6. Digitális rendszertervezés I. - RTL modellezés és szimuláció

Szerző: Horváth Péter

Az *RTL modellezés és szimuláció* c. laboratóriumi gyakorlat során egyszerű logikai és szinkron szekvenciális hálózatok RTL modelljeit készítjük el VHDL nyelven, majd elvégezzük azok funkcionális verifikációját HDL-alapú lineáris testbench segítségével. E rövid elméleti összefoglalónak nem célja a VHDL nyelv részletes ismertetése, ezért csak a legalapvetőbb nyelvi szerkezetek bemutatására kerül sor, amelyek megfelelő kombinálásával azonban már egészen összetett funkciót megvalósító áramkörök is modellezhetők[4]. A legfontosabb VHDL nyelvi konstrukciókon túlmenően az elméleti összefoglaló néhány egyszerű példa segítségével betekintést nyújt a HDL-alapú lineáris tesztkörnyezetek készítésébe is.

A digitális integrált áramkörök tervezése ma már - a nagyon egyszerű részáramköröktől eltekintve - szinte mindig valamilyen hardverleíró nyelvre (*Hardware Description Language*, HDL) épülő módszerrel történik. A HDL-ek (VHDL, Verilog, SystemC, SystemVerilog stb.) többféle **elvonatkoztatás** (absztrakció) modellezésére képesek; segítségükkel készíthetők időzítési információktól mentes **viselkedési** leírások, automatizált szintézisre optimalizált **regiszter-transzfer szintű** (*Register-Transfer Level*, RTL) és akár **kapuszintű** modellek is egyazon nyelvi eszköztár felhasználásával.

A modern VLSI áramkörök fejlesztése során a kézi HDL modellezésnek az RTL elvonatkoztatási szinten van a legnagyobb jelentősége. Az RTL tervezőnek a specifikáció általában valamilyen magas szintű programozási nyelven megfogalmazott algoritmus formájában (futtatható specifikáció, *executable specification*) áll rendelkezésére. Az ő feladata az algoritmus egyes fázisainak végrehajtásához szükséges **műveletvégző erőforrások** (tároló elemek, pl. regiszterek, regisztertömbök, valamint aritmetikai/logikai elemek), valamint azok összeköttetéseinek meghatározása, figyelembe véve az áramkörrel szemben támasztott, **erőforrásigényre**, **számítási teljesítményre** és **fogyasztásra** vonatkozó követelményeket. Gondoskodnia kell továbbá az összekapcsolt erőforrások megfelelően időzített vezérlő jeleinek előállításáról. Ezt a feladatot általában **állapotgép**-alapú vezérlő logikák végzik.

Az RTL tervezés fontos aspektusa az egyes erőforrások működésének szinkronizációjához elengedhetetlen **fázisjelséma**, amely legegyszerűbb esetben egyetlen órajel[5]. Általános

---

[4] A VHDL nyelv teljeskörű bemutatását a [6] forrás tartalmazza.

[5] Egyes források úgy hivatkoznak az RTL elvonatkoztatású tervezésre, mint az a tervezési lépés, amelynek feladata annak meghatározása, hogy a rendszerben mi történjen az órajel két felfutó éle között. Mivel a fázisjelséma egy funkcionális egységen (makrocellán) belül is több, eltérő fázisú és frekvenciájú órajelet tartalmazhat, a pontosabb megfogalmazás inkább úgy hangzana, hogy "*...mi történjen a fázisjelséma két aktív éle között.*"

esetben a VLSI áramkörök egyes részáramkörei (makrocellái) más-más frekvenciájú órajeleket igényelnek, amelyek fázisviszonya sem feltétlenül kötött. Az ilyen, egyedi fázisjelsémát alkalmazó makrocellák ún. aszinkron **órajeltartományokat** *(clock domain)* alkotnak, amelyek között a megbízható adatcserét megvalósító áramköri részletek megtervezése (*Clock Domain Crossing*, CDC) ugyancsak az RTL tervező feladata.

6.1. RTL modellezés VHDL nyelven

E laboratórium témája az RTL modellezés és RTL szimuláció. Az alkalmazott HDL esetünkben a VHDL *(Very High Speed Integrated Circuit Hardware Description Language)* lesz, amelynek RTL modellezésben használatos eszközeit ezen elméleti összefoglaló lépésenként, az egyszerű logikai kapukat leíró nyelvi szerkezetektől kezdve mutatja be.

- Az első példacsoportban bemutatjuk az egyszerű **logikai kapuk** VHDL modellezését. A példák szemléltetik a VHDL tervezési egységek szerkezetét (egyed deklaráció és architektúra), a standard logikai vektor típus, valamint a konkurens értékadás használatát.
- A második példacsoport **összeadó áramkörök** modellezésének lehetőségeit mutatja be, felhasználva az első példacsoport logikai kapuit. Az itt tárgyalt VHDL nyelvi elemek a komponensbültetés (példányosítás), értékhordozó-példányosítás (jelek), az aritmetikai könyvtárak, a többszörös implementáció és a típuskonverziók.
- A harmadik példacsoport témája a **szinkron szekvenciális hálózatok** modellezése. Az alkalmazott VHDL nyelvi eszközök a folyamat *(process)*, az érzékenységi lista, a feltételes utasítás, a generikus interfész és a statikus kiértékelésű ciklus. Itt vezetjük be a delta-delay fogalmát is.
- A negyedik példacsoport egy ***Shift&Add*** elvű **szorzó** egység RTL modelljének egy lehetséges VHDL megvalósítását mutatja be. E példaáramkör célja az állapotgéppel megvalósított adatfeldolgozó rendszerek bevezetése.

Az RTL elvonatkoztatás fent bemutatott tulajdonságai alapján az első két példacsoport modelljei természetesen nem tekinthető RTL szintűnek, a szükséges nyelvi szerkezeteket azonban jól szemléltetik.

6.1.1. Első példacsoport - Logikai kapuk

A VHDL tervezési egységek két fő alegységből, az **egyed-deklarációból** *(entity)* és a **funkcionális leírásból**, más néven **architektúrából** *(architecture)* állnak. Az egyed-deklaráció tartalmazza a tervezési egység interfészét, amely azt az információt foglalja magában, amelyre a modul felhasználójának szüksége van a modul példányosításakor:

- ***portlista***: az egyed ki- és bemeneti kapcsai, azok típusa és mérete
- ***generikus paraméterek listája***: az egyed beültetésekor (példányosításakor) megadható paraméterek, amelyek az egyed alapvető funkciójára nincsenek hatással, de

a kód újrafelhasználhatóságát jelentősen javítják (pl. egy regiszter bitjeinek száma, egy logikai kapu késleltetése stb.).

Az egyed funkcionális leírása (implementációja) a generikus paramétereket és a portlista elemeit felhasználva leírja az egyed működését. A 6-1. ábra egy AND logikai kapu példáján mutatja be a VHDL tervezési egységek szerkezetét.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity and_gate is
5      port ( a:  in  std_logic;
6             b:  in  std_logic;
7             y:  out std_logic);
8  end entity and_gate;
9
10 architecture a1 of and_gate is
11 begin
12     y <= a and b;
13 end architecture a1;
```

*A felhasznált **könyvtárak** és **csomagok** deklarációja.*

***Egyed-deklaráció**: a tervezési egység interfészét (portjainak nevét, irányát és típusát), valamint a generikus paramétereit (lásd 3. példacsoport) tartalmazza.*

*A tervezési egység **funkciójának leírása**. Egy egyed-deklarációhoz (interfészhez) több architektúra (implementáció) is készíthető (lásd 2. példacsoport).*

6-1. ábra A VHDL tervezési egység szerkezeti elemei

Az AND kapu két bemeneti porttal rendelkezik *(a és b)*, amelyek típusa *std_logic*. A kapu egyetlen kimenete *(y)* ugyancsak *std_logic* típusú. Az *std_logic* típus szabványos ugyan, de nem a VHDL beépített típusa. Lehetséges értékei[6], valamint az ilyen típusú értékhordozókon (jeleken, változókon) végzett műveletek (pl. logikai és aritmetikai műveletek) az *ieee* könyvtár csomagjaiban vannak definiálva. Ha csak logikai műveletekre és egyenlőségvizsgálatra van szükségünk, akkor elegendő az *std_logic_1164* csomagot használni, de aritmetikai műveletek esetén már egyéb csomagokra is szükség van (lásd harmadik példacsoport).

A 6-2. ábrán az OR, az XOR és a NAND kapuk VHDL modellje látható. Az *std_logic_1164* csomag természetesen több logikai műveletet is definiál (*not*, *xnor* stb.). Az ábrán látható három alapkapu-típusra a következő példákban szükségünk lesz; segítségükkel összetettebb funkciót megvalósító áramköröket fogunk modellezni.

---

[6] A szokásos logikai alacsony és magas szinten kívül az *std_logic* további 7 értéket képes ábrázolni, amelyek a különböző meghajtóképességű áramkör-kimeneteket hivatottak modellezni. Pl. egy arány típusú logikai kapu (pl. egy TTL kapu) kimenete ún. erős alacsony *(strong low)* és gyenge magas *(weak high)* logikai értékeket tud előállítani. A kifejezések arra utalnak, hogy egy ilyen áramkör kimenetét terhelve a logikai alacsony szint a terhelés értékétől független, a magas szint viszont nagy terhelőáram esetén “leromlik”, lecsökken.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
------------------------------
entity or_gate is
    port (a:  in  std_logic;
          b:  in  std_logic;
          y:  out std_logic);
end entity or_gate;
------------------------------
architecture o1 of or_gate is
begin
    y <= a or b;
end architecture o1;
```

```vhdl
library ieee;
use ieee.std_logic_1164.all;
------------------------------
entity xor_gate is
    port (a:  in  std_logic;
          b:  in  std_logic;
          y:  out std_logic);
end entity xor_gate;
------------------------------
architecture x1 of xor_gate is
begin
    y <= a xor b after 2 ns;
end architecture x1;
```

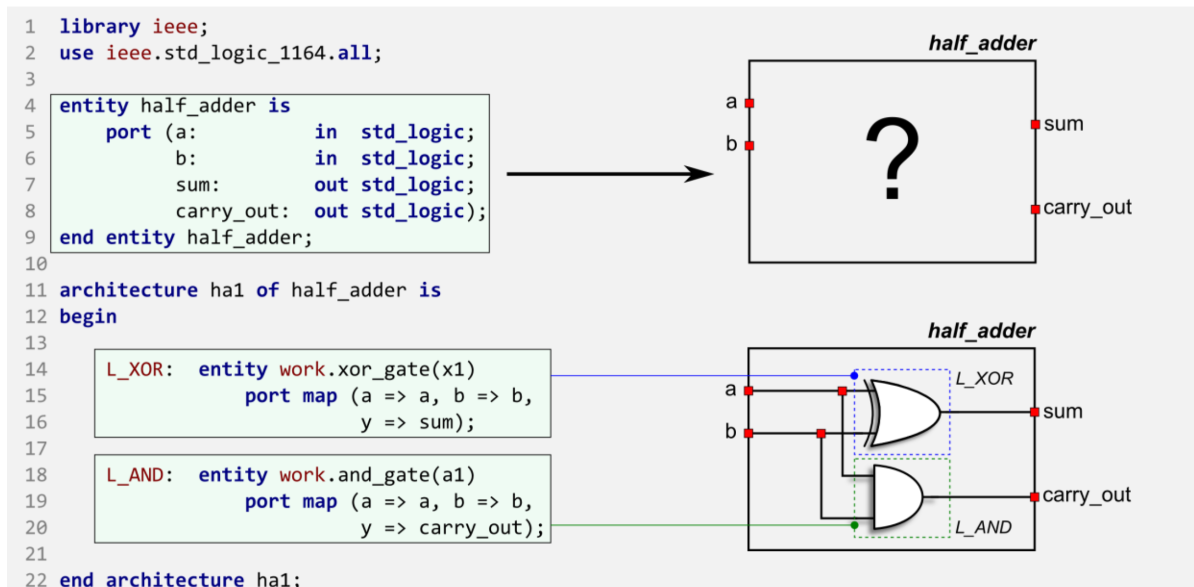
```vhdl
library ieee;
use ieee.std_logic_1164.all;
------------------------------
entity nand_gate is
    port (a:  in  std_logic;
          b:  in  std_logic;
          y:  out std_logic);
end entity nand_gate;
------------------------------
architecture n1 of nand_gate is
begin
    y <= a nand b;
end architecture n1;
```

6-2. ábra OR, XOR és NAND logikai kapuk VHDL modellje

Az XOR kapu modelljében látható, miként írható le egy logikai kapu késleltetése. Fontos megjegyezni, hogy a késleltetés effajta leírása az ún. *nem szintetizálható* nyelvi elemek közé tartozik; az automatizált RTL szintézis eszköz az ilyen és ehhez hasonló, explicit időzítési információt teljesen figyelmen kívül hagyja. Az áramkör késleltetése a szintézis során a megvalósítás alapjául szolgáló technológiától függően alakul ki. Az explicit késleltetés leírásának lehetőségét az RTL modellezés során gyakorlatilag nem használjuk ki, ugyanakkor a tervezés alatt álló áramkör modelljének funkcionális verifikációja során jelentős szerepük van az ilyen jellegű utasításoknak (lásd 6.2 pont).

6.1.2. Második példacsoport - Összeadó áramkörök

Az első példacsoport logikai kapuit felhasználva összetettebb áramkörök állíthatók össze, kihasználva a VHDL tervezési egységek moduláris felépítését és a komponensbeültetés (példányosítás) lehetőségét. A 6-3. ábra egy 1-bites fél-összeadó kapcsolási sémáját és VHDL modelljét mutatja be.



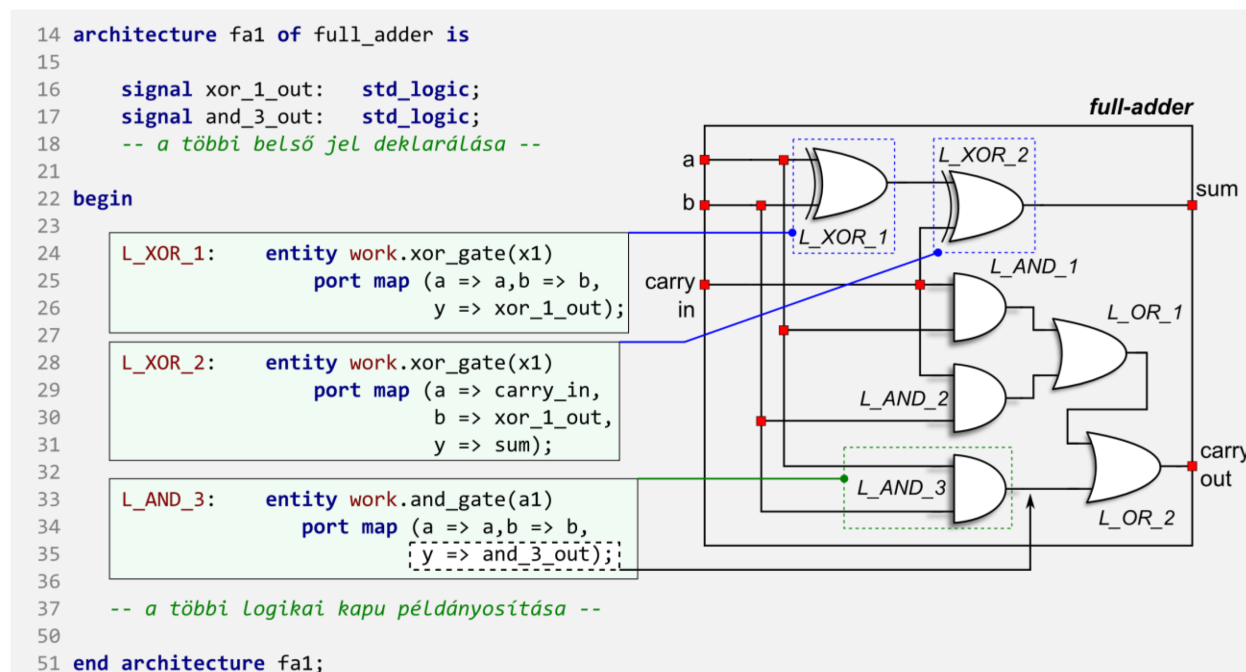
6-3. ábra Fél-összeadó kapuszintű VHDL modellje és kapcsolási sémája

A VHDL többféle módon képes leírni a hierarchikus felépítésű terveket. A 6-3. ábrán látható megoldás az ún. *közvetlen beültetés*, amely során a beépítendő komponensek nevét *(xor_gate, and_gate)*, implementációjuk nevét *(x1, a1)*, az aktuális tervezési egységen belüli példánynevet *(L_XOR, L_AND)*, valamint azt a könyvtárnevet kell megadni, amelyben a beültetendő komponens megtalálható. Esetünkben ez az alapértelmezett *work* könyvtár. Minden általunk készített VHDL komponens automatikusan ennek a könyvtárnak a része lesz.

A 6-3. ábra egy fontos tanulsága a következő: **a VHDL modell *architecture* részében az egyes utasítások sorrendje nem kötött**. A két logikai kapu sorrendjének felcserélésével ugyanazt a modellt kapjuk. Ez az állítás nem csak akkor igaz, ha az utasítások komponensek beültetését írják le. A VHDL modell ***architecture* mezője** egy ún. **konkurens utasításblokkot** képez, ami azt jelenti, hogy az itt felsorolt **utasítások párhuzamosan működő áramköri részleteket írnak le**. Az utasítások sorrendje nem befolyásolja a funkcionalitást.

FONTOS: a konkurens utasításblokkon kívül a VHDL ún. sorrendi (szekvenciális) utasításblokkokat is definiál (ilyen például a harmadik példacsoportban bemutatott *process*), amelyeken belül az utasítások sorrendjének fontos szerepe van.

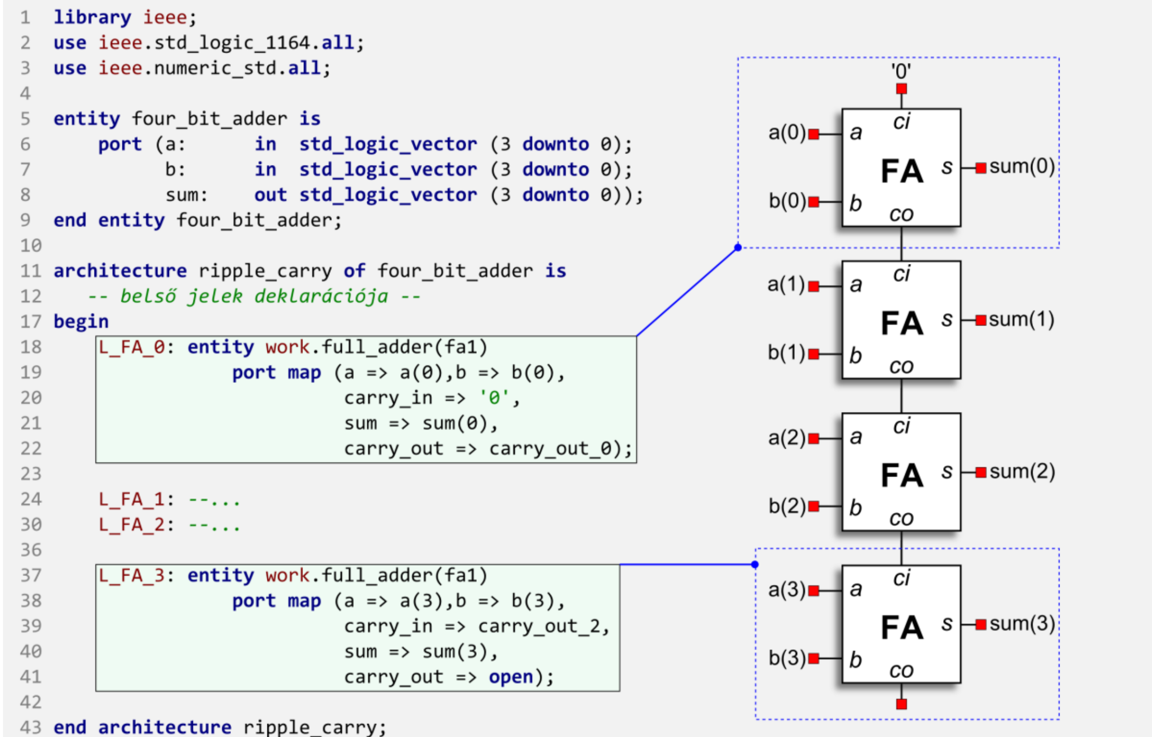
Hasonló módon készíthető el a teljes összeadó VHDL modellje is. A terjedelmi korlátokra való tekintettel ennek az áramkörnek már csak a kapcsolási sémáját és VHDL modelljének egyes részleteit mutatjuk be[7] (6-4. ábra).



6-4. ábra Teljes összeadó kapuszintű VHDL modelljének részlete és kapcsolási sémája

A következő példa a tervezési hierarchia egy további szintjét mutatja be. A 6-5. ábrán egy 4-bites, ripple-carry típusú összeadó áramkör VHDL modellje látható, amelyben felhasználtuk az előző példa teljes összeadóját.

[7] A teljes kód megtalálható a kiadott forrásfájlokban.



6-5. ábra 4-bites ripple-carry összeadó VHDL modelljének részlete és kapcsolási sémája (FA: Full-Adder)

A fenti példákból látható, hogy a kapuszintű modellezéssel már egy viszonylag egyszerű funkció leírásához is rengeteg kódsorra van szükség, ráadásul a modellben nagyon sok a hibalehetőség, a hibák felderítése pedig nehézkes. A fejlesztés gyorsításához magasabb elvonatkoztatású modellekre van szükségünk. A HDL-eket ezért általában nem kapuszintű, hanem RTL szintű modellezésre használják. Egy RTL modellben az összeadás funkció egy előre elkészített és általában szabványosított könyvtár valamely csomagjában megtalálható, a kívánt típusra túlterhelt operátorként jelenik meg. Ezt szemlélteti a 6-6. ábrán látható példa, amely a 4-bites összeadót az *std_logic* típusra definiálja. Egy egyed-deklarációhoz (interfészhez) több implementáció *(architecture)* is tartozhat. A példában az látható, hogy a szabványos *ieee* könyvtár *numeric_std* csomagjának felhasználásával az összeadó áramkör modellezéséhez csupán egyetlen utasításra van szükség, amelyben felhasználjuk az *std_logic_vector* (az *std_logic* típus több-bites változata) típusra túlterhelt + operátort.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity four_bit_adder is
6      port (a:   in  std_logic_vector (3 downto 0);
7            b:   in  std_logic_vector (3 downto 0);
8            sum: out std_logic_vector (3 downto 0));
9  end entity four_bit_adder;
10
11 architecture rtl of four_bit_adder is
12 begin
13     sum <= std_logic_vector(unsigned(a) + unsigned(b));
14 end architecture rtl;
```

Logikai vektorok közötti aritmetikai műveleteket is használunk, ezért szükség van egy ilyen operátorokat tartalmazó csomagra is.

Az áramkör interfésze teljesen azonos az előző példában bemutatottéval, ezért e két implementáció egyazon egyed-deklarációhoz is tartozhat.

Az összeadás funkciót egyetlen operátorhívással valósítjuk meg.

6-6. ábra 4-bites összeadó RTL modellje

A 6-6. ábrán látható példában tetten érhető a VHDL nyelv erősen típusos jellege. Az aritmetikai műveletek a *numeric_std* csomagban csak bizonyos, jól meghatározott típusú operandusokra vannak értelmezve. Ennek következtében az *std_logic_vector* típusú operandusokat először az ugyancsak a *numeric_std* csomagban definiált *unsigned* típusra kell konvertálni, majd az *unsigned* típusú eredményt vissza kell alakítani *std_logic_vector* típusra ahhoz, hogy az érték hozzárendelhető legyen az *std_logic_vector* típusú kimeneti porthoz. Az aritmetikai csomagok, mint amilyen a *numeric_std*, funkcionálisan teljesek, tehát tartalmazzák mindazokat a konverziós függvényeket, amelyek lehetővé teszik a csomag önálló használatát.

6.1.3. Harmadik példacsoport - Szinkron számlálók

A harmadik példacsoport bemutatja a szinkron hálózatok VHDL modellezésének módját két egyszerű számláló áramkör példáján keresztül. A **szinkron hálózatok** modellezésének elsődleges eszköze a VHDL nyelvben az ún. **folyamat** *(process)*, amely egy érzékenységi listából[8], egy deklarációs részből és egy sorrendi jellegű utasítástörzsből áll:

- **érzékenységi lista**: Jelek *(signal)* listája. A listán szereplő jelek bármelyikének megváltozásakor a folyamat törzsének utasításai sorban lefutnak.
- **deklarációs rész**: A folyamat törzsén belül, és csak ott használt erőforrások, pl. változók, függvények, eljárások stb. deklarációit tartalmazza. Példáinkban nem lesz szükségünk olyan erőforrásokra, amelyek csak a folyamaton belülről hozzáférhetők.
- **utasítástörzs**: A folyamat - mivel a VHDL architektúrának, tehát egy konkurens utasításblokknak a részét képezi - egy önálló áramköri részletként fogható fel, amelynek

[8] Az érzékenységi lista nélküli folyamatot a 6.2 pontban mutatjuk be.

funkcióját az utasítástörzs írja le. Az utasítástörzs sorrendi blokkot képez, a benne foglalt utasítások egymás után hajtódnak végre.

Bár a folyamat törzsének utasításai **sorban, de még egyazon szimulációs cikluson belül** hajtódnak végre. Ennek az a következménye, hogy az *n.* utasítás eredményét az *n+1.* utasítás "még nem látja"[9]. Tegyük fel, hogy két, egymást követő utasítás mindegyike egy egyszerű értékadás. **Ha az *n+1.* értékadás az *n.* értékadás eredményét olvassa, akkor mindig az előző szimulációs ciklusban - tulajdonképpen a folyamat előző lefutásakor - érvényes értéket fogja látni.** Ennek az a magyarázata, hogy a szimulációs cikluson belül az értékadások végrehajtása két lépésben, egy jobbérték-kiértékelési és egy balérték-frissítési lépésben hajtódik végre. **A balértékek frissítésére csak akkor kerül sor, ha már minden jobbérték kiértékelődött.** Egy jobbérték kiértékelése és a hozzá tartozó balérték frissítése között tehát több esemény is lezajlik. Ezt a jelenséget **delta-késleltetésnek** (delta-delay) nevezzük. **FONTOS**: Az elnevezés félrevezető lehet: a delta-késleltetésnek nincs semmiféle "idő jellegű" fizikai tartalma, a kétlépéses értékadás-kiértékelés egyazon szimulációs időpillanatban megy végbe, a VHDL szimulátorokban az eseményeknek ez a fajta sorrendisége nem, csak kumulált eredményük figyelhető meg. A 6-7. ábra egy általános célú fel-le számlálót mutat be, amelyben a funkcionalitás megvalósításának alapja a VHDL folyamat *(process)* utasítása.

---

[9] A kijelentés csak a signal típusú értékhordozókra igaz. A VHDL másik értékhordozó-típusa, a változó *(variable)* esetén az érték "azonnal látható". Példáinkban nem használunk változó típusú értékhordozókat.

```vhdl
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity counter is
6      port (clk:          in  std_logic;
7            reset_n:      in  std_logic;
8            enable:       in  std_logic;
9            direction:    in  std_logic;
10           parallel_in:  in  std_logic_vector (3 downto 0);
11           load:         in  std_logic;
12           cout:         out std_logic_vector (3 downto 0));
13 end entity counter;
14
15 architecture rtl of counter is
16
17     signal counter: std_logic_vector (3 downto 0) := (others => '0');
18
19 begin
20
21     L_COUNTER: process (clk, reset_n)
22     begin
23         if ( reset_n = '0' ) then counter <= (others => '0');
24         elsif ( rising_edge(clk) ) then
25             if ( enable = '1' ) then
26                 if ( load = '1' ) then counter <= parallel_in;
27                 elsif ( direction = '1' ) then counter <= std_logic_vector(unsigned(counter) + 1);
28                 else counter <= std_logic_vector(unsigned(counter) - 1);
29                 end if;
30             end if;
31         end if;
32     end process;
33
34     L_OUTPUT: cout <= counter;
35
36 end architecture rtl;
```

A számlálóban már logikai vektorok közötti aritmetikai műveleteket is használunk, ezért szükség van egy ilyen operátorokat tartalmazó csomagra is. (3. sor)

*Az **std_logic** típus több-bites változata az **std_logic_vector**. Zárójelben az MSB és az LSB indexe, tehát a vektor mérete adható meg.* (10. sor)

Belső jel a számláló értékének tárolására. (17. sor)

Az felfutó élre való érzékenyítés egy lehetséges megvalósítása. (24. sor)

6-7. ábra Egy általános célú fel-le számláló áramkör RTL szintű VHDL modellje

A 6-7. ábra VHDL modellje két részáramkört ír le, amelyek mindegyike egy-egy címkével *(L_COUNTER, L_OUTPUT)* van ellátva[10]. Az első összetevő magát a számláló értékét állítja elő a különböző vezérlő bemenetek alapján, a második pedig csupán egy vezetéket modellez, amely a belső számláló regiszter értékét a kimeneti porttal kapcsolja össze. A számlálót leíró folyamat az órajelre *(clk)* és a reset bemenetre *(reset_n)* érzékeny. E két jel bármelyikének megváltozásakor a folyamat törzsében foglalt utasítássorozat lefut. Reset esetén nincs más

[10] A konkurens utasítások különálló áramköri részleteket modelleznek, ezért mindig érdemes őket elnevezni egy-egy címkével, amelyeket a szimulációs és szintézis szoftverek is az adott kódrészlet/részáramkör azonosítására használnak. Hiányukban ezek a programok csak a VHDL kódsor száma alapján tudják az adott utasítást azonosítani, ami nagyon megnehezíti a hibakeresést.

dolgunk, mint nullázni a számlálót. Az órajel tetszőleges megváltozásakor a folyamat lefut, ezért annak érdekében, hogy az áramkör csak a felfutó élek esetén reagáljon a külvilág változásaira, az órajelbemenet változásait tovább kell szűrni. Erre szolgál a *rising_edge()* függvény. A reset bemenetnél is megfigyelhető egy hasonló szűrés. A kétféle szűrés között annyi a különbség, hogy míg a reset esetén bármely változás, ami logikai alacsony szinten ér véget, érvényes átmenetnek bizonyul (pl. a határozatlan értékről logikai alacsony szintre lépés), míg a *rising_edge()* függvény csak azokat a változásokat tekinti érvényes átmenetnek, amelyek erős logikai alacsony szinten *(strong low)* indulnak és logikai magas szinten *(strong high)* érnek véget. Ez a finom különbségtétel a flip-flopok élérzékeny órajelbemenetét hivatott modellezni.

A példacsoport második tagja (6-8. ábra) egy generikus gyűrűs számlálót mutat be.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity ring_counter is
6      generic (width:          integer := 8);
7      port (clk:            in  std_logic;
8            reset_n:        in  std_logic;
9            parallel_in:    in  std_logic_vector (width-1 downto 0);
10           load:           in  std_logic;
11           cout:           out std_logic_vector (width-1 downto 0));
12 end entity ring_counter;
13
14 architecture rtl of ring_counter is
15
16     signal counter: std_logic_vector (width-1 downto 0) := (others => '0');
17
18 begin
19
20     L_COUNTER: process (clk, reset_n)
21     begin
22         if ( reset_n = '0' ) then counter <= (others => '0');
23         elsif ( rising_edge(clk) ) then
24             if ( load = '1' ) then counter <= parallel_in;
25             else
26                 counter(0) <= counter(width-1);
27                 for i in 1 to width-1 loop
28                     counter(i) <= counter(i-1);
29                 end loop;
30             end if;
31         end if;
32     end process;
33
34     L_OUTPUT: cout <= counter;
35
36 end architecture rtl;
```

Generikus paraméter: **width**
A számláló mérete példányosításkor adható meg.
Az alapértelmezett méret 8 bit.

Statikus kiértékelésű for ciklus.
A ciklus fordítási időben kifejthető
a ciklustörzs n-szeri megismétlésével.

6-8. ábra Egy generikus gyűrűs számláló áramkör RTL szintű VHDL modellje

A 6-8. ábra VHDL modellje a(z) 6-7. ábrán láthatóhoz hasonló. Különbség csak abban van, hogy a számláló "mit csinál két órajel-felfutóél között". Az egyszerű fel/le számláló esetén nem okoz gondot annak értelmezése, hogy mi történik az áramkörben két felfutóél között, mert a

vezérlő bemenetektől függetlenül mindig csak egy értékadás valósul meg. A gyűrűs számláló esetében azonban, ha a párhuzamos betöltést jelző vezérlő bemenet inaktív, akkor egy egész sor műveletnek kell végrehajtódnia, hiszen a belső számláló regiszter minden flip-flop-jának értékét frissíteni kell. Az egyes bitek értékének frissítését megfogalmazhatnánk a bitszámnak megfelelő számú értékadással. Nem tudjuk azonban, hogy pontosan hány bitről is van szó, hiszen az csak a példányosítás pillanatában derül ki. Ilyen esetekben alkalmazható a **VHDL ciklus** utasítása, amely - a fentinek megfelelő környezetben - úgy értelmezendő, hogy **az órajel felfutó élénél a ciklustörzs *n*-szeri lefutása megtörténik egyazon szimulációs cikluson belül**. Ebben az esetben a ciklus tulajdonképpen nem több, mint egy egész sor esemény tömör megfogalmazási módja (statikusan kiértékelt ciklus). Figyeljük meg a lényeges különbséget egy a programozási nyelvekben megszokott ciklushoz képest: Jelen esetben a ciklus *n.* lefutásakor a *counter* regiszter *i.* bitje az *i-1.* bitet kapja értékül. Egy ilyen ciklus egy programozási nyelv esetén csak azt eredményezheti, hogy végül minden bit ugyanazt az értéket tárolja. A delta-késleltetés miatt azonban a folyamat kiértékelése során előbb minden jobbérték kiértékelésére kerül sor, majd csak az összes jobbérték-kiértékelést követően indul el a balértékek frissítése. Ezt gyakorlatilag azt jelenti, hogy a ciklus törzsében a jobbérték kifejezés mindig az *i-1.* bit **előző szimulációs ciklusbeli, tehát az előző órajelciklusbeli** értékét adja át az *i.* bitnek. Tehát pontosan azt írtuk le, amit egy gyűrűs számlálótól elvárunk.

6.1.4. Negyedik példacsoport - Shift&Add szorzó egység

Az utolsó példa azt mutatja be, hogy miként lehet egyszerű adatfeldolgozási feladatot ellátó rendszereket modellezni VHDL segítségével. A példa egy *Shift&Add* elvű, generikus szorzó egység ún. FSMD *(Finite State Machine with Datapath)* modelljét írja le. Az elnevezés azt jelenti, hogy a rendszer egy vezérlési funkciót ellátó állapotgépből *(FSM)* és adatfeldolgozó elemeket tartalmazó műveletvégző elemekből *(datapath)* áll. Az alábbi példában ezek a szerkezeti elemek nem különülnek el egymástól a kód szintjén, de bonyolultabb esetekben az egyes vezérlő és műveletvégző erőforrásokat érdemes külön-külön tervezési egységekként megvalósítani. Az ilyen strukturált HDL modellek túlmutatnak e segédlet keretein.

A *Shift&Add* szorzás. A feladat az, hogy a szorzandó értékét annyiszor adjuk hozzá önmagához, amennyi a szorzó értéke (ez tulajdonképpen a szorzás definíciója). A probléma ezzel a naiv megközelítéssel, hogy a számítás időtartama erősen függ a szorzó értékétől, ami nemcsak azt jelenti, hogy bizonyos esetekben nagyon lassan kapjuk meg az eredményt, hanem azt is, hogy a számítási idő nem determinisztikus. A *Shift&Add* szorzás során a szorzó egyes helyiértékein jobbról balra (az LSB-től az MSB felé) végiglépkedünk. Ha a szorzó *n.* helyiértékén egyest találunk, akkor a kezdetben 0 értékű kimeneti regiszterhez hozzáadjuk a szorzandó 2^n-szeresét. Ez úgy valósul meg, hogy a szorzón való végiglépdelés során az akkumuláláson kívül a szorzandó eggyel való balra shiftelését (kettővel való szorzását) is elvégezzük. Így az n. lépésben előáll a szorzandó $2^{(n+1)}$-szerese, amelyet a következő (n+1). iterációban felhasználhatunk az akkumulálás során (feltéve, hogy a szorzó megfelelő bitje logikai 1). Ennek az algoritmusnak az előnye, hogy így a szorzás művelet időigénye csak az operandusok bitszámától, nem pedig azok értékétől függ. Mivel az áramkörnek egy konkrét példánya csak

egyféle operandusmérettel tud dolgozni, a végrehajtási idő mindig ugyanannyi. Az itt bemutatott áramkör hand-shake jellegű kommunikációt valósít meg beágyazó környezetével; a hívó a *request* vonalon jelzi, hogy a bemeneteken elhelyezte az operandusokat és a szorzás művelet elkezdődhet. Ha a szorzó egység a műveletet befejezte, a szorzatot a megfelelő kimeneti portra kapcsolta, a *ready* vonalon jelzi, hogy a szorzat kiolvasható. A 6-9. ábra a szorzó áramkör egyed-deklarációját mutatja.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity multiplier is
6      generic (width: integer range 4 to 32 := 32);
7      port (clk:        in  std_logic;
8            reset_n:    in  std_logic;
9            request:    in  std_logic;
10           ready:      out std_logic;
11           op_a:       in  std_logic_vector (width-1 downto 0);
12           op_b:       in  std_logic_vector (width-1 downto 0);
13           product:    out std_logic_vector (2*width-1 downto 0));
14 end entity multiplier;
15
16 architecture shift_and_add of multiplier is
17     -- az implementáció belső erőforrásai --
18 begin
19     -- a funkció leírása --
20 end architecture shift_and_add;
```

*A szorzó egység hand-shake jellegű interfésszel kommunikál beágyazó környezetével. A **request** vonal logikai magas értékre állításával a hívó jelzi, hogy az operandusokat elhelyezte a bemenetekre, a szorzás művelet elindulhat. A **ready** vonalon a szorzó modul jelzi, ha a szorzat előállt.*

6-9. ábra A Shift&Add szorzó áramkör VHDL modelljének egyed-deklarációja

A 6-10. ábrán a szorzó áramkör belső erőforrásainak deklarációja látható. Az állapotgépek megvalósításának legelterjedtebb módja - nemcsak VHDL-ben, hanem szinte bármilyen formális nyelven - a saját felsorolt típuson alapuló megoldás.

```
   -- könyvtárak és csomagok deklarációja --
   -- egyed-deklaráció --

16 architecture shift_and_add of multiplier is
17
18     type state_t is (wait_for_request,s1,s2,hold_ready);
19     signal state: state_t := wait_for_request;
20     signal op_a_reg:    std_logic_vector (2*width-1 downto 0) := (others => '0');
21     signal op_b_reg:    std_logic_vector (width-1 downto 0) := (others => '0');
22     signal product_reg: std_logic_vector (2*width-1 downto 0) := (others => '0');
23     signal i:           integer range 0 to width := 0;
24
25 begin
26     -- a funkció leírása --
27 end architecture shift_and_add;
```

Saját felsorolt típus, amelynek lehetséges értékei az állapotgép állapotainak feleltethetők meg. Létrehozunk egy jelet is ezzel a típussal, amely az állapotregisztert reprezentálja.

A VHDL-ben az előjeles egész típus beépített típusként is megtalálható. Az std_logic_vector aritmetikai könyvtárai általában ezzel a típussal is kompatibilisek.

6-10. ábra A Shift&Add szorzó áramkör VHDL modelljének belső erőforrásai

Az egyedileg definiált felsorolt típus (6-10. ábra: 18. sor) lehetséges értékei az állapotgép egyes állapotait reprezentálják. Magának a típusnak a létrehozása után természetesen szükségünk

van egy értékhordozó erőforrásra is, amelynek típusaként az új felsorolt típust adjuk meg. Esetünkben ez egy jel *(signal)* lesz (6-10. ábra: 19. sor), amely az áramkör belső állapotát reprezentálja. Az áramkör belső működését egyetlen, az órajelre és a reset bemenetre érzékenyített folyamat írja le, amely szerkezetét tekintve hasonló ahhoz, amit a számláló áramkörök példájában láthattunk; a reset bemenet aktiválásának hatására minden belső erőforrás (beleértve a belső állapotot reprezentáló *state* jelet) és kimenet alapállapotba kerül (6-11. ábra: 29-31. sor).

```
-- könyvtárak és csomagok deklarációja --
-- egyed-deklaráció --

16 architecture shift_and_add of multiplier is
17     -- az implementáció belső erőforrásai --
25 begin
26     L_FSM:  process (clk, reset_n)
27     begin
28         if ( reset_n = '0' ) then
29             state <= wait_for_request; ready <= '0'; op_a_reg <= (others => '0');
30             op_b_reg <= (others => '0'); product_reg <= (others => '0');
31             product <= (others => '0'); i <= 0;
32         elsif ( rising_edge(clk) ) then
33             case state is
34                 when wait_for_request =>
35                     if ( request = '1' ) then
36                         ready <= '0';
37                         op_a_reg(2*width-1 downto width) <= (others => '0');
38                         op_a_reg(width-1 downto 0) <= op_a;
39                         op_b_reg <= op_b; i <= 0; product_reg <= (others => '0');
40                         state <= s1;
41                     end if;
42                 when s1 =>
43                     if ( op_b_reg(i) = '1' ) then
44                         product_reg <= std_logic_vector(unsigned(product_reg) +
45                                                         unsigned(op_a_reg));
46                     end if;
47                     i <= i + 1; op_a_reg <= std_logic_vector(shift_left(unsigned(op_a_reg), 1));
48                     state <= s2;
49                 when s2 =>
50                     if ( i = width ) then
51                         product <= product_reg; ready <= '1'; state <= hold_ready;
52                     else state <= s1;
53                     end if;
54                 when hold_ready => state <= wait_for_request;
55                 when others => report "FAIL!" severity failure;
56             end case;
57         end if;
58     end process;
59 end architecture shift_and_add;
```

6-11. ábra A Shift&Add szorzó áramkör VHDL modelljének implementációja

Az órajel felfutó élére való érzékenyítés a számlálóknál látottakkal azonos módon történik. Jelentős különbség van azonban az órajel felfutó éle esetén végbemenő folyamatok leírásában. Mivel az áramkör a szorzás művelet különböző fázisaiban más-más viselkedést kell hogy

megvalósítson, ezért az elvégzendő műveleteket is a számítás fázisától, vagyis a rendszer állapotától tesszük függővé. A rendszer állapotát a *state* jel írja le, tehát ezzel a jellel kell létrehoznunk egy feltételes szerkezetet, amelyre - sok lehetséges feltétel esetén - a *case* utasítás (a jól ismert programozási nyelvek *switch-case* szerkezetével azonos nyelvi elem) a kézenfekvő megoldás. Az áramkör tehát az órajel felfutó élénél a *state* jeltől - vagyis a rendszer állapotától - függően más-más műveletsort végez el.

6.2. HDL-alapú lineáris tesztkörnyezet készítése

A hardverleíró nyelvű RTL modellek funkcionális verifikációjának eszköze egy ugyancsak hardverleíró nyelven specifikált tesztkörnyezet, ún. **testbench**, amelynek feladata, hogy példányosítsa a verifikálandó tervezési egységet (*Design Under Test*, DUT[11]), előállítsa annak **gerjesztő jeleit** és **megfigyelhetővé tegye** az egység kimeneteit. Attól függően, hogy a verifikálandó modul belső állapota és belső jelei milyen mértékben figyelhetők meg, a következő esetekről beszélhetünk:

- **Black-box** verifikáció esetén a DUT belső állapota és belső jelei nem hozzáférhetők. Ez akkor fordulhat elő, ha a DUT egy ún. IP *(Intellectual Property)*, az azt készítő cég “szellemi terméke”.
- **Grey-box** verifikáció esetén a DUT belső jeleinek egy részhalmaza megfigyelhető. A megfigyelhető jelek általában éppen a verifikáció megkönnyítése céljából férhetőek hozzá.
- **White-box** verifikáció esetén a DUT belső állapota, jelei és működése teljes mértékben megfigyelhető.

Esetünkben természetesen white-box verifikációról van szó, hiszen magát a DUT-t is mi készítettük, rendelkezésre áll a teljes RTL modell. A tesztkörnyezet a DUT-hez hasonlóan egy VHDL egyed lesz, amelynek legfőbb jellegzetessége, hogy deklarációja teljesen üres, nem tartalmaz sem generikus paramétereket, sem portokat. Ennek oka, hogy a testbench tulajdonképpen a DUT-t körülvevő külvilágot modellezi, így interfészjelekről nem beszélhetünk, és mivel ennek a modulnak a példányosítására sosem kerül sor, ezért a generikus paraméter fogalma is értelmét veszti[12].

---

[11] A DUT kifejezés elterjedten használt, bár pontatlan. Szigorúan véve nem tesztelésről, hanem verifikációról van szó. Tesztelés alatt a gyártás utáni, *példányonkénti* kipróbálást, verifikáció alatt pedig az absztakt modellek funkcióinak leellenőrzését értjük. Ennek ellenére e két fogalmat a verifikációval foglalkozó szakirodalom is gyakran szinonímaként használja.

[12] Úgy is fogalmazhatnánk, hogy a testbench az univerzumnak egy erősen leegyszerűsített modellje :).

A 6-1. ábra AND kapujának verifikációjára alkalmas testbench modult a 6-12. ábra mutatja. A tesztvektorokat előállító nyelvi elem általában az **érzékenységi lista nélküli folyamat, amely egy egyetlen szimulációs ciklusban végtelenül ismétlődő utasítássorozat**ként értelmezendő. A ciklikus ismétlődés azonban ebben az esetben gyakorlatilag nem valósul meg, mivel a futást a *wait* utasítás különböző változatai felfüggesztik. A folyamat utolsó *wait* utasítása végleg felfüggeszti a futást, így az egyébként végtelen ciklust leíró folyamat tulajdonképpen egyszer sem fut le teljesen.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity testbench_logic_gates is
5  end entity testbench_logic_gates;
6
7  architecture behavior of testbench_logic_gates is
8
9      signal a:       std_logic := '0';
10     signal b:       std_logic := '0';
11     signal y:       std_logic;
12
13 begin
14
15     L_DUT: entity work.and_gate(a1)
16                port map (a => a,b => b,y => y);
17
18     L_GENERATE_TEST_VECTORS: process
19     begin
20         wait for 100 ns; a <= '0'; b <= '1';
21         wait for 100 ns; a <= '1'; b <= '0';
22         wait for 100 ns; a <= '1'; b <= '1';
23         wait for 100 ns; a <= '0'; b <= '0';
24         wait;
25     end process;
26
27 end architecture behavior;
```

A testbench interfésze mindig üres. A generikus paramétereknek és a portoknak itt nincs szerepe.

A testbench belső jelei a DUT bemeneteinek és kimeneteinek értékét hordozzák. A DUT által generált jeleknek a testbench nem adhat értéket.

A DUT példányosítása és portjainak bekötése.

A gerjesztő jelek előállítását érzékenységi lista nélküli folyamatok végzik.

6-12. ábra Az AND kapu verifikációjára alkalmas tesztkörnyezet

A fenti példából jól látható, miért nevezzük a tesztkörnyezetek e típusát *lineáris* testbench-nek. Az egyes tesztvektorok megadása egyesével, egymás után történik. A verifikáció alatt álló modul válaszának megvizsgálása után a következő tesztvektor előállítása manuálisan történik. Ez a fajta verifikációs módszer a hullámforma-megjelenítőkkel együtt alkalmazva viszonylag egyszerű rendszerek esetén hatékonyan alkalmazható, de összetett SoC áramkörök teljes funkcionalitásának verifikációjára a szükséges tesztvektorok óriási száma miatt már nem alkalmas. Az ilyen rendszerek magas szintű verifikációs módszerei (regressziós tesztek futtatására alkalmas, ún. "öntesztelő" self-checking tesztkörnyezetek, eRM, UVM stb.) kívül esnek e segédlet témakörén.

A HDL-alapú lineáris testbench-ekben alkalmazott nyelvi elemek természetesen az egyszerű értékadásnál bonyolultabbak is lehetnek. A negyedik példacsoport *Shift&Add* szorzó egységének tesztkörnyezetét mutatja a 6-13. ábra.

```
1  -- könyvtárak és csomagok deklarálása --
2  -- egyed-deklaráció --
3
7  architecture behavior of testbench_multiplier is
8
9     constant clock_period:  time := 100 ns;          A később generált órajel periódusidejének definiálása.
10    -- gerjesztő jelek és a DUT kimeneti jeleinek deklarálása --
18
19  begin
20
21    L_DUT:  entity work.multiplier(shift_and_add)
22              generic map (width => 8)
23              port map (clk => clk,reset_n => reset_n,
24                        request => request,ready => ready,
25                        op_a => op_a,op_b => op_b,
26                        product => product);
27
28    L_TEST_SEQUENCE: process
29    begin
30        wait for 356 ns; reset_n <= '1';
31        wait for 345 ns;
32        op_a <= X"12"; op_b <= X"04"; request <= '1';        Hand-shake kommunikáció
33        wait until rising_edge(ready); request <= '0';       viselkedési modellezése.
34        wait for 500 ns;
35        op_a <= X"13"; op_b <= X"03";  request <= '1';
36        wait until rising_edge(ready); request <= '0';
37        wait;
38    end process;
39
40    L_CLOCK: process begin wait for clock_period / 2; clk <= not clk; end process;    Órajel-generálás
41
42 end architecture behavior;
```

6-13. ábra A Shift&Add szorzó tesztkörnyezete

A szorzó áramkör tesztkörnyezetében megfigyelhető, hogy a DUT-vel való kölcsönhatás megfogalmazására az egyszerű értékadásnál “intelligensebb” nyelvi szerkezetek is használhatók. Az is látható, hogy egy DUT gerjesztő jeleit nem feltétlenül egyetlen folyamat állítja elő.

A testbench továbbá egyéb tervezési egységeket is példányosíthat. A szorzó áramkör egyszerű hand-shake kommunikációjának modellezésére elegendő egyetlen folyamat definiálása, de bonyolultabb kommunikációs protokollok esetén (pl. UART, SPI, PCI stb.) indokolt lehet különálló VHDL modulok kifejlesztése kifejezetten a kommunikációs interfészek verifikálása céljából. Ezeket a verifikáció komponenseket példányosítva és a DUT-vel összekapcsolva olyan testbench-et kapunk, amely a DUT-t egy a későbbi felhasználásának helyét szimuláló környezetbe helyezi. Ez a megoldás már kevéssé nevezhető lineáris testbench-nek, és már a korábban említett magas szintű verifikációs módszerek irányába mutat.

6.3. Ajánlott irodalom

[1] Peter J. Ashenden, **Digital Design - An Embedded Systems Approach Using VHDL**, ELSEVIER, 2008

[2] David Money Harris, Sarah L. Harris, **Digital Design and Computer Architecture**, 2nd Edition, ELSEVIER, 2012
[3] Richard S. Sandige, Michael L. Sandige, **Fundamentals of Digital and Computer Design with VHDL**, McGraw-Hill Companies, Inc., 2012
[4] Sanjay Churiwala, Sapan Garg, **Principles of VLSI RTL Design - A Practical Guide**, SPRINGER, 2011
[5] Pong P. Chu, **RTL Hardware Design Using VHDL - Coding for Efficiency, Portability, and Scalability**, John Wiley and Sons, Inc., 2006
[6] Peter J. Ashenden, Jim Lewis, **The Designer’s Guide to VHDL**, 3rd Edition, ELSEVIER, 2008