2) antivalencia kapcsolatából nyerjük.

```
----------------------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;
entity fdiv_n is
	Generic (
		n: natural := 5 -- n+1 számláló beállítása
		);
	Port (
		clkin : in  STD_LOGIC;
		reset : in  STD_LOGIC;
		ce : in  STD_LOGIC;
		q : inout STD_LOGIC_VECTOR (2 downto 0);
		fki : out  STD_LOGIC);
end fdiv_n;
architecture Behavioral of fdiv_n is
	subtype divtype is natural range 0 to n-1;
	signal szamlalo: divtype;
	signal en_tff1: std_logic;
	signal en_tff2: std_logic;
	signal div1: std_logic;
	signal div2: std_logic;
begin
fosztas: process(clkin, reset, szamlalo)
begin
				if (reset = '1') then
					szamlalo <= 0;
					q <= (others => '0');
				elsif rising_edge(clkin) then
					if (szamlalo = (n-1)) then
						szamlalo <= 0;
						q <= (others => '0');
					else
						szamlalo <= szamlalo + 1;
						q <= q+1;
					end if;
				end if;
			end process;
en_tff1 <= '1' when szamlalo = 0 else '0';
en_tff2 <= '1' when szamlalo = (((n-1)/2)+1) else '0';
paratlan1: process(clkin, reset, en_tff1, div1)
begin
				if (reset = '1') then
					div1 <= '1';
				elsif rising_edge(clkin) then
					if (en_tff1 = '1') then
						div1 <= not(div1);
					end if;
				end if;
			end process;

paratlan2: process(clkin, reset, en_tff2, div2)
begin
				if (reset = '1') then
					div2 <= '1';
				elsif falling_edge(clkin) then
					if (en_tff2 = '1') then
						div2 <= not(div2);
					end if;
				end if;
			end process;
fki <= div1 xor div2;

end Behavioral;
----------------------------------------------------------------------------------------------------------------
```

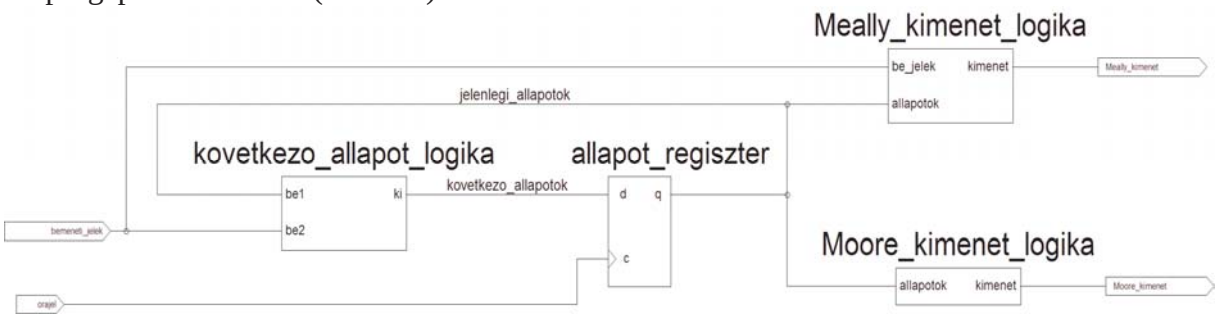
53. ábra 1:5 frekvenciaosztó ötven százalékos kitöltési tényezővel

$$fki = div1 \oplus div2; \qquad (3.)$$

Ily módon előállíthatunk bármilyen páratlan számú frekvenciaosztást, amelynek a kitöltési tényezője 50%. Természetesen a páros számú 50%-os kitöltési tényezőjű frekvenciaosztást hasonló módszerrel lehet megvalósítani.

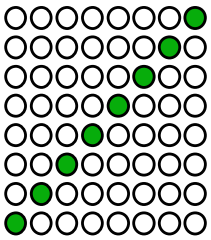
1.9.5 Véges állapotú állapotgépek

Ahogyan a cím is mutatja ebben a fejezetben tárgyalt sorrendi hálózatok rendelkeznek belső állapotokkal. Ellentétben a többi sorrendi hálózattal (flip-flop, regiszterek, számlálók), az állapotgépek állapotátmeneti feltétele nem egyszerűen csak az órajeltől függ, hanem a bementi jelek és a jelenlegi állapotok bonyolult függvényének eredménye. Formálisan egy állapotgépet öt paraméterrel jellemezhetünk: szimbolikus állapotok, bemeneti jelek, kimenetek, következő állapot függvénye, kimeneti függvény [2.]. A kimeneti függvények értéke meghatározza a kimeneti jelek állapotát. Amennyiben a kimeneti függvényt kizárólag a jelenlegi állapotok értékéből nyerjük, ebben az esetben *Moore* állapotgépről beszélünk. Ha a kimeneti függvényt a bemeneti jelek és a jelenlegi állapotokból nyerjük, akkor *Mealy* állapotgépről beszélünk (54. ábra).



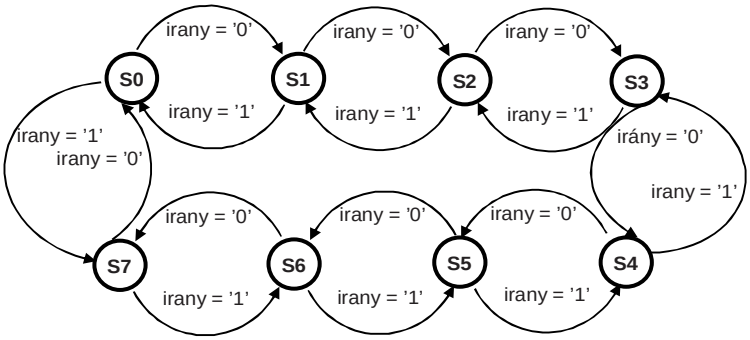
54. ábra Meally és Moore állapotgépek

Az 54 ábrán mindkét állapotgépet ábrázoltuk. Az alkalmazások többségében mindkét megvalósítás megtalálható ugyanazon állapotgépen, hiszen a kimenetek megvalósulhatnak az jelenlegi állapotban és az állapotátmenet ideje alatt is. Az állapot_regiszter, tároló elemként működik, amelyet a rendszer órajel szinkronizál. A „következő_állapot_logika” melynek a kimeneti függvényét (kovetkezo_allapotok) a bemeneti_jelek és a jelenlegi_allapotok határozzák meg. A kimeneti jeleket (Moore_kimenet, Meally_kimenet) a megfelelő kimeneti logika (Moore, Meally) képezik. Az állapotgépek alkalmazása ott lényeges, ahol logikailag egymás után következő műveleteket kell megvalósítani, amelyeknek ütemét a rendszer órajel határozza meg. Kétféle állapotgépet üzemmódban működik: vezérlő (vezérlő jeleket állít elő) és adatfeldolgozó (a rendszer adatait irányítja – koordinálja – és dolgozza fel).
Az alábbi példánk egy adatfeldolgozó állapotgépet mutat be (amennyiben egy futófény előállítása adatirányítási műveletet jelent). A futófény 8 LED diódájának egyenkénti vezérlését mutatja be az 55. ábra. Az ábra zölddel jelöli azokat a LEDeket, amelyek világítanak. A futófény kijelzési irányát egy külső kapcsoló (irany) a határozza meg.



55. ábra Futófény állapotainak ábrázolása

Ha a irany = ’0’ akkor a futófény jobbról-balra halad, ha irany = ’1’, akkor a futófény balról jobbra halad. A feladat állapotgráfjának ábrázolása az 56. ábrán látható, míg a feladat megvalósítása VHDL programmal és ennek szimulációja az 57. ábrán illetve az 58. ábrán látható.

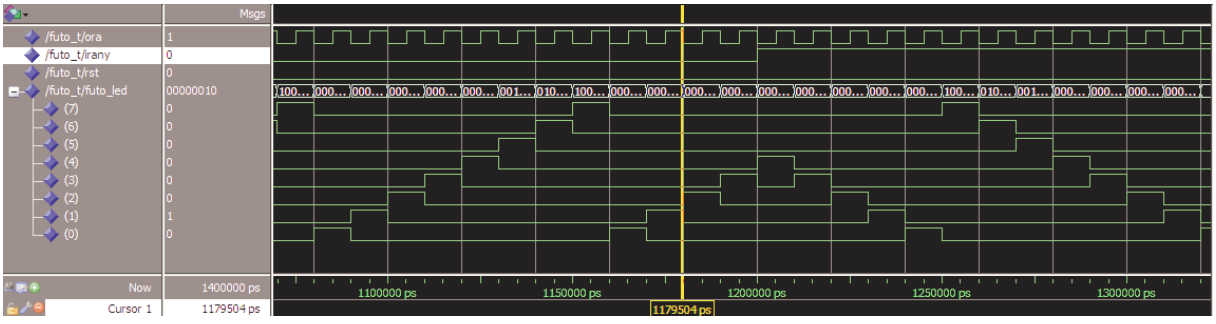


56. ábra Futófény állapotainak ábrázolása

A VHDL programban az egyes állapotokat (s0 .. s7) *case* struktúrával valósítottuk meg. A folyamat az órajel (ora) és a reset jel (rst) változására élesedik. Az állapotgép futásának irányát a feltétel kiértékelés fogja megadni, azaz a következő állapot meghatározása (jell_allapot) határozza meg a kimeneti jelkombinációt.

```
----------------------------------------------------------------------------------------------------------------
end Behavioral; library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_logic_unsigned.all;

entity fsm_futo_feny is
    Port ( ora : in  STD_LOGIC;
                              irany: in STD_LOGIC;
           rst : in  STD_LOGIC;
           futo_led : out  STD_LOGIC_VECTOR (7 downto 0));
end fsm_futo_feny;

architecture Behavioral of fsm_futo_feny is
signal jell_allapot: STD_LOGIC_VECTOR (2 downto 0);
begin

            process (ora,rst)
            begin
            if rst = '1' then
                        jell_allapot <= (others => '0');
                        futo_led <= "00000000";
            elsif rising_edge(ora) then
                        if irany ='0' then
                                    jell_allapot <= jell_allapot + 1;
                        else
                                    jell_allapot <= jell_allapot - 1;
                        end if;
            end if;
            case (jell_allapot) is
                  when "000" =>
                   futo_led <= "00000001";
                 when "001" =>
                   futo_led <= "00000010";
                 when "010" =>
                   futo_led <= "00000100";
                 when "011" =>
                   futo_led <= "00001000";
                 when "100" =>
                   futo_led <= "00010000";
                when "101" =>
                   futo_led <= "00100000";
                 when "110" =>
                   futo_led <= "01000000";
                 when "111" =>
                   futo_led <= "10000000";
            when others =>
                   futo_led <= "00000000";
            end case;
            end process;

end Behavioral;

----------------------------------------------------------------------------------------------------------------
```

57. ábra Futófényt megvalósító állapotgép

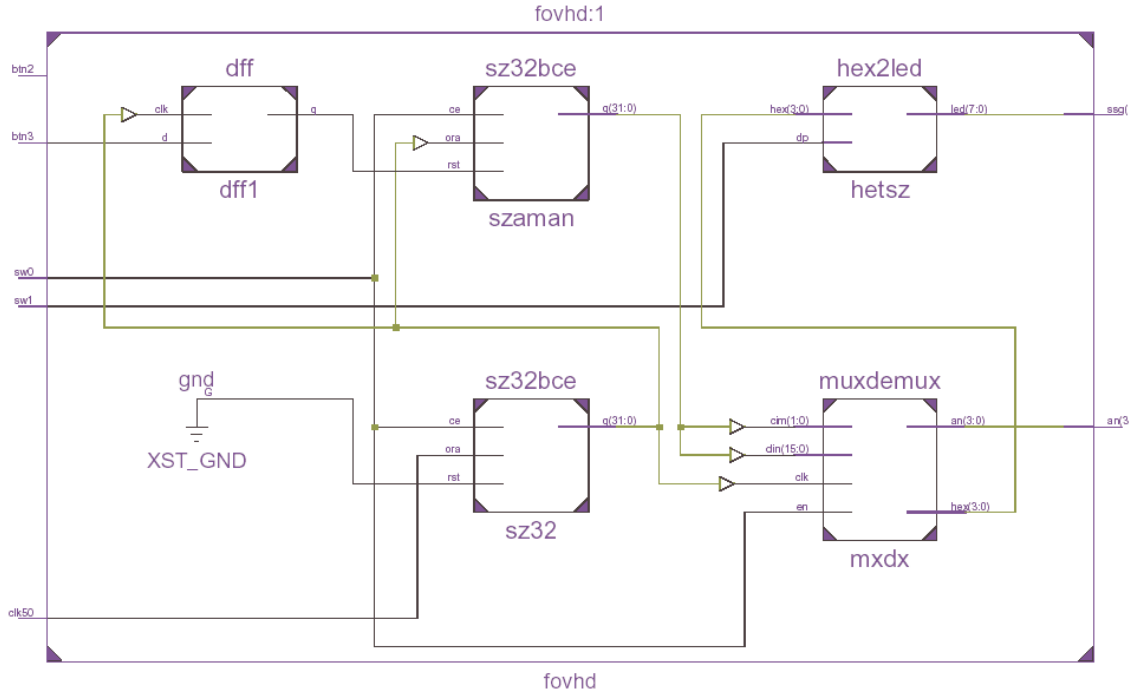


58. ábra Futófény állapotgép szimulációs eredménye

Az áramköri szimuláció eredményén látszik a futófény irányának változása az irany jel változtatásánál.

1.10 Komplex feladat

Feladat: Tervezze és valósítsa meg VHDL nyelven egy 16 bites számláló kimeneteinek ábrázolását a négydigites hétszegmenses kiselzőn (Digilent kártya), a rendszer órajel 50MHz. A feladat megvalósításához használjuk fel a VHDL fejezet ismereteit. Segítségül az 59. ábrán ábrázoltuk a feladat egy megvalósításának kapcsolási rajzát. A feladat egyes elemei: sz32 – frekvencia osztó sz32bce – 32 bites számláló; dff – reset áramkör; hex2led – hétszegmenses kijelző vezérlő, muxdemux- adatelosztó és adat szinkronizálás (digit – adat megfeleltetés lásd 7. ábra). A 60. ábrán a feladat legfelső hierarchiájának (fovhd) VHDL programját adtuk meg.
Megjegyzés: az egyes alkatrészek összekötésénél azon port jeleket amelyeket nem kötünk össze más jellel (port map) *open* atributummal jelezzük, hogy a port jele nem csatlakozik sehova.



59. ábra Komplex feladat kapcsolási rajza.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity fovhd is
    Port ( clk50 : in  STD_LOGIC; -- 50MHz orajel
                          sw0 : in STD_LOGIC; -- kapcsolo
                          sw1 : in STD_LOGIC; -- kapcsolo1
                          btn2 : in STD_LOGIC; -- nyomogomb
                          btn3 : in STD_LOGIC; -- nyomogomb
                          an : out  STD_LOGIC_VECTOR (3 downto 0); -- anodok
                          ssg: out STD_LOGIC_VECTOR (7 downto 0); -- hetszegmens
           led : out  STD_LOGIC_VECTOR (7 downto 0)); -- ledek
end fovhd;

architecture Behavioral of fovhd is

signal orajel: std_logic_vector (31 downto 0); -- az orajel leosztasahoz
```

```
signal adc: std_logic_vector (3 downto 0); -- adat vezeték
signal reset: std_logic; -- belső reset
signal reset1: std_logic; -- másik reset
signal pereg: std_logic; -- pregésmentesítéshez az orajel
signal orajel1: std_logic; --dff2 órajel
signal asin: std_logic_vector (1 downto 0); -- A belso anód sín
signal data16: std_logic_vector (15 downto 0); -- héstszegmesre bemenő adatok
signal hex : std_logic_vector (3 downto 0); -- a hetszegmenses vezerlo bemenete

-- 32 bites szamlalo
component sz32bce -- 32bit szamlalo 50Mhz osztashoz
port ( ora : in  STD_LOGIC;
       ce  : in  STD_LOGIC;
       rst : in  STD_LOGIC;
       q : inout  STD_LOGIC_VECTOR (31 downto 0));
end component sz32bce;

-- negybites szamlalo
component sz4bce
Port (  ce : in  STD_LOGIC;
        clk : in  STD_LOGIC;
        rst : in  STD_LOGIC;
        q : inout  STD_LOGIC_VECTOR (3 downto 0));
end component sz4bce;

-- dff
component dff
   Port ( clk : in  STD_LOGIC;
          d : in  STD_LOGIC;
          q : out  STD_LOGIC);
end component dff;

-- 7 szegmens
component hex2led
         Port ( hex : in  STD_LOGIC_VECTOR (3 downto 0);
                        dp  : in STD_logic;
        led : out  STD_LOGIC_VECTOR (7 downto 0));
end component hex2led;

-- mux 16rol 4re
--component mux16_4
--    Port ( adat : in  STD_LOGIC_VECTOR (15 downto 0);
--                        asin: in STD_LOGIC_VECTOR (1 downto 0);
--          hex : out  STD_LOGIC_VECTOR (3 downto 0));
--end component mux16_4;

component muxdemux
         Port ( clk : in  STD_LOGIC;
         en : in  STD_LOGIC;
                        cim : in STD_LOGIC_VECTOR (1 downto 0);
```

```
        din : in  STD_LOGIC_VECTOR (15 downto 0);
        an : out  STD_LOGIC_VECTOR (3 downto 0);
        hex : out  STD_LOGIC_VECTOR (3 downto 0));
end component muxdemux;

begin
reset1 <= reset or adc(3);
pereg  <= orajel (10);

sz32: sz32bce -- 32 bites számláló
port map (
                        ora => clk50,
                        ce  => sw0,
                        rst => '0',
                        q => orajel
);
sz4: sz4bce -- 4 bites számláló
port map ( ce =>  sw0,
                        clk => orajel(21),
                        rst => reset1,
                        q   => adc
                                );
dff1: dff -- pergésmentesítés
port map (clk => orajel(10),--pereg,
        d => btn3,
                        q => reset
);
dff2: dff -- pergésmentesítés
port map (clk => pereg,
        d =>  btn2,
                        q => orajel1
);
--cb2: cb2ce
--port map ( q => asin,
--          ce => SW0,
--          clk => orajel1,
--          rst => reset
--);
szaman: sz32bce -- 32bit szamlalo hetszegmenses anod és 16 bites számláló
port map ( ora => orajel (17),
                        ce  => sw0,
                        rst => reset,
                        q (1 downto 0) => asin,
                        q (3 downto 2) => open,
                        q (19 downto 4)       => data16,
                        q (31 downto 20) => open
                        );
hetsz: hex2led -- het szegmenses vezerlo
port map ( hex => hex,
                        dp      => sw1,
```

```
        led => ssg

                        );
--mux: mux16_4
--port map ( adat => data16,
--                      asin => asin,
--          hex => hex
--                      );
mxdx: muxdemux
        port map( clk => orajel (17),
        en => sw0,
                        cim => asin,
        din => data16,
        an  => an,
        hex  => hex
                        );

end Behavioral;
```

---

60. ábra Komplex feladat megvalósításának fő programja

1.11 Összefoglalás

Jelen fejezetben ismertettük a VHDL nyelv alapjait a VHDL szintaktikától és elemeitől kezdve egészen a folyamatokig. Az egyes elemek ismertetését példákon keresztül szemléltettük, amelyeknek működőképességét szimulációval ellenőriztük. Bemutattuk a kombinációs és sorrendi hálózatok megvalósítását, különböző példákon keresztül, majd végül a véges állapotú állapotgépek megvalósítását mutattuk be.
A fejezet alapismereteket szolgáltat a VHDL programozás alapjainak bemutatásával. VHDL ismeretek elmélyítéséhez az irodalomban megadott VHDL témakörben írt könyveket ajánljuk.

Irodalom

[1.] Dr. Gál Tibor: *Programozható logikák*, jegyzet, Műegyetemi Kiadó, Budapest,

[2.] Pong. P. Chu: *RTL Hardware Desing Using VHDL – Coding for Efficiency, Portability and Scalability*, John Wiley and Sons, ISBN-13: 978-0-471-72092-8, ISBN-10: 0-471-72092-5, 2006, pp. 694.

[3.] Yalamanchili S.: *Introductory VHDL From Simulation to Synthesis*, Prentice Hall, ISN 0-13-080982-9, 2001, pp. 401.

[4.] Digilent Inc: *Nexsys2 Board Reference Manual*, www.digilentinc.com, 2008, pp. 17.

[5.]

Tartalom

1 Hardver leíró nyelvek – VHDL 1
1.1 Digitális rendszerek tervezése 1
1.2 Hardver absztrakciós szintek 2
1.3 Digitális áramkörök szöveges leírása 2
1.3.1 VHDL Hardverleíró nyelv 3
1.3.2 Verilog hardverleíró nyelv 3
1.3.3 System C hadverleíró nyelv 3
1.4 VHDL alapok 4
1.4.1 Az tervezési egység/egyed (Entity) mint modell interfész 5
1.4.2 Megvalósítási egység (építmény - architecture) 6
1.4.3 Tervezési egység, könyvtár 7
1.4.4 A VHDL program feldolgozása 8
1.5 A VHDL szintaktika és elemek 8
1.5.1 Lexikális elemek 8
1.5.2 Objektumok 10
1.5.3 Adat típusok, műveletek 11
1.5.4 Szabványos VHDL adattípusok 11
1.5.5 Általános VHDL kódolási ajánlások 14
1.6 Egyidejű értékadási azonosságok a VHDL nyelvben 14
1.6.1 Kombinációs/sorrendi hálózatok 15
1.6.2 Egyszerű értékadás 15
1.6.3 Feltételes értékadás 16
1.6.4 Multiplexer megvalósítása 16
1.6.5 Négydigites hétszegmenses kijelző anódjainak vezérlése 17
1.6.6 Jelkombinációk szerinti értékadás 19
1.6.7 Hétszegmenses kijelző vezérlése 20
1.7 Szekvenciális értékadás 23
1.7.1 Folyamatok 24
1.7.2 Folyamatok felfüggesztése „wait” állapotokkal 25
1.7.3 Változók 26
1.7.4 IF Feltételes elágazás 27
1.7.5 CASE feltételes elágazás 27
1.8 Kombinációs hálózatok leírása 28
1.8.1 Összeadó- kivonó áramkör 28
1.8.2 Komparátor áramkör 30
1.8.3 Barrel shifter (eltoló) áramkör 32

1.8.4 Prioritás kódoló áramkör 4to2 33
1.8.5 Bináris-Gray kódátalakító áramkör 33
1.9 Sorrendi hálózatok leírása VHDL 35
1.9.1 D tároló 35
1.9.2 D flip-flop 37
1.9.3 Shift regiszter 38
1.9.4 Számláló áramkörök 42
1.9.5 Véges állapotú állapotgépek 46
1.10 Komplex feladat 49
1.11 Összefoglalás 52
Irodalom 52