Hosszú Gábor - Keresztes Péter

VHDL ALAPÚ RENDSZERTERVEZÉS

Budapest, 2006.

Tartalomjegyzék

1. Bevezetés

A „VHDL alapú rendszertervezés” c. könyv célja megismertetni az olvasót az elektronikai áramkörök és rendszerek tervezésének egy nagyon jól használható eszközével, amely leíró nyelvként szolgál a különböző tervező rendszerek bemenetéhez.

Az elektronikai rendszertervezés szakterületére jellemző, hogy állandó fejlődésben, átalakulásban van — nap, mint nap jelennek meg új igények, amelyeket csak az addiginál összetettebb digitális rendszerrel lehet kielégíteni. Mindezek motorja a technológiai fejlődés, amely a mikroelektronikai eszközök jellemzőinek folyamatos javulásában nyilvánul meg.

A jelen könyv segítséget kíván nyújtani mindazoknak, akik valamilyen digitális elektronikai rendszert szeretnének kifejleszteni, továbbá tankönyvként szolgál a VHDL-lel kapcsolatos egyetemi, főiskolai képzések keretében tartott tantárgyakhoz, de felhasználható más tanfolyamok keretében és azok számára is, akik egyénileg kívánnak ezzel a tématerülettel megismerkedni.

A könyvnek a fentieken kívül célja a VHDL alapú rendszertervezés magyar szaknyelvének fejlesztése is. Egyes esetekben tudatosan alkalmaztunk szenvedő szerkezetet, mivel a legfontosabb cél az volt, hogy a könyv nyelvezete tömör legyen. A mai felgyorsult korban ugyanis akkor lehet a nyelvünket megőrizni és fejleszteni, ha az alkalmazkodik a mostani követelményekhez.

Budapest, 2006. szeptember

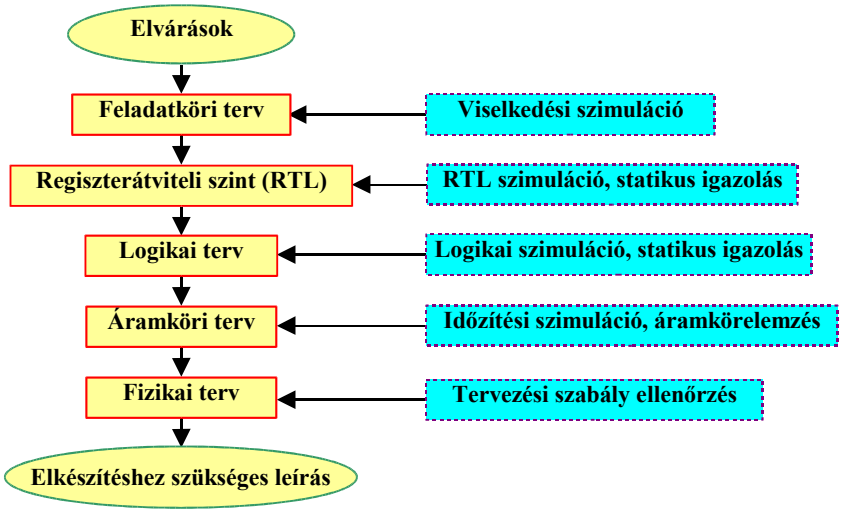
A szerzők

2. A VHDL mint rendszertervező eszköz

2.1. A gépi tervezés

A gépi leíró nyelvek (HDL) célja az egyes termékek egységesítése, logikai szimulációhoz leíró nyelv biztosítása, a terv hierarchikus felépítésének tükrözése és a nagy tervek áttekinthetővé tétele. Többféle nyelv (VHDL, Verilog, Model) van, de a VHDL vált világszabvánnyá. A HDL nyelvek lehetővé teszik a mérnök számára, hogy a terv felépítését és feladatköri jellemzőit a hagyományos kapuszintnél magasabb elvonatkoztatású megjelenési szinten írja le.

A következő ábra egy szokásos tervezési folyamatot mutat be a kezdeti elvárásoktól a gyártó számára való leírásig. Ez utóbbi a programozható kapumátrixok esetén azt a programot jelenti, amivel fel kell programozni az eszközt, hogy a kívánt áramkörként működjön.



2.1-1. ábra. Digitális rendszer szokásos tervezési folyamata.

2.2. A VHDL általános jellemzése

A VHDL a „VHSIC Hardware Description Language” (VHSIC Hardver Leíró Nyelv), a VHSIC a „Very High Speed Integrated Circuits” (Nagyon Nagy Sebességű Integrált Áramkörök) rövidítése. A nyelv a 80-as évek elejéről származik, jelenleg egyike a legfontosabb

szabványos nyelveknek az ***előírás*** (specification), az ***igazolás*** (verification) és az elektronikai tervezés területén. Ez annak köszönhető, hogy a nyelv alapleírása teljesen eszközfüggetlen, ezért a megfelelő szintéziseszközök széles skálája készülhetett el hozzá.

Az amerikai villamosmérnök szervezet, az ***Amerikai Villamosmérnökök Egyesülete*** (**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers, **IEEE**, **IE^3**) 1987-ben szabványosította, a szabvány neve: IEEE 1076-1987 és 1993-ban létrehozták a VHDL szabvány továbbfejlesztett változatát. Az a tény, hogy 1987. óta nemzetközi szabvány, így ***hivatalosan támogatott***, nagymértékben hozzájárult az egyre szélesebb körű felhasználásához. Így a VHDL-re jellemző a **tervek átcserélhetősége**, a VHDL modellek garantáltan futnak bármelyik rendszeren, ezzel támogatja a csoportmunkát.

A VHDL egy olyan általános hardvermodellező nyelv, amely a digitális áramkörök különböző elvonatkoztatási szinteken történő, egységes leírására alkalmas. Felhasználható a legkülönbözőbb elektronikai rendszerek tervezésére, különböző cél-technológiákkal, pl. tervezhetünk vele szilícium alapú integrált áramkört, hibrid IC-t, nyomtatott áramkört vagy éppen programozható kapumátrixot is. Ilyen programozható kapumátrix áramkör a Xilinx FPGA vagy az Altera EPLD. Ez utóbbiak kész integrált áramkörök, amelyek felprogramozhatók úgy, hogy különböző felhasználó által tervezett áramkörként működjenek. Így segítségükkel egészen kis darabszám esetén is lehetőség nyílik integrált áramkör alkalmazására digitális berendezésekben.

A VHDL nyelv kiemelkedő tulajdonságai, hogy nyelvi szerkezete átgondolt, és technológia-független. Ez utóbbiból adódik, hogy nem kötődik egyetlen szimulátorhoz vagy adatbázishoz sem, és nem erőltet rá a tervezőre egyetlen tervezési módszertant sem. A VHDL lehetővé teszi továbbá az új technológiák beépítését már létező tervekbe. A **technológia függetlenség**, azaz berendezés és IC gyártási eljárástól független, el lehet választani egy szintet, ahol még technológia független a terv, de az egyes logikai kapukhoz ezután a különböző technológiáktól függően különböző könyvtárakat lehet rendelni (CMOS, NMOS, GaAs, FPGA, diszkrét elemek, stb.). Így technológiaváltás után is használhatóak a tervek.

Használatának előnyei közé tartoznak az elektronikai terv fejlesztési idejének lerövidítése és a terv későbbi módosításának egyszerűsítése. A VHDL-t eredetileg előírási és modellező nyelvnek szánták. Az első szimulátorokat a 80-as évek végén fejlesztették ki. Mivel a VHDL nyelv szabványos, a különböző VHDL alapú tervezési eszközök bemeneti és kimeneti felületei megfeleltethetők egymásnak. Sajnos a VHDL-t nem szabványosították áramkörszintézisre, ugyanakkor az utóbbi időben a számítógépi teljesítmény lehetővé teszi az áramkörszintézist is.

A VHDL elképzelése az, hogy ún. egyedeket határoz meg, melyeket külső és belső oldalról közelít meg. Ezek az egyedek (összetevők, áramkörök, rendszerek) megoszlanak egy külső, látható (egyednév és kapcsolódások), és egy belső vagy rejtett rész (egyed algoritmus és megvalósítás) között. Az egyedek egymást ezen a külső határfelületen keresztül használhatják.

A VHDL tervezési és modellezési nyelv. Segítségével a számítógép és az ember számára is olvasható alakban le lehet írni a digitális gépi rendszerek, áramköri kártyák és az összetevők, alkatrészek **szerkezeti felépítését** és **feladatkörét**. Egy tervezési tárgy VHDL nyelvű leírását **VHDL modell**nek hívjuk.

A VHDL nyelv azért jobb a többi gépi leíró nyelvnél, mert az ezekkel szembeni különböző elvárások **mindegyikét** kielégíti, pl. **támogatja a különböző tervezési**

módszereket és tervezési technikákat. Ennek keretében különböző tervezési módszereket támogat, pl. a fentről-lefelé, ill. az alulról-felfelé (könyvtárra támaszkodó) módszert. Hasonlóan különböző tervezési technikákat támogat, pl. szinkron és aszinkron, ill. PLA és random logika.

A VHDL egy ***egyidejű*** (concurrent) programozási nyelv, ezért kifejezetten alkalmas igen nagy bonyolultságú digitális rendszerek viselkedésének leírására.

2.3. Tervezési eljárás VHDL-lel

Egy berendezést különböző szinteken lehet leírni, attól függően, hogy az egész rendszer működését akarjuk ábrázolni, vagy annak csak egy részegységét. Ezen kívül az egyes leírási módok különböznek attól függően, hogy az egység működésmódját írjuk le, vagy pedig a szerkezeti felépítésére, röviden a ***szerkezet***ére (structure) vagyunk kíváncsiak. A VHDL mindkét fajta leírást lehetővé teszi. Segítségével egy tetszőleges digitális áramkört különböző elvonatkoztatási szinteken modellezhetünk. Ezt tükrözi, hogy a VHDL-ben megfogalmazott feladatok különböző három fő elvonatkoztatási szinten írhatók le:

- ***viselkedési*** (behavioral) leírás: a terv feladatköri vagy algoritmikus megadása sorozatos VHDL folyamatban leírva.
- ***adatáramlási*** (dataflow) leírás: az adatokat a terv bemenetétől kimenetéig, mint egy folyamat elemeit írjuk le, az egyes lépések egyszerű adat-átalakítások.
- ***szerkezeti*** (structural) leírás: a modellben ***összetevők*** (components) egymáshoz való kapcsolódása van megadva.

Jellemző rá a **leíró képesség széles tartománya**, támogatja a digitális rendszer tisztán viselkedési, azaz feladatköri leírását, de támogatja a tisztán szerkezeti modellezését és támogatja a fő logikai egységeivel és az azok közötti kapcsolattal való leírását is. Így pl. elképzelhető, hogy egy nagyobb rendszert szerkezeti leírással kisebbekre bontanak, amiket már viselkedési leírással lehet modellezni.

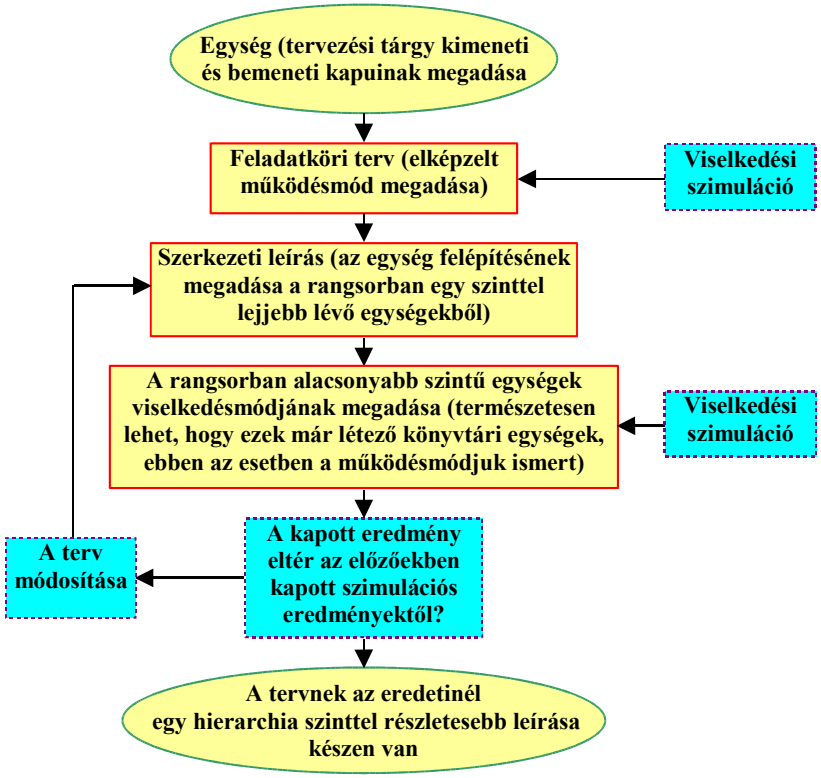
A VHDL alapú tervezés általában a felülről-lefelé és az alulról-felfelé módszer együttes felhasználásával történik. Ennek során, miután meghatározásra került az egész áramkör általános működése, az áramkör funkcionális részegységekre lett lebontva, majd ezen egységek működésének viselkedési leírásának, és határfelületeiken keresztüli összekapcsolódásának meghatározása következik. Természetesen az egész áramkör ki- és bemeneteit is megadjuk.

Amikor egy áramköri modellt meghatároztunk, szimulálhatjuk a viselkedési leírásának (vagy az összetevői viselkedési leírásának) végrehajtásával. Ennek során az adott időtartamot diszkrét lépésekben vizsgáljuk. Egy adott szimulációs időpontban az áramköri egység bemenetén egy bemeneti kapocs értékének változását látjuk, ha az új érték különbözik az előzőtől.

Ez a tervezési lépés a leírásokon kívül magába foglalja a részegységek, illetve a teljes áramkör szimulációját, működési helyességének ellenőrzését. A következő lépés (alulról-felfelé) a részegységek szerkezeti szintű leírása, szabvány elemek felhasználásával. A részegységek szimulációja után azok összeköttetésének megadása következik, majd az egész rendszer szimulációja. A végső ellenőrzés a kétféle leírás szimulációs eredményeinek összehasonlítása.

A részegységek modellezésénél optimalizálás is végezhető azzal, hogy a problémás részegységekre több eltérő megvalósítást adunk. Ezek a megvalósítások aztán a vizsgálni kívánt paraméterek alapján összehasonlíthatók.

A ***szintézis*** (synthesis) mindig két elvonatkoztatási tervezési szint között történik, amelynek során egy magasabb elvonatkoztatási szintről egy alacsonyabbra jutunk úgy, hogy közben többlet információkat építünk be a modellbe. A tervezés menete fentről lefelé haladva az egyszerűbb tervekből a részletesebbek felé haladva szintézislépésekből áll. Egy ilyen lépést mutat be a következő ábra.



2.3-1. ábra. Egy szintézis lépés a VHDL alapú tervezés folyamatában

3. A VHDL nyelv alapszabályai

3.1. Tárgyak és adattípusok

Egy VHDL forrás fájl lefordítását bináris alakba a fájl ***elemzés***ének (analyzing) nevezik. Ebben az ütemben észlelik a leírás hibáit. A bináris fájlnak a tervezési környezetbe való beolvasását ***kidolgozás***nak (elaboration) nevezik. A lefordított leírások bináris ábrázolását ***tervezési egység***eknek (design unit) nevezik, és könyvtárakba gyűjtik ezeket. A könyvtárakat szimbolikus nevekkel jelölik, az alapértelmezett könyvtárat ***munka*** (work) könyvtárnak hívják.

3.1.1. A VHDL szabványban rögzített típusfajták

A VHDL erősen típusos nyelv; minden tárgynak van egy típusa, és csak ilyen típusú értéket vehet fel. A tárgy típusát a ***bejelentés***ben (declaration) rögzítik. A típusok előre meghatározott fajtájúak lehetnek. Ezek közül *skalár* típusok a ***fizikai*** (physical), a ***lebegőpontos*** (floating point), a ***felsorolás*** (enumeration) és az ***egész*** (integer). Az összetett típusok a ***rácsrend*** (array) és a ***jegyzék*** (record). Ezen kívül lehetőség van ***hozzáférés*** (access) és ***fájl*** (file) bejelentésére is. Ez utóbbi esetén külső fájlokat lehet meghatározni, amelyből a szimulátor olvashat, vagy amelyekbe írhat.

A kettős kötőjel „--” azt jelzi, hogy a sor további része megjegyzés.

Az egyes tárgyak adattípusait mindig statikusan határozzák meg, és a VHDL csak azonos típusú elemeket enged csatlakoztatni egymáshoz, így a tervező még a terv szimulációja előtt felfedezheti az ilyen jellegű tervezési hibákat.

A) A fizikai típus

A ***fizikai*** (Physical) típus egy tétel mért mennyiségét írja le. Ezt egy kijelölt tartományban való mérés alapegységének többszörösével fejezik ki, pl.:

```
type measure is range 0 to 100
units
  mm;
  cm=10mm;
end units;
```

B) A lebegőpontos típus

A ***lebegőpontos*** (Floating point) típus számok gyűjteményét határozza meg, amelyek a valós számok közelítését nyújtják. Ezzel kapcsolatban van egy gond, miszerint nem lehetséges a végtelen hosszú valós számok hardveres kezelése. A lebegőpontos számok típuselőírására egy példa:

```
type half_hour is range 0.0 TO 29.99;
```

A lebegőpontos típuselőírásoknál a tartományhatárokat lebegőpontos számokkal adják meg:

```
type fraction is range -1 + 0.1E-10 to 1 - 0.1E-10;
```

C) A felsorolás típus

A ***felsorolás*** (Enumeration) típus értékek igényhez szabott halmazának meghatározását teszik lehetővé. A szövegelemek lehetnek azonosító szövegelemek (betűk, aláhúzások, számjegyek, stb.), vagy karakteres szövegelemek. A ***szövegelem*** (literal) olyan szókészleti elem, amely önmagát képviseli. Pl. a „184” számjegyek egy tízes alapú szövegelemet képviselnek, amely a 184, mint egész szám.

A felsorolt típusok azonosítókból vagy felsorolt nyomtatható ASCII karakterekből állhatnak. Míg az azonosítóknál nem különböztetjük meg a kis- és nagybetűket, addig a karakteres felsorolt típusban igen. Pl.:

```
type threeLevelLogic is ('0', '1', 'X');
```

Az állapotgépek modellezése a felsorolt adattípusok használatának szokásos területe. Az alábbiakban két felsorolt típus előírást láthatunk, majd egy jel bejelentést.

```
type wireColor is (red, black, green);
type stateType is (red, yellow, green);
signal trafficLight: stateType;
```

A VHDL szimulációban a *trafficLight* jel a három előre meghatározott érték valamelyikét fogja kapni. Ezek az önmagyarázó jelnevek megkönnyítik a terv szimulációját. A szintézis eszközök támogatják a felsorolt adattípusokat. A legutóbbi példában a három felsorolt érték megjelenítésére egy 2-bites vektor értékeit használja a szintézis eszköz.

D) Az egész típus

Az ***egész*** (Integer) típus pozitív vagy negatív egész számok halmazait határozhatja meg. A tartományuk gépfüggő, a legnagyobb ill. legkisebb értékei jellemzően ±2.147.483.648 a 32-bites rendszerekben. Az egész típusú változók bejelentésre láthatók a következőkben példák:

```
type byte is range -128 to 127 ;
type bitPosition is range 7 downto 0 ;
type decimalInt is range -1E9 to 1E9 ;
type testInt is range -100 to 100;
type integer is range -2147483648 to 2147483647;
```

A bejelentésbeli határfeltételek előírják a megadott típusra vonatkozó határértékeket, az ezen kívül eső értékek az ilyen típusú tárgyak számára nem megengedettek. Az „integer” szó nincs lefoglalva a VHDL-ben, ezért lehetséges az, hogy a fenti példák közül az utolsó egy „integer” nevű konkrét típus bejelentése.

E) A rácsrend típus

Egy nevesített ***rácsrend*** (Array) azonos típusú elemek gyűjteménye, amely egy vagy több dimenzióban alakítható ki. Az egyes rácsrend elemek egy vagy több indexértékkel hivatkozhatók, pl.:

```
type array10 is array (0 to 10) of integer;
```

A **bit_vector** a következőképpen határozható meg:

```
type bit_vector is array (natural range <>) of bit;
```

A *bit_vector* előbbi meghatározását tartalmazza a *Standard* csomag, így ez a vektortípus külön bejelentés nélkül használható a VHDL tervekben

F) A jegyzék típus

A ***jegyzék*** (Record) típus egy összetett típus, amelynek elemei különböző típusúak lehetnek. A célja különböző típusú tárgyak csoportba foglalása azért, hogy így egyetlen tárgyként lehessen kezelni azokat. A record ***jelölésmód***ja (syntax):

```
type <identifier> is record
  record meghatározás
end record;
```

Néhány példa a record típus előírására:

```
type coordinates is record
xValue,yValue,zValue:integer;
end record;
```

G) A hozzáférés típus

A ***hozzáférés*** (access) típus a C nyelvbeli ***mutató***ra (pointer) hasonlít, ***LIFO*** (**L**ast **I**n **F**irst **O**ut, ***utolsónak be elsőnek ki***) és ***FIFO*** (**F**irst **I**n **F**irst **O**ut, ***elsőnek be elsőnek ki***) modellezésre célszerű, mert láncolt listák megjelenítésére alkalmas. Ehhez két előre meghatározott függvény, a **new location** és a **deallocate** áll a VHDL-ben rendelkezésre. Csak változókat lehet bejelenteni hozzáférés típusúnak. Ezekre nem a nevükkel, hanem egy hozzáférési értékkel történik a hivatkozás. Az alábbiakban egy példát láthatunk a hozzáférés típusú változó bejelentésére és a fenti függvények alkalmazására.

```
type mutatoHely is access location;
variable mH1, mH2, mH3: mutatoHely;
mH1:= new location; -- új tárgy jött létre, amire mH1 mutat
mH2:= mH1;          -- mH2 arra a tárgyra mutat, mint mH1
mH3:= new location; -- új tárgy jött létre, amire mH3 mutat
mH1:= mH3;          -- mH1 innentől arra a tárgyra mutat, mint mH3
deallocate(mH2);    -- mH2 most nullára mutat
```

H) A fájl típus

A ***fájl*** (File) típus lehetővé teszi külső fájlok bejelentését, amelyek előírt típusú tárgyakat tartalmaznak. A fájl típus pontos meghatározása a fordító és szimulátor programtól függ.

3.1.2. Vektorok létrehozása

Két módon lehet bejelenteni vektorokat VHDL-ben, a legkisebb indexű elemtől a legnagyobbikig **to**-val és a fordított irányban **downto**-val, amint azt a következő két példa bemutatja:

```
signal a:std_logic_vector(0 to 3);
signal b:std_logic_vector(3 downto 0);  --  Javasolt
```

Az ajánlott módszer az, hogy a vektorokat mindig **downto**-val érdemes bejelenteni. Ennek az oka az, hogy ha a vektorokat **downto**-val jelentik be, akkor a ***legfontosabb bit*** (**M**ost **S**ignificant **B**it, **MSB**) mindig a legnagyobb indexű lesz, míg **to**-val való bejelentésnél az MSB lesz a 0 indexű.

3.1.3. Csomagokban meghatározott típusok

A) A bit típus

Az **Std** könyvtárbeli **Standard** egy olyan csomag, amelyet alapértelmezésben a futása kezdetén beolvas a VHDL fordító vagy szimulátor, így az abban meghatározott, szabványos típusok mindig rendelkezésre állnak. Ilyenek pl. az egész típusúak közül az „Integer”, a lebegőpontos „Real”, a felsorolt típusú „Boolean”, „Bit”, „Severity_level”, a „Character” és a „Time”. Közülük az Integer és a Real széles tartománnyal vannak meghatározva, a Boolean két lehetséges értéke a „False” és a „True”, a Bit-é a '0' és az '1', a Character pedig tartalmazza a 128 ASCII karaktert.

B) Az std_ulogic és az std_logic típus

A Standard csomagban szerepel a bit típus meghatározása, de ennek csak '0' és '1' értéke lehet. Ebből adódó megszorítás, hogy nem lehetséges leírni vele a harmadik állapotot, nem lehet azonos jelnek több meghajtója, nem lehet *ismeretlen* vagy ***érdektelen*** (don't care) értéket a jelhez hozzárendelni. Nem lehet továbbá leírni a felhúzást (gyenge egy), a lehúzást (gyenge nulla) és azt, hogy a jel még nem kapott kezdőértéket.

Ezen segít saját adattípusok használata, de így a VHDL kód eszközfüggővé válhat, ezért mindenképpen érdemes szabványos csomagokban meghatározott típusokat használni. Ilyen szabványos csomag pl. az **IEEE** által kidolgozott **std_logic_1164** csomag, amely az *std_logic* és az *std_ulogic* típusokat tartalmazza. Az *std_logic_1164* csomagot rendszerint az *ieee* nevű könyvtárban tárolják szemben a *standard* csomaggal, amelyet az *std* könyvtárban helyeznek el. Ezeknek a típusoknak a következő táblázatban bemutatott logikai értékeik lehetnek:

Logikai érték	Jelentés magyarul	Jelentés angolul
'U'	Kezdőértékre nem állított	Uninitialized
'X'	Kényszerített ismeretlen	Forcing Unknown
'0'	Kényszerített 0	Forcing 0
'1'	Kényszerített 1	Forcing 1
'Z'	Nagy impedanciájú	High Impedance
'W'	Gyenge ismeretlen	Weak Unknown

Logikai érték	Jelentés magyarul	Jelentés angolul
'L'	Gyenge 0	Weak 0
'H'	Gyenge 1	Weak 1
'-'	Nem számít	Don't care

3.1-1. táblázat. Az std_logic és az std_ulogic típus logikai értékei

Egy példa az 'X' értékre:

```
if John (6 downto 0) = "1XXXX0X" then ...
```

ami a szintézis során a következő alakot kapja:

```
if (John(6)='1' and John(1)='0') then ...
```

A 'Z' érték jelentése mind a szimuláció, mind a szintézis során: nagy impedancia. Ha egy céltechnológia (pl. egy FPGA) nem ismeri a harmadik állapotot, akkor megoldható a gond a jel ***nyalábolás***ával (multiplexing).

Az **std_ulogic_vector** a következőképpen van meghatározva:

```
type std_ulogic_vector is array (natural range <>) of std_ulogic;
```

Az **std_logic_vector** pedig az alábbiak szerint:

```
type std_logic_vector is array (natural range <>) of std_logic;
```

3.1.4. Jelek, változók és állandók

A tárgyak előírt típusú értékek tartályai, lehetnek ***jel***ek (signal), ***változó***k (variable) vagy ***állandó***k (constant). Ha a tárgyat egy bizonyos típusúnak jelentettek be, a tárgyon a műveletek a ***típus bejelentés***ben (type declaration) beállított korlátok között végezhetők csak el. Ha a különböző típusú tárgyak keverednének vagy túllépnék a típus bejelentésben beállított határolóikat, hiba lép fel. Az alábbi VHDL listán néhány példa látható az állandó, a változó és a jel bejelentésére.

```
constant romMeret: integer:= 16#FFFF#;
variable hiba: boolean;
signal engedelyez: bit;
signal clk, clear: bit:= '0';  -- Két jel bejelentése:
 -- Integer típusú változó bejelentése
 -- hozzáadott tartomány határfeltételekkel:
variable cim: integer range 0 to romMeret;
```

Ha a tervező mindig az elvonatkoztatás legalsó fokán írná le a hardvert, akkor egyetlen adattípus elegendő lenne. Pl. a három-szintű logika értékei: a '0', az '1' és a 'Z'. Azonban számos leírási cél esetén a három-szintű logikai típus nem megfelelő. Pl. a hardver leíró nyelv lehetővé akarja tenni a lebegőpontos processzorra vonatkozó előírások megadását a valós számokon végzett átalakításokkal kifejezve. A VHDL sokféle szintű tervet és sokféle technológiát támogat, ezért a VHDL lehetővé teszi, hogy a tervező maga határozza meg az általa használt adattípusokat.

3.1.5. Altípusok

Egy már bejelentett ***alaptípus*** (basetype) egy részhalmaza ***altípus***ként (subtype) a következőképpen jelenthető be:

```
subtype identifier is basetype limit;
```

Szükséges annak előírása, hogy az új altípus milyen értékeket vehet fel, azaz az alap típus értékeinek egy részhalmazát. Másképpen, az alaptípus értéktartományának egy korlátozott hosszúságát, pl. egy vektorhosszúságot írnak elő. Pl.:

```
subtype myInt is integer range 0 to 3215;             -- Jó
subtype byte is std_logic_vector (7 downto 0);        -- Jó
type byte2 is array (7 downto 0) of std_logic;        -- Rendben
type byte3 is std_logic_vector (7 downto 0);          -- Hibás
subtype byte4 is array (7 downto 0) of std_logic;     -- Hibás
```

A ***rácsrend*** (array) csak új típusok bejelentésére használható, altípusokéra nem. Az std_logic_vector (7 **downto** 0) nem használható új típus bejelentésekor.

A **type** használatának a **subtype** helyett az a hátránya, hogy a bejelentett típus teljesen új típussá válik, azaz nem lehet hozzárendelni egy eredeti típusú jel egy szeletét az új adattípusú jelhez a jel átváltása nélkül. Egy rossz példa:

```
architecture rossz of pelda is
 type byte is array (7 downto 0) of std_logic;
                --Rendben, de nem tökéletes
signal a: byte;
signal c: std_logic_vector (7 downto 0);
begin
  a<=c;    -- Hibás, a és c nem azonos adattípusú
end;
```

Ha a *byte* altípusként lett volna bejelentve, a fenti példát hiba nélkül átengedte volna a VHDL fordító. Az altípusok használata azért is célszerű, mert azon függvények, amelyek egy adott típust kezelnek, használhatók az ahhoz a típushoz tartozó altípusokra is. Van két előre meghatározott altípus a VHDL-ben:

```
subtype natural is integer range 0 to megvalósításfüggő_érték;
  -- jellemzően: 2147483647
subtype positive is integer range 1 to megvalósításfüggő_érték;
  -- jellemzően: 2147483647
```

A szintézis szempontjából viszont nem lényeges, hogy egy adattípus altípusként vagy külön típusként van bejelentve.

3.1.6. Operátorok

A VHDL-ben a következő előre meghatározott ***viszonyoperátor***ok (relational operator) vannak:

Szimbólum	Operátor jelentése
=	egyenlő

Szimbólum	Operátor jelentése
/=	nem egyenlő
<	kisebb mint
>	nagyobb mint
<=	kisebb mint vagy egyenlő
>=	nagyobb mint vagy egyenlő

3.1-2. táblázat. Viszonyoperátorok

Ezek az operátorok Bool algebrai értékeket vehetnek fel, azaz értékük lehet ***igaz*** (true) vagy ***hamis*** (False). A fenti operátorok közvetlenül használhatók pl. **integer** (egész), **bit_vector** vagy **std_logic_vector** típusú értékekre. Az = és /= operátorok minden előre meghatározott vagy bejelentett adattípusra használható. A szintézis eszközök támogatják a viszonyoperátorokat.

A VHDL-ben meghatározott logikai operátorok a következők: **NOT, AND, NAND, OR, NOR, XOR, EXOR**; elnevezésük fenntartott, így ezeket a szavakat nem szabad semmilyen, felhasználó által kialakított névként alkalmazni. A következő programlistán egy példát láthatunk a használatukra.

```
architecture rtl of pelda is
signal int: std_logic;  --  Belsőjel bejelentés
begin
  int <= not (((a nand b) nor (c or d)) xor e);
  aOut <= int and f;
end;
```

A következő táblázat a VHDL-ben előre meghatározott számtani operátorokat. Ezeket a számtani operátorokat előre meghatározták az egész, a lebegőpontos (kivéve a **mod** és a **rem** operátort) és az idő adattípusokra. Ugyanakkor nincsenek meghatározva, pl. az *std_logic_vector* és az *std_ulogic_vector* típusokra.

Szimbólum	Operátor jelentése
+	összeadás
-	kivonás
*	szorzás
/	osztás
abs	abszolút érték
rem	maradék
mod	moduló
**	hatványozás

3.1-3. táblázat. Számtani operátorok

Ha az *std_logic_vector* típusra is akarjuk ezeket alkalmazni, akkor külön kell meghatározni ezeket a műveleteket. Ilyen meghatározások már rendelkezésre állnak megfelelő csomagokban.

A szintézis eszközök többsége csak a „+”, „-”, „*” és „**” műveleteket támogatják és azokat is csak az egészekre.

3.1.7. Bitfüzér szövegelemek

Vannak előre meghatározott ***bitfüzér szövegelem***ek (bit string literals), amelyeket a **bit_vector**-hoz rendelnek a következő táblázatban leírtak szerint:

Bitfüzér szövegelem magyarul	Bitfüzér szövegelem angolul	Példa
Kettesalapú	Binary	B"11000"
Nyolcasalapú	Octal	O"456"
Tizenhatosalapú	Hex	X"FFA5"
Tízesalapú	Decimal	342 (csak állandókra)
Valós	Real	5.42E-5 (szintézisnél nem támogatott)

3.1-4. táblázat. A bitfüzérek különböző alakjai

A bitfüzér szövegelemek előnye az, hogy a VHDL kód gyakran könnyebben olvasható, ha egy vektorhoz kettes helyett tizenhatosalapú értéket rendelnek. Szintézis szempontból azonban nincs különbség közöttük. Ha egy vektorhoz kettesalapú értéket rendelnek, a B-t nem kell kiírni a vektor előtt.

Az aláhúzás (_) használható a bit vektorban az olvashatóság javítása érdekében, azonban ebben az esetben a bitfüzér szövegelemeket ki kell írni.

3.1.8. Minősítők

Néha nem világos, hogy egy kifejezésnek mi az adattípusa. Ha a fordító nem tudja ezt egyértelműen eldönteni, akkor hiba keletkezik. Ennek elkerülésére a fordító számára világossá lehet tenni a típust egy ***minősítő*** (qualifier) használatával. Ennek a jelölésmódja az, hogy az adattípus neve után egy ***vonásjel*** (tick mark, „**'**”) áll, amit az a kifejezés (vagy egy érték) követ, aminek a típusát minősíteni akarjuk. Pl.:

```
ROMType'("01","10","00");
```

3.1.9. Álnevek

Az **alias** (álnév) parancs arra használható, hogy a tárgyaknak egy alternatív nevet adjon. Gyakran arra használják, hogy ezzel a kódot olvasmányosabbá tegyék. Pl.:

```
alias adat: std_logic_vector(7 downto 0) is adat_be(15 downto 8);
```

3.1.10.Kisbetűk és nagybetűk

A VHDL jelölésmódja nem tesz különbséget a nagybetűk és a kisbetűk között. Az egyetlen kivétel az egyes idézőjelek (' ') vagy a kettős idézőjelek (" ") közötti karakterek esetén

van. Pl. a 'Z' értéket hozzárendelve egy *std_logic* típusú jelhez, a 'Z'-nek nagybetűsnek kell lennie:

```
sig <= 'Z';          -- Jó
sig <= 'z';          -- Rossz
```

3.2. Tervezési egyed felépítése

A tervezési egyed az ***egyed bejelentés***ből (entity declaration) és egy ***építmény***ből (architecture) áll. Az egyed bejelentés meghatározza a tervezési egyed és a külvilág közötti határfelületet. Az építmény pedig a kapcsolatot írja le a tervezési egyed bemenetei és kimenetei között. Az építmény két részből áll, egy *bejelentési rész*ből és egy *építménytest*ből. Az építménytest az egyed egy belső képét adja: leírja az összetevő viselkedésmódját vagy a szerkezetét.

3.2.1. Az egyed bejelentés

A VHDL alapú tervek ***egyed***ekből (entity) állnak. Az egyed képviselheti a tervhierarchia egyik szintjét, vagy egy teljes tervet; lehet egy létező hardver összetevő vagy egy VHDL által meghatározott tárgy is.

Az ***egyed bejelentés*** (entity declaration) meghatározza a határfelületet a tervezési egyed és a környezet között. Az egyed bejelentés jelölésmódját mutatja a következő modell.

```
entity <identifier_name> is
port([signal] <identifier>:[<mode>] <typeIndication>;
        ...
        [signal] <identifier>:[<mode>] <typeIndication>);
end [entity] [<identifier_name>];
```

A ***port*** (kapocs) módjai a következők lehetnek: **in, out, inout, buffer, linkage.**

- **in**: az összetevő csak olvassa a jelet,
- **out**: az összetevő csak írja a jelet,
- **inout**: az összetevő olvassa vagy írja a jelet (kétirányú jel),
- **buffer**: az összetevő írja és visszaolvassa a jelet (nem kétirányú jel, a jel csak kifelé megy az összetevőből),
- **linkage**: csak dokumentációban használt.

Az egyed megadásakor meghatározott kapcsok az építményen belül jelként látszanak annyi eltéréssel az építményen belül bejelentett jelektől, hogy a kapcsok a fentiek szerint irányítottak lehetnek, míg az építményen belül bejelentett jelek mindig kétirányúak, lényegében a villamos vezetéket modellezik. A következő programlista példa egy egyed bejelentésére.

```
entity pelda is
port(signal AIn:    in  std_logic;    -- bemenet
        signal  BOut: out std_logic);  -- kimenet
end pelda;
```

Az *inout* módot csak kétirányú jelek esetén kell használni. Ha a jelet újraolvassák, akár *buffer* mód, akár belső ***látszatos*** (dummy) jel használható. A *jel* értelmű *signal* kulcsszó szokás

szerint kihagyható a *port* bejelentésből, mivel nem jelent többlet információt. Az ***in*** mód és az egyednév az **end** után szintén kihagyható. Az alábbi leírás példát mutat az egyszerűsítésre.

```
entity pelda is
port(signal a,b: in std_logic;
        signal c: out std_logic);
end pelda;

entity pelda is  -- Azonos a fenti példával
port(a,b: std_logic;
       c: out std_logic);
end;
```

Bármit, amelynek lehet értéke (pl. egy jel) egy ***tárgy***nak (object) hívunk. A VHDL-ben minden tárgynak van típusa, amely előírja, hogy milyen fajta értéket vehet fel a tárgy.

A VHDL-ben létező ***tárgy***ak (object) lehetnek ***jel***ek (signal), ***változó***k (változó) és ***állandók*** (constant). Egy változó abban különbözik egy jeltől, hogy egy jelhez hozzárendelt érték csak egy meghatározott idővel később válik érvényes értékké, míg a változóhoz rendelt érték azonnal érvényessé válik. Szemben a jelekkel, amelyek a hardverben analógok a vezetékekkel, a változóknak nincsenek közvetlen analógjaik. A változók sokkal alkalmasabbak a hardver működésmód magasabb szintű modellezésében szükséges számításokhoz.

A kifejezett tárgy bejelentéseken kívül vannak másfajta tárgyak is, amelyeket más módon kell megadni. Ezek közül a legfontosabb az ún. ***kapocs*** (port) tárgy. A kapcsokat meg lehet adni egyed bejelentésekben, összetevő bejelentésekben és ***tömb*** (block) utasításokban.

A hardver szerkezet részei: ***összetevő*** (összetevő), ***kapocs*** (port), ***jel*** (signal). A kapocs egy összetevő külvilággal való összeköttetési pontja. A jel egyaránt része a viselkedési és a szerkezeti leírásnak. A jel egy különleges változó, mely az egyes adatértékeket hordozza, ugyanakkor a jel köti össze az egyes összetevőket. Ha a jel értéket kap, akkor az új értékét csak a legközelebbi szimulációs ciklusban veszi fel, míg a változó azonnal felveszi az új értékét. A jelekhez való hozzárendelést a **<=** módon jelöljük, míg a változókhoz való hozzárendelést a sok más programozási nyelvben is alkalmazott **:=** módon jelöljük.

Az egyed bejelentésben az egyedet magát, és az egyes kapcsait azonosítónak nevezzük. A VHDL azonosító karakterek sorozata, amely a következő követelményeket elégíti ki:

- szóközt nem tartalmazhat,
- kis- és nagybetű, számjegy vagy aláhúzás szerepelhet a karakterfüzérben,
- az első karakter mindig betű,
- a füzér nem tartalmazhat aláhúzással össze nem vont karakter sorokat, valamint az utolsó karakter nem lehet aláhúzás.

Minden egyed két részből áll: az ***előírás***ból (specification) és az ***építmény***ből (architecture). Az egyed előírása a külső határfelülete, az építménye a belső megvalósítása. Egy egyednek csak egy előírása, de több építménye lehet. Amikor az egyedet egy hardver tervbe lefordítanak, egy ***kialakítás*** (configuration) írja elő azt, hogy melyik építményt kell használni.

Az egyed előírása és építménye lehet külön vagy közös VHDL forrásfájlban is.

A) Az általános (generic) utasítás

Az ***általános*** (**generic**) előírások egyed paraméterek. Az ***általánosság***ok (generics) használata ajánlott ott, ahol csak lehet a méretezhető VHDL modellek létrehozása érdekében. Az általánosságok arra használhatók, hogy információkat vigyenek be a modellbe, pl. időzítési adatokat. Az általánosságot a ***kapocs*** (port) utasítás előtt be kell jelenteni az egyedben.

Egy általánosságnak lehet alapértelmezett értéke. Az általánosság az egyeden belül állandó értékű, csak az egyed beültetésénél változhat az értéke. Az általánosságnak bármilyen adattípusa lehet, de a szintézis eszközöknek vannak megszorításaik. Példa:

```
entity genPelda is
  generic (delay: time:=10 ns);
  port (a,b: in std_logic; c: out std_logic);
end;
architecture dtf of genPelda is
begin  c <= a and b after delay;
end;
```

A fenti összetevőnek *delay* ns-nyi ***általános*** (generic) késleltetése van, ahol a késleltetés alapértelmezésben 10 ns-nak van meghatározva. Ez az érték változhat, ha az összetevőt példányosítják, azaz beültetik. Az általános jellemző használható ***általános összetevő*** (generic component) tervezéséhez is.

3.2.2. Az építmény

Két név van megjelölve az építmény leírásában: az ***összetevő*** (component) név, amely leírja, hogy az építmény hova tartozik és az *építmény egyedi neve*. Az egyed bejelentésben szereplő ***egyed azonosító*** (entity identifier) azonos az *összetevő* névvel. Az építmény jelölésmódját mutatja a következő modell.

```
architecture <architecture_name> of <entity_identifier> is
[<architecture_declarative_part>]
begin
  <architecture_statement_part>  -- Az építmény teste
end [architecture] [<architecture_name>];
```

Az építmény bejelentési része az első „**begin”** szóig tart és típus-, alprogram-, összetevő- és jel bejelentésekből áll. Pl. a következő leírás egy inverter építményét mutatja be.

```
architecture dtf of pelda is
begin
  bOut <= not aIn;
end dtf;
```

Egy NAND összetevő építményének két változatát mutatja be a következő leírás.

```
entity nandComp is
   port(a,b: in std_logic; c: out std_logic);
end;

architecture dtf1 of nandComp is
begin
```

```
  c <= not (a and b);
end;
--
architecture dtf2 of nandComp is
signal Int: std_logic;  --  Belsőjel bejelentés
begin
  Int <= a and b;
  c <= not Int;
end;
```

A *nandComp* összetevő meghívását mutatja a következő VHDL utasítás, ahol a ***példányosított*** (azaz ***beültetett***, instantiated) összetevő neve *U1*.

```
U1: nandComp port map (a => wire1, b => wire2, c => wire3);
```

A *nandComp* minden egyes meghívásakor egyedi nevének kell lennie.

3.3. Utasítások csoportosítása

Az építmény mindig ***egyidejű*** (concurrent) utasításokat tartalmaz, melyek között lehetnek pl. ***folyamat***ok, egyidejű jelhozzárendelések, egyidejű eljáráshívások és *tömb*ök. A tömbök folyamatok és egyéb egyidejű utasítások csoportosítására szolgál, a **block** utasítással hozzák létre. A tömbök csoportjaiból újabb tömbök is képezhetők a **block** utasítással.

Fontos annak az ismerete, hogy a tervezési tárgy építményében lévő különböző VHDL utasítások közül melyek egyidejűek és melyek sorrendiek. Röviden azt mondhatjuk, hogy az építményben lévő utasítások általában egyidejűek, kivéve a **process**, a **function** és a **procedure** utasításokon *belül* lévő utasításokat, mert azok ***sorrendi***ek (sequential). Ezt mutatja a következő VHDL kódváz.

```
architecture viselk of pelda is
egyidejű bejelentési rész
begin
  egyidejű VHDL
   process (...)
   sorrendi bejelentési rész
   begin
       sorrendi VHDL
   end process;
  egyidejű VHDL
end;
```

3.3.1. Egyidejű (concurrent) utasítások

Ezek az utasítások az egyidejűleg végrehajtandó hardver feladatokat írják le. Az egyidejűleg végrehajtható utasításokat a következő táblázat foglalja össze.

Egyidejű utasítások	Értelmezésük
<=	Egyidejű jelhozzárendelés, adatáramláson végzett művelet leírására szolgál

Egyidejű utasítások	Értelmezésük
After időpont	A szimulációs kezdetétől számított idő
Block (tömb) utasítás	A hardver feladatkör kisebb építőelemekre való szétbontására szolgál, egyidejű utasításokat lehet vele csoportba foglalni, több tömbutasítás egymásba is ágyazható (egyidejű utasítások tömbösítése)
Egyidejű eljáráshívás	Eljárások hívása
Generate (létrehoz) utasítás	Többször beültetés hatékony leírására szolgál
Assert (követelés) utasítás	Valamilyen követelmény megfogalmazása, amely ha nem teljesül egy üzenet jön létre
Összetevő beültetési utasítás	A bejelentett összetevők aktuális felhasználását végzi, ezt példányosításnak vagy beültetésnek is nevezzük
Procedure, function	Eljárás és függvény meghatározások
Process (folyamat) utasítás	Elvonatkoztatás, amelyen belül algoritmikusan adható meg a hardver feladatkör
Típusok és állandók bejelentése	Átfogóak lesznek, az egész tervben láthatóak és felhasználhatóak
When else (mikor még) utasítás	Adatáramláson végzet művelet leírására szolgál
With választó utasítás	Adatáramláson végzet művelet leírására szolgál

3.3-1. táblázat. Az egyidejű részben használható utasítások

Az összes egyidejű utasítás azonosítható címkével:

```
CímkeNév: b <= a;
```

Ez a címke csak dokumentációs célokra használatos és nincs feladatköri jelentősége. Más programozási nyelvekkel szemben itt nincs lehetőség a címkére ugrani.

3.3.2. Sorrendi (sequential) utasítások

A következő táblázatban megadott VHDL utasítások és műveletek végrehajtása egymásután, sorrendi módon történik egy bizonyos időlépésen belül:

Sorrendi utasítások	Értelmezés
:=	Változó-hozzárendelési (értékadás)
<=	Sorrendi jelhozzárendelés (értékadás)
After késleltetés	Az adott szimulációs időhöz képesti késleltetés, a jelenleg végzett művelet időigénye leírható vele
Függvény és eljáráshívások	
If-then-else, case, wait	Vezérlő utasítások
Követelés (assert) utasítás	Valamilyen követelmény megfogalmazása, amely ha nem teljesül, egy üzenet jön létre
Loop, next, exit	Hurokképző utasítások
Null utasítás	
Procedure, function	Eljárás és függvény meghatározások
Return utasítás	Visszatérési érték beállítás
Típusok és állandók bejelentése	Helyiek lesznek egy folyamaton belül
Variable utasítás	Változó bejelentés

3.3-2. táblázat. Sorrendi utasítások

3.4. Értékadás

3.4.1. A változó-hozzárendelés (értékadás)

A változó-értékadás egy változó tárgy értékét kicseréli egy új értékkel, amit a hozzárendelés jobb oldalának kiértékeléséből adódik. A változó-értékadás (***hozzárendelés***, assignment) jelölésmódja a következő:

```
változóNév := kifejezés;
```

A változót egy folyamatban vagy egy alprogramban lehet bejelenteni. Ha egy változót egy folyamatban jelentenek be, az végig a szimuláció során megtartja értékét, azaz sohasem állítódik ismét kezdőértékre. Az alprogramban bejelentett változók mindannyiszor újra kezdőértékre állnak, ahányszor az alprogram ismét meghívásra kerül. Ennek az oka az, hogy a folyamat folyamatosan végrehajtódik (akár tevékeny, akár felfüggesztett), míg az alprogram csak akkor kerül végrehajtásra, amikor meghívják.

Szemben a jelhozzárendeléssel, a változó-hozzárendeléshez nem társul késleltetés. A változó tárgy az új értéket **azonnal** megkapja a hozzárendelés kiértékelését követően. Ez azt jelenti, hogy a változó-hozzárendelést követő, a változót használó sorrendi utasítások már a változó új értékét fogják használni.

3.4.2. A jelértékadás

A jel forrása azt jelenti, hogy egy jel egy aktuális értékből és egy vetített kimeneti ***hullámalak***ból (waveform) áll. Az aktuális értékelem mindig a jel azon értékét mutatja, amelyet bármely folyamat olvashat. Ha egy jel nem feloldott jel, akkor csak egyetlen **process** (folyamat) utasítás van, amely értéket adhat a jelnek. Ezt a folyamatot a jel forrásának nevezik. Egy jel forrása kapcsolódhat a folyamatot tartalmazó összetevő **out**, **inout** és **buffer** módú kapcsához.

A ***jel meghajtó*** (signal driver) azt jelenti, hogy az értékeket, amelyek keresztülhaladnak egy adatúton, egy adott időpontban az adatutat meghatározó jel meghajtója tartalmazza. A meghajtó érték/idő párok gyűjteménye, amelyeket ***lebonyolítás***oknak (transaction) neveznek.

Amikor egy jel értéke megváltozik, akkor úgy mondjuk, hogy egy ***esemény*** (event) jelent meg a jelen. Amikor egy értéknek egy jelhez egy bizonyos késleltetéssel való hozzárendelése időzítésre került, azt a jel ***meghajtó***ján (driver) elkezdődött ***lebonyolítás***nak (transaction) nevezik. Az is lebonyolítás, amely nem változtatja meg a jel értékét, ezért nem okoz eseményt a jelen. Új lebonyolítás nem kezdődik el a jelen, amíg az előző nem fejeződött be.

3.4.3. A jel- és változó-értékadás összehasonlítása

A **jel**eket és a **változó**kat gyakran használják a VHDL kódban, de egészen különböző a megvalósításuk. A **változók**at sorrendi végrehajtásnál használják, hasonlóan más, algoritmikus

programhoz, míg a jeleket az egyidejű végrehajtásban alkalmazzák. A **változó-** és a **jelértékadás**t a „:=“ és a „<=“ szimbólumok használatával különböztetik meg.

Egy változót a VHDL-nek *csak a sorrendi részében lehet bejelenteni*. A jeleket a VHDL-nek *csak az egyidejű részében lehet bejelenteni*, de használhatók mind a sorrendi, mind az egyidejű részben is. Egy változó bejelenthető egy jellel teljesen azonos adattípussal. Szintén megengedett egy változónak egy jel értékét adni és fordítva, feltéve, hogy azonos az adattípusuk.

A fő különbség egy jel és egy változó között az, hogy a jelhez az értékét csak egy delta késleltetés után lehet hozzárendelni, míg a változó azonnal megkapja az értékét.

A jel és a változó feldolgozásban az a különbség, hogy a sorrendi VHDL-ben a feldolgozás sorról-sorra halad, ezért nevezik sorrendi VHDL-nek. Az egyidejű VHDL kódban a sorokat csak akkor dolgozzák fel, ha egy esemény megjelent az érzékenységi listájukon.

A jelek és változók közötti különbséget mutatja a következő listán látható két példa is.

```
-- Első példa:
-- sum1 és sum2 jelek
P0: process
begin
  wait for 10 ns;
  sum1<=sum1+1;
  sum2<=sum1+1;
end process;
--
-- Második példa:
-- sum1 és sum2 változók
P1: process
variable sum1, sum2: integer;
begin
  wait for 10 ns;
  sum1:=sum1+1;
  sum2:=sum1+1;
end process;
```

A következő táblázat bemutatja a két folyamat időzítési különbségeit.

Szimulációs idő	Sum1	Sum2	Sum1	Sum2
0	0	0	0	0
10	0	0	1	2
10 + Δ	1	1	1	2
20	1	1	2	3
20 + Δ	2	2	2	3
30	2	2	3	4
30 + Δ	3	3	3	4

3.4-1. táblázat. A folyamaton belüli jel- és változó-hozzárendelés közötti különbség.

Információ átvitele tekintetében a különbség az a jelek és a változók között, hogy a változók nem vihetnek át információt azon VHDL kódon kívülre, ahol a bejelentésük történt. Az előző példabeli *P1* folyamatbeli *Sum1* vagy *Sum2* értékére szükség van a folyamaton kívül is, akkor ezeket jelekként kell bejelenteni, vagy pedig az értéküket hozzá kell rendelni egy jelhez.

```
entity pelda is
  port(sum1Sig, sum2Sig: out integer);
end;
```

3.4.4. Vektorértékadás

A VHDL-lel való tervezésnél a vektorokat rendszerint adattípusként használják. Egy vektorhoz sokféleképpen lehet értéket hozzárendelni. Ha egy kettesalapú értéket rendelnek hozzá, ez a következőképpen történhet:

```
a_vect <= "110010";
```

Ha logikai operátorokat, mint amilyen az *and* használnak a vektorokra, az eredmény egy bitenkénti művelet lesz. A logikai operátorokra való tekintettel elvárás, hogy mind a két vektor azonos hosszúságú, és az értéket megkapó vektor hossza is megfelelő legyen.

3.4.5. Összevonás

Nagyobb vektorok esetén, ha ugyanazt az értéket akarjuk hozzárendelni az egész vektorhoz, akkor ez a következőképpen tehető:

```
architecture rtl of pelda is
  signal a: std_logic_vector(4 downto 0);
begin
  a<=(others=>'0');
end;
```

A fenti parancs azonos a következővel: a<="00000";. Az **others** előnye az, hogy kevesebbet kell írni nagyobb vektorok hozzárendelésénél és a hozzárendelés teljesen független a vektor hosszától. A vektorhoz **others**-szel való hozzárendelés ***összevont hozzárendelés***ként (aggregate assignment), röviden ***összevonás***ként (aggregate) ismert.

Egy példa: mennyi az *a* és a *b* vektor értéke a következő hozzárendelés után?

```
architecture rtl of pelda is
  signal a,b: std_logic_vector(4 downto 0);
  signal c: std_logic_vector(0 to 1);
begin
  a<=(1=>'0', 3=>'1', others=>b(2));
  b<=(1=>'1', 3=>'0', others=>c(1));
  c<="10";
end;
```

A megoldás: a="01000", b="00010".

Lehetséges hozzárendelni egy vektor néhány bitjéhez, majd a többi értékadásánál az **others**-t alkalmazni:

```
a<=(1=>'1', 3=>'1', others=>'0');
```

Szintén lehetséges összevonással hozzárendelni más jelek értékét a vektorhoz:

```
a<=(1=>c(2), 3=>c(1), others=>d(0));
```

Az *a* vektorhoz való fenti hozzárendelés másik lehetősége az ***összefűzés*** (concatenation) használata. A következő példában feltételezzük, hogy az *a* hossza 5 bit:

```
a<=d(0) & c(1) & d(0) & c(2) & d(0);
```

Ennek a leírási módszernek a hátránya, hogy a hozzárendelés hosszfüggő lett és módosítani kell, ha az *a* hossza megváltozik. Szintézis szempontjából azonban nem jelent különbséget. Mindkét leírási módszert támogatják a szintézis eszközök.

Tekintsük a következő példát, amely egy ***jegyzék*** (record) típusának bejelentését és értékadását mutatja be:

```
architecture dtf of pelda is
  type data_date is record
    month:   integer range 1 to 12;
    date:          integer range 1 to 31;
    data:          std_logic_vector(31 downto 0);
  end record;
  signal d: data_date;
begin
  d.month<=2; d.date<=14; d.data<=dataIn;
end;
```

Másképpen a fenti jegyzéknek összevonva is hozzá lehetett volna rendelni:

```
d<=(2, 14, dataIn);
```

3.4.6. Többdimenziós rácsrendek értékadása

Elvben meg lehet határozni bármilyen dimenziójú adattípust. Rendesen két- vagy háromnál nagyobb dimenziójú adattípust nem használnak. Egy többdimenziós adattípus meghatározásánál ***rácsrend***et (array) használnak a típus bejelentésben, pl.:

```
type data4x8 is array (0 to 3) of std_logic_vector(7 downto 0);
type data3x4x8 is array (0 to 2) of data4x8;
```

A *data4x8* adattípus kétdimenziós (4 x 8 bit), míg a *data3x4x8* háromdimenziós (3 x 4 x 8 bit). A *data4x8* típusbejelentése pl. a következő alakban is írható:

```
type byte is array (7 downto 0) of std_logic;
type data4x8 is array (integer range 0 to 3) of byte;
```

Egy kétdimenziós vektorhoz való hozzárendelés többféleképpen történhet. Egy rácsrendbeli elemhez hozzá lehet rendelni egy értéket az indexe használatával, vagy az egész kétdimenziós vektornak értéket lehet adni az ***összevonás*** (aggregate) alkalmazásával. A következő példa bemutat néhány hozzárendelést.

```
architecture rtl of pelda is
  type data4x8 is array(0 to 3) of std_logic_vector(7 downto 0);
  signal d,e,f,g,h,i: data4x8;
```

```
  signal b1,b2,b3,b4: std_logic_vector(7 downto 0);
begin
  d(0)<="01010110";
  d(1)<="10101000";
  d(2)<="01110110";
  d(3)<="1011011";
  e<=(others=> (others=>'0')); -- Tisztázza az egész 2 dimenziós jelet
  f(0)(0)<='1';
  f(0)(1)<='0';
   ...
  g<=(b1, b2, b3, b4);
  h<=(others=>b1);
  process(h);
  begin
    l1: for n in 0 to 3 loop
      i(n)<=h(n);
    end loop;
  end process;
end;
```

A többdimenziós hozzárendelés választása függ az alkalmazástól és attól, hogy az olvashatóság kívánalom-e. A szintézis eszközök többsége, amely elfogadja a kétdimenziós adattípusokat, az összes fenti módszert támogatja, a közülük való választásnak rendszerint nincs a szintézis szempontjából jelentősége.

3.4.7. Értékadás bitfüzér szövegelemmel

Egy vektornak való értékadás a ***bitfüzér szövegelem***mel (bit string literal) éppen olyan rugalmas, mint egy egésznek való értékadás. Mind a vektorhoz, mind az egészhez hozzá lehet rendelni bármely számrendszerbeli számot. Pl. egy egésznek értéket lehet úgy adni, hogy a számrendszer előírható egy számmal, amit közvetlenül két *kettőskeresz*t (hash mark, #) közötti érték követ, pl.:

```
architecture rtl of pelda is
  constant myInt:integer:=16#FF#;       -- myInt=255
  signal int1,int2,int3: integer range 0 to 1023;
begin
  int1<=16#FE# ;                  -- 16#FE# = 254
  int2<=2#100110# ;               -- 2#100110# = 38
  int3<=8#17# ;                   -- 8#17# = 15
  ...
end;
```

3.4.8. Rácsrend szelete (slice of array)

Egy érték hozzárendelése egy bithez vagy egy vektor egy részéhez a következőképpen történhet:

```
architecture rtl of pelda is
  signal aVect: std_logic_vector(4 downto 0);
  signal bVect: std_logic_vector(0 to 4);
 begin
```

```
  aVect(4)<='1';
  aVect(3 downto 0)<="0110";
  bVect(4)<='0';
  bVect(0 to 3)<="1001";
end;
```

Amikor egy rácsrendszelethez hozzárendelnek egy értéket, a szelet irányának olyannak kell lennie, mint ahogy az a bejelentésben volt, azaz **downto** vagy **to**.

Lehetséges egy vektorhoz egy másik vektor egy szeletét hozzárendelni:

```
architecture rtl of pelda is
  signal aVect: std_logic_vector(4 downto 0);
  signal bVect: std_logic_vector(0 to 4);
  signal c: std_logic;
begin
  aVect(4) <= "01101";
  bVect(4) <= c;
  bVect(0 to 3) <= aVect(3 downto 0);
end;
```

A vektorok adattípusának meg kell egyeznie a hozzárendelés mindkét oldalán, és a vektorok hosszának szintén egyeznie kell.

Attól még, hogy két vektor indexsorrendje különbözőképpen van bejelentve, még lehetséges az egyik vektor értékét a másikhoz hozzárendelni, de érdemes figyelni arra, hogy melyik bit melyik értéket kapja:

```
architecture rtl of pelda is
signal a,b,c: std_logic_vector(2 downto 0);
signal d: std_logic_vector(0 to 2);
begin
  a<=d;
  b<=c;
end ;
```

Az eredmény: a(2)<=d(0), a(1)<=d(1), a(0)<=d(2) és b(2)<=c(2), b(1)<=c(1), b(0)<=c(0).

3.4.9. Összefűzés

Az ***ésjel*** (ampersand, &) ***összefűzés***t (concatenation) jelent a VHDL-ben. Az összefűzés nagyon hasznos lehet a vektorértékadásban:

```
architecture rtl of pelda is
  signal a: std_logic_vector(5 downto 0);
  signal b,c,d: std_logic_vector(2 downto 0);
begin
  b<='0' & c(1) & d(2);
  a<=c & d;
end ;
```

Az összefűzés használható az **if** utasításban is, ahogy azt a következő példa bemutatja:

```
 if c & d = "001100" then
   ...
```

A következő példa az összefűzés hibás használatát mutatja be. Meg nem engedett ugyanis az értékadás bal oldalán használni az összefűzést azért, hogy az értékadás mindkét oldalán azonos hosszúságot kapjunk. A következő példa ezért hibás:

```
architecture rosszRtl of pelda is
  signal a: std_logic_vector(2 downto 0);
  signal b,c,d: std_logic_vector(3 downto 0);
begin
  '0' & a<=b;              -- Hiba!
end;
```

3.4.10.Megosztott változók

Míg a VHDL-87-ben a *változók* csak ideiglenesen tárolhatnak értékeket a folyamaton, függvényen vagy eljáráson belül, a VHDL-93 bevezette az átfogó ***megosztott változók***at (shared variable), amelyek átviszik az információt a folyamaton kívülre.

```
architecture bhv of pelda is
begin
  P1: process
  variable sum1, sum2: integer;
  begin
    wait for 10 ns;
    sum1:=sum1+1; sum2:=sum1+1;
    sum1Sig<=sum1; sum2Sig<=sum2;
  end process;
end;
```

Ahogy az alábbi lista is mutatja, a *megosztott változó*t az építmény bejelentési részében kell bejelenteni, azonban nem érhető el a VHDL kód egyidejű részében, csak a folyamatokon belül. Egy megosztott változót nem lehet belefoglalni a folyamat érzékenységi listájába sem. A megosztott változók növelhetik a bizonytalanságot, a szintézis eszközök nem támogatják az alkalmazásukat.

```
architecture bhv of pelda is
shared variable v: std_logic_vector(3 downto 0);
begin
  P0: process(a, b)
  begin
    v:=a & b;
  end process;
  P1: process(d)
  begin
    if v="0110" then c<=d;
    else c<=(others=>'0');
    end if;
  end process;
end;
```

3.4.11. Jelek és változók használatával kapcsolatos gyakori hibalehetőség

A változókat folyamatban vagy alprogramban adatok ideiglenes tárolására használják. Így változót (hacsak nem megosztott) csak a VHDL sorrendi részében lehet bejelenteni. A jelhozzárendelés lehet sorrendi és egyidejű is. A jeleket azonban csak az VHDL kód egyidejű részében lehet bejelenteni. A változók legnagyobb haszna, hogy azonnal felveszik az értéküket, míg a jelek csak egy minimum delta késleltetés múlva. Tegyük fel, hogy a következő logikai függvényt kell leírni VHDL-ben:

```
int<=a and b and c
q<=int or d;
```

Ezeket közvetlenül is be lehetne írni az építménybe egyidejű jelhozzárendeléssel, ami egy adatáramlási szintű leírás lenne, de a következő két építményben a viselkedési szintű leírásbeli használatukat vizsgáljuk:

```
entity ex1 is port(a,b,c,d: in std_logic; q: out std_logic); end;
-- Építmény jelként bejelentett belső jellel:
architecture sig of ex1 is
signal int:bit;
begin
  process(a,b,c,d,int)
  begin
     int<=a and b and c;
     q<=int and d;
  end process;
end;
-- Építmény változóként bejelentett belső jellel:
architecture var of ex1 is
begin
  process(a,b,c,d)
  variable int:bit;
  begin
    int:=a and b and c;
    q<=int and d;
  end process;
end;
```

A jelként bejelentett belső jel esetén a „`q<=int and d;`” jelhozzárendelés végrehajtásakor az *int* még a régi értékét tartja. Ezért fel kellett venni a folyamat érzékenységi listájába, így egy delta késleltetés után a ***folyamat*** (process) újra tevékennyé válik. A folyamat egy olyan kód sorozat, amely mindig végrehajtásra kerül, ha az érzékenységi listáját szereplő jelek valamelyikén változás jelentkezik. A példából is látható a változó használatának előnye, hiszen a jel esetében figyelni kell a delta késleltetésre és a folyamatnak kétszer kell lefutnia.

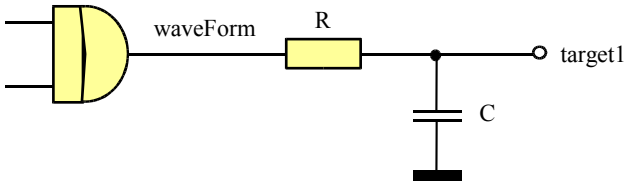
A kétféle módszer elvileg nem okoz eltérést a szintézis eredményében. Azonban a belső jel használata rejtett hibához vezethet a szintézisben, mivel a folyamat érzékenységi listáját figyelmen kívül hagyja sok szintézis eszköz. *Mindezek alapján a legjobb a változók használata egy érték ideiglenes tárolására*.

3.4.12. Tehetetlenségi és szállítási késleltetés

A jelhozzárendelés lehet ***tehetetlenségi*** (inertial) vagy ***szállítási*** (transport). Nézzük példának a következő két jelértékadást.

```
target1 <= waveForm after 5 ns;          -- tehetetlenségi
target2 <= transport waveForm after 5 ns; -- szállítási
```

Ha egy 5 ns-nál rövidebb impulzus lép fel a *waveForm* jelen, az nem fog megjelenni a *target1* jelen (ennek feltétele az is, hogy az **after** parancs is szerepeljen a késleltetésben), ugyanolyan széles impulzus a *target2*-n viszont meg fog jelenni, természetesen 5 ns-os késleltetéssel. A tehetetlenségi hozzárendelés a kapacitív hálózatok modellezésére használható. Az áramkörökben a kapu késleltetéseket a kapuk belső ellenállása és kapacitása okozza. Ezt a helyzetet szemlélteti a *3.4-1. ábra*.



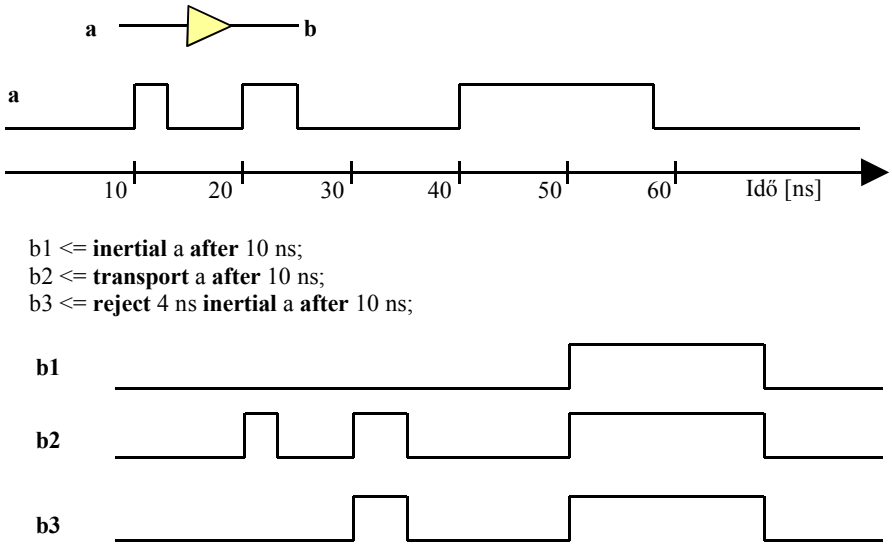
3.4-1. ábra. A tehetetlenségi késleltetés megjelenítése egy RC késleltetéssel

A *3.4-1. ábra* szerinti kapcsolásban, amely egy logikai kapura kötött integráló tagot tartalmaz, ha az integráló tag bemenetére (*waveForm*) adott értéknél rövidebb ***tüske*** (***tűimpulzus***, spike) kerül, a kimeneten ugyan megjelenik változás, de az logikai jelként nem értelmezhető. Ha azonban adott értéknél hosszabb impulzus kerül az integráló tag bemenetére (*waveForm*), akkor az a kimeneten logikai jelváltozásként is megjelenik.

Ehhez hasonló jelenség lép fel pl. a CMOS logikai áramkörökben is. Ha pl. egy *tüske* 2 ns-os és egy 10 ns-os késleltetésű CMOS áramköri összetevő bemenetére kerül, a tüske nem lesz látható a kimeneten. Azonban gondot jelent, hogy az 5 ns-os, vagy hosszabb bemeneti impulzusok meg fognak jelenni a kimeneten. Ezt a gondot a **reject** (visszautasít) paranccsal lehet megoldani. A **reject** segítségével meg lehet határozni azt a tüskeszélességet, amelyet az összetevő már átenged. Tekintsük pl. a következő VHDL kódot.

```
c <= reject 4 ns inertial b after 10 ns;
```

Ebben az esetben a szimulátor minden bemeneti tüskét mellőzni fog, amely 4 ns-nál rövidebb, az ennél hosszabb pedig 10 ns elteltével látható lesz a *c* kimeneten. A **reject** parancs használatakor kifejezetten elő kell írni a tehetetlenségi késleltetést az **inertial** paranccsal. A *3.4-2. ábra* bemutat néhány értékadást és az ezekből kapott jelalakokat.



3.4-2. ábra. Egy példa a tehetetlenségi és a szállítási késleltetésre

A vezetékek kapacitása miatt létrejövő késleltetések az elektronikus áramkörökben szintén tehetetlenségiek. Mivel a tűimpulzusokat a hosszúságuktól függetlenül továbbítja a szállítási késleltetés, ezért ez alkalmas a rezisztív összeköttetések modellezésére, míg a tehetetlenségi késleltetés az összetevők késleltetésének leírására előnyös. A VHDL-ben a tehetetlenségi hozzárendelés az alapértelmezett, ezért az **inertial** parancsot nem kell külön kiírni a forrás állományban. Egy későbbi szintézis során az időzítési feltételek figyelmen kívül maradnak.

3.4.13.Egyidejűség - két buffer példája

A hardver természeténél fogva párhuzamos, és ezért a VHDL is az. Ebből az következik, hogy az összes VHDL-beli egyidejű nyelvi szerkezet egyidejűleg elvégezhető, ami a kódbeli sorrendet érdektelenné teszi. Tekintsünk két példát, ahol az egyidejű utasítások sorrendje különbözik:

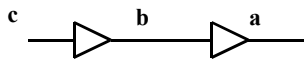
```
architecture rtl1 of signAss is
begin
  a<=b;
  b<=c;
end;
```

```
architecture rtl2 of signAss is
begin
  b<=c;
  a<=b;
end;
```

Az egyidejű jelértékadásnál a kódbeli sorrend jelentéktelen, mivel az egyidejű VHDL parancsok eseményvezéreltek. Pl.:

```
a <= b;   -- Akkor fog végrehajtásra kerülni, ha változik b értéke
```

Ez azt jelenti, hogy amikor a bemeneti *b* jel változtatja az értékét, az összes sor, amelyben *b* a hozzárendelés szimbólum jobb oldalán van, végrehajtódik. Ha a szintézist elvégezték az előző példában, akkor az eredmény (nem optimalizáltan):



3.4-3. ábra. Két szintetizált buffer

Amint a hardver mutatja, az *a* jel nem változik a *b* jel értékének változása előtt. Hasonlóan a *b* jel nem változik, mielőtt a *c* jel értéke változna. Ugyanez a helyzet az eseményvezérelt VHDL kóddal is. Ez azt jelenti, hogy az összes egyidejű VHDL utasítás bármilyen sorrendben írható anélkül, hogy a terv feladatköre változna.

3.4.14.Feloldott jelek

Lehetséges, hogy egy modell több jelhozzárendelési utasítást tartalmaz, amelyek megkísérelnek egyidejűleg különböző értékeket hozzárendelni ugyanahhoz a jelhez. Amikor ez megtörténik, a modellnek egy feloldási függvényt kell biztosítania, amely előírja, hogy hogyan oldják fel a hozzárendelést. Minden olyan jel, amelynek szüksége van feloldási függvényre a jel bejelentésben hivatkozik a megfelelő függvényre. Pl.:

```
signal total: wired_or integer;
```

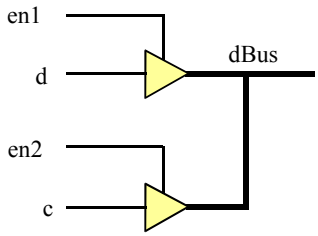
Ez a jel bejelentés egy feloldási függvényt hív, amelynek neve: *wired_or*.

Az IEEE std_logic feloldott adattípus alkalmazása

Mivel nem lehetséges két jelet egyesíteni, ha mindkettőnek nem ugyanaz az adattípusa, erre egy megoldás az IEEE által meghatározott *std_logic* és *std_ulogic* adattípusok bevezetése, melyeket az *std_logic_1164* nevű *ieee* könyvtárbeli csomagban bezettek be. Az *std_ulogic* és az *std_logic* ugyanazon értékeket vehetik fel, csak az a különbség közöttük, hogy az *std_logic*-ot a következőképpen határozzák meg.

```
subtype std_logic is resolved std_ulogic;
```

A *3.4-4. ábrán* egy adatbusz meghajtó áramköre látható, ahol két kapu kimenete ugyanazt a vezetéket hajtja meg.



3.4-4. ábra. Egy adatbusz meghajtó áramköre

A 3.4-4. ábrán bemutatott adatbusz-meghajtás VHDL modellje a következő.

```
entity pelda is
  port (d,c,en1,en2: in std_logic; dBus: out std_logic);
end;
architecture rtl of pelda is
begin
  dBus <= d when en1 = '1' else 'Z';
  dBus <= c when en2 = '1' else 'Z';
end;
```

A fenti modellben az *std_logic* áll az *std_ulogic* logikai típus helyett, mivel az *std_ulogic* típussal nem lehetne helyesen leírni ezt a modellt, mivel az *std_logic* ***feloldott*** (resolved) típus, az *std_ulogic* pedig egy ún. ***feloldatlan*** (unresolved), ahogy ezt a következő példa is mutatja:

```
-- dBus meghajtó std_ulogic-kal -  H I B Á S !
entity pelda is
  port (d,c,en1,en2: in std_ulogic; dBus: out std_ulogic);
end;
architecture rtl of pelda is
begin
  dBus <= d when en1 = '1' else 'Z';
  dBus <= c when en2 = '1' else 'Z';
end;
```

Az, hogy az *std_logic* feloldott azt jelenti, hogy ha egy jelet több meghajtó meghajt, egy előre meghatározott ***feloldási függvény*** (resolution function) kerül meghívásra, amely feloldja az ütközést és eldönti, hogy melyik értéket kapja a jel. Mindebből adódik, hogy az *std_logic* használható háromállapotú buszokhoz, amelyekben sok meghajtó meghajthatja ugyanazt a vezetéket a buszon, de rendesen nem egyidejűleg. Ha egy *std_ulogic* jelet több mint egy meghajtó hajt meg, akkor ez hibához vezet, mivel a VHDL nem engedi, hogy egy feloldatlan jelet több mint egy meghajtó hajtson meg.

Összehasonlítva a két típust, az *std_logic* előnyösebb, mert könnyebb az egész terven keresztül ugyanazt a típust használni. A hátránya azonban, hogy nem jelenik meg hiba, ha tévedésből a VHDL kódban ugyanazon jelnek két meghajtója van.

3.4.15.A kezdőérték és megváltoztatása

A nulla szimulációs időben az összes jel a ***kezdőérték***ét (initial value) kapja. Ez az érték attól függően változik, hogy milyen adattípusnak van bejelentve. Hacsak másképpen nincs előírva, az összes jel azt kapja, ami a *datatype'left* érték. A *'left* jelzőérték a bejelentésben az adattípus első (bal oldali) értékét jelenti. A *bit* és az *std_logic* típus a következőképpen van bejelentve a *standard,* ill. az *std_logic_1164* csomagban:

```
type bit is ('0', '1');
type std_logic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

Mivel bit'left = '0' és std_logic'left = 'U', ezért a *bit* kezdőértéke a '0' lesz, az *std_logic*-é a 'U'. Ha ismerni akarjuk, hogy egy adattípus mit kap kezdőértékként, tudni kell, *hogyan van bejelentve*.

Ha szükséges, a VHDL kódban a kezdőérték megváltoztatható egy új érték megadásával. Ez egy kapocsnak az egyedbejelentéssel való megadásakor vagy a jelnek az építményben történő kifejezett bejelentésekor történhet. Pl.:

```
entity pelda is
  port(a: in std_logic:='0'; b,c: in std_logic; q: out std_logic);
end;
architecture dtf of pelda is
  signal     i1: std_logic:='1';
  signal     i2: std_logic:='H';
  signal i_vect: std_logic_vector(3 downto 0):="00LL";
begin
  i1<=a or b;
  ...
end;
```

A fenti példában a *b* és *c* kezdőértéke 'U'. Ha pl. az *i1* belső jelhez 0 ns időpontban az építményben '0'-et rendelnek hozzá, az *i1* értéke delta ideig az '1' kezdőérték lesz és csak 0 ns + 1 delta időpontban fogja a '0' értéket megkapni.

Az egyedek *out* vagy *inout* módú kapcsain is lehet a jelek kezdőértékét megváltoztatni. Ha sok jel kapja ugyanazt a kezdőértéket, ez egy sorban is történhet:

```
signal a,b: std_logic_vector(2 downto 0):="001";
```

A szintézis eszközök nem támogatják a kezdőértékeket. Elfogadják a létezésüket a VHDL kódban, de egy figyelmeztetéssel mellőzik. Ezért nem lehetséges egy bemeneti jelet pl. 'H'-ra, mint kezdőértékre állítani, majd azt feltételezni, hogy a szintézis eszköz egy ***felhúzó***t (pull-up) köt a jelre. Ha felhúzót kell használni a tervben, akkor azt kifejezetten be kell építeni a VHDL leírásba. A kezdeti érték használatának az a kockázata, hogy a szintézis előtti VHDL modell és a szintetizált modell szimulációs eredménye nem lesz azonos. A hardverben ennek a bemeneti jelnek más kezdeti értéke lehet beindításkor, ami eltérésre vezethet a szintézis előtti és a szintézis utáni, igazolásul végzett kapuszintű szimuláció között.

3.5. *Alprogramok*

Az alprogramokat pl. logikai egyenletek megjelenítésére, ***típusváltás***ra (type conversion) vagy késleltetés kiszámítására használják. Az alprogramok a folyamatoktól eltérően nem tudnak közvetlenül olvasni/írni jeleket az építmény többi részéből/részébe. Minden adatközlés az alprogram határfelületén keresztül zajlik, minden egyes alprogram hívásnak saját határfelületi jelkészlete van. Kétféle alprogram van, ***függvény*** (function) és ***eljárás*** (procedure), ez utóbbi lehet egyidejű vagy sorrendi.

3.5.1. Eljárás

Az eljárás nulla vagy több, a határfelületén felsorolt értékkel tér vissza, de mellékhatásai is lehetnek. Bemenetei (**in** módú), kimeneti (**out** módú) és kétirányú (**inout** módú) paramétereket fogad el. Nem kell visszatérnie semmilyen értékkel vagy értékekkel. Nem szükséges a leírásában **return** (visszatérés) utasítás.

Az egyidejű eljáráshívás jelölésmódja a következő:

```
concurrent_procedure_call ::=
[ label : ] [ postponed ] procedure_call ;
```

A sorrendi eljáráshívás jelölésmódja az alábbi:

```
procedure_call_statement ::=
    [ label : ] procedure_call ;
```

A paraméterek átvitele nélküli eljáráshívásra mutat példát a következő sor:

```
a_proc;
```

A következő példa egy nevesített eljáráshívást mutat be *sig1*, *sig2*, *var3* átviteli paraméterekkel, amelyek a helyükkel vannak megadva:

```
lab: my_proc(sig1, sig2, sig3);
```

A **return** (visszatérés) utasítás jelölésmódja az alábbi:

```
return_statement ::=
[ label : ] return [ expression ] ;
```

A következő példában nincs visszatérési érték.

```
return;
```

A **return** egy értékkel tér vissza:

```
return value;
```

A visszatérési érték a *myFunction* függvény eredménye:

```
return myFunction(data, 5 pF);
```

A visszatérési érték az *a* + *b* + 5 ns összege:

```
return a + b + 5 ns;
```

A visszatérési érték egy láncolt füzér (az összefűzés művelettel):

```
return "szerzonev: " & nev;
```

3.5.2. Függvény

A függvény közvetlenül, egyetlen értékkel tér vissza, nincs mellékhatása. Csak bemeneti (**in** módú) paramétereket fogad el. A visszatérés csak egy érték. Mindig használja a **return** fenntartott szót. A függvényt csomagban, építményben és folyamatban lehet meghatározni, ezek közül a csomagban meghatározottat újra fel lehet máshol használni. Egy példa:

```
function f(signal a,b: bit) return bit is
begin
  if a'event then
    return b;
  else
    return a;
  end if;
end;
```

A függvény meghatározásban (ahogy a példa is mutatja) lehet több **return** utasítás is, a feltétel az, hogy csak egy hajtódik végre. A példában azért kell a „signal” jelölés, mert ki kellett zárni, hogy változók (változó) legyenek a paraméterek, amelyek nem megengedettek, pl. egyidejű függvényhívásban. A következő példában egy függvény egy csomagban való bejelentése látható:

```
package myPack is
  function max (a,b: in bit_vector) return bit_vector;
end;
package body myPack is
  function max (a,b: in bit_vector) return bit_vector is
  begin
    if a>b then
      return a;
    else
      return b;
    end if;
  end;
end;
```

A csomagbeli függvény alkalmazása:

```
use work.myPack.all;
entity pelda is
  port(...
end;
architecture viselk of pelda is
begin
  ...
  q<=max(d1,d2);                  -- Egyidejű függvényhívás
  process (data,g)
  begin
    dataOut<=max(data,g);  -- Sorrendi függvényhívás
  end process;
end;
```

Függvény bejelentése építményben:

```
architecture viselk of pelda1 is
  function max (a,b: in bit_vector) return bit_vector is
  begin
    if a>b then
      return a;
    else
      return b;
  end max;
   ...
begin
  q<=max(d1,d2);                -- Egyidejű függvényhívás
  process(data,g)
  begin
    dataOut<=max(data,g)        -- Sorrendi függvényhívás
  end process;
  ...
end;
```

Függvény bejelentése folyamatban:

```
architecture viselk of pelda2 is
begin                       -- Építménytest kezdete
  process(data,g)
    function max (a,b: in bit_vector) return bit_vector is
    begin
      if a>b then
        return a;
      else
        return b;
    end max;
  begin             -- A folyamat kezdete
    dataOut<=max(data,g);
  end process;
  ...
end;
```

A függvényeket és az eljárásokat rendszerint úgy határozzák meg, hogy a be/kimenetei paraméterek vektorhosszúságait nem adják meg. Így tetszőleges vektorhosszúságú adatokkal felhasználhatók. A függvénybejelentés paraméterezésével kapcsolatban látható egy példa az alábbiakban.

```
use work.mypack.all;
entity pelda is
  port (a,b: in bit_vector(3 downto 0);
        c,d: in bit_vector(5 downto 0);
         q1: out bit_vector(3 downto 0);
         q2: out bit_vector(5 downto 0));
end;

architecture rtl of pelda is
begin
  q1<=max(a,b);             -- A vektorhosszúság 4 bit
```

```
  q2<=max(c,d);            -- A vektorhosszúság 6 bit
end;
```

3.5.3. Eljárás és függvény összehasonlítása egy példán

A következő példában egy bájtról egészre átváltó eljárást mutatunk be. Az ezekben hívásában szereplő típust előzetesen meg kell adni a következő sorral:

```
subtype word_8 is bit_vector (7 downto 0);
```

Ezek után a típusváltó eljárás a következő:

```
procedure byteToInt(inByte: in word_8; outInt: out integer) is
  variable result: integer:=0;
begin
  for index in 0 to 7 loop
    if inByte(index)='1' then
     result:=result+2**index;
    end if;
  end loop;
  OutInt:=Result;
end byteToInt;
```

A bemutatott típusváltást a következő példában láthatjuk függvényként felírva:

```
function byteToInt(InByte: word_8) return integer is
  variable Result: integer:=0;
begin
  for Index in 7 downto 0 loop
    Result:= Result*2 + bit'pos(InByte(Index));
  end loop;
  return Result;
end byteToInt;
```

3.6. Az építmény leírása

Az építményben lévő különböző utasítások mind egyidejűek (concurrent). Az építmény modelljének legfontosabb jellemzője az elvonatkoztatási szintje. A VHDL nyelvben különböző elvonatkoztatási szinteken tudjuk leírni a digitális elektronikai rendszert, amelyeket a következőkben tekintünk át.

3.6.1. A viselkedési modell

A feladatköri (rendszer-) szinten algoritmusokat modellezünk, mely nem tartalmaz semmilyen időzítési információt. A viselkedési szinten pedig a rendszer viselkedését és átfogó időzítési tulajdonságait írjuk le, de építmény ebben még nincsen. A viselkedési modell előnye, hogy nagyon gyorsan lehet létrehozni olyan modellt, amely a be- és kimenetek közötti kapcsolatot megteremti. Viselkedési szinten szintén algoritmikus leírást ad. Az algoritmusokat egy elvonatkoztatott nyelvi elem, ún. ***folyamat***on (process) belül szokás megadni. Míg a VHDL-ben az utasítások a valódi hardvernek megfelelően általában egyidejűek, addig a folyamatban az utasítások sorrendiek, hasonlóan a hagyományos programozási nyelvekhez.

A viselkedési modell egy rendszer feladatköri leírását tartalmazza, mielőtt a rendszer egyáltalán megtervezésre került volna. A viselkedési modell alapját a *folyamat*ok alkotják, amelyek között az információt átfogó tulajdonságú ***jel***ek (signals) közvetítik.

A viselkedési leírásban az időzítés fő sajátsága az, hogy a viselkedési modell időzítése független az ***órajel***től, röviden ***órá***tól (clock) és nem szükségszerűen felel meg a megvalósított áramkörön (pl. a chipen) mérhető késleltetéseknek vagy más, valósághűbb modelleknek.

Rendszerint az időzítést a viselkedési modellben akkor vezetik be, amikor egy memóriából olvasnak, vagy abba írnak. Így az időzítés szerepe csak a memóriával való összehangolás, így ugyanazt a memória modellt tudják használni a viselkedési és a részletesebb modellekhez is.

A VHDL már akkor is használható egy rendszer ***viselkedési szintű*** (behavioural level) leírására, mielőtt a rendszert megtervezték volna, míg az ***adatáramlási*** (dataflow) modellt akkor, amikor a fő tervezési döntések már megszülettek. A viselkedési leírás segíti a tervezőket annak igazolásában, hogy megértették-e a feladatot, míg az ***adatáramlási*** leírás a terv busz- és a regiszter-szerkezetének igazolására használható.

A VHDL-ben két szint van, ahol a tervezőnek meg kell határoznia egy diszkrét rendszer viselkedését: ***sorrendi*** (sequential) szint és ***egyidejű*** (concurrent) szint. A sorrendi szint maga után vonja az egyes, a modellben használandó folyamatok programozását. Az egyidejű szint maga után vonja ezen folyamatok közötti kapcsolat meghatározását bizonyos figyelemmel a köztük lévő közlésre. A **process** (folyamat) utasítás az a szerkezet, amely hidat képez a két szint között.

A viselkedési leírás, bár nagyon elvonatkoztatott a fizikai valóságtól, egyszerűbb áramkörök esetén mégis alkalmas áramkörszintézisre. Tekintsük pl. a következő viselkedési leírást, ennek a modellnek az áramkörszintézis után egy ***nyaláboló*** (multiplexor) fog megfelelni.

```
process (sel,a,b)
begin
  if sel = '1' then
    c <= b;
  else
    c <= a;
  end if;
end process;
```

A szintézis során a változókkal óvatosan kell bánni, hogy ahol nem feltétlenül szükséges, ne kelljen regisztert is szintetizálni. Pl. az **If-Then** szerkezet rendszerint igényli az **Else** utasítást is azért, hogy teljesen meg legyen határozva. Így elkerülhető a többlet regiszter létrehozása a szintézis során.

3.6.2. Adatáramlási (RTL szintű) modell

Ez a modell az adatokat a bemenettől a kimenetig a terven keresztülhaladva tekinti. Egy műveletet ebben a modellben úgy lehet tekinteni, mint egyidejű utasításokból álló adat-átalakítások sorozatát. Az RTL- vagy adatáramlási modellben a tervet az információnak a bemenettől a kimenet felé áramló információ leírásával modellezzük.

Az RTL feladatkör a műveletekkel és a regiszterek közötti adatátviteleket okozó jelhozzárendelésekkel írható le. A ***tároló***k (latch) nem kifejezetten vannak ***bejelentve*** (declare), de következnek abból, ha a jelértékeket rákövetkező óraciklusokban is el kell érni.

Adatáramlási modell általában akkor készül a rendszerről, amikor a fő tervezési döntések már megszülettek.

3.6.3. A szerkezeti modell

A logikai (kapu-) szintű modellt ún. ***szerkezeti*** (*structural*) leírással szokás megadni, amelyben a tervet áramköri kapcsolási rajzra vagy ***hálózati listá***ra (netlist) emlékeztető módon az összetevők összeköttetéseivel szokás megadni. Tekintsük először néhány, a VHDL nyelvben szokásos fogalom elnevezését. A **hardver viselkedésmódot** a ***viselkedési*** (behavioral), más néven ***feladatköri*** (functional) modell írja le, amely adat átalakításokat és időzítést tartalmaz.

A h**ardver szerkezet** összetevőkből, kapcsokból és jelekből áll. A **kapocs** egy összetevő külvilággal való összeköttetési pontja. A **jel** (signal) egyaránt része a viselkedési és a szerkezeti leírásnak. A jel egy változó, mely az egyes adatértékeket hordozza, ugyanakkor a jel köti össze az egyes összetevőket.

A VHDL-ben egy összetevőt a ***tervezési tárggy***al (design entity) képviselnek. A tervezési egyed egyrészt a ***tervezési egyed bejelentés***ből (design entity declaration) és az ***építménytest***ből (architecture body) áll. Az egyed bejelentés az összetevő kívülről való láthatóságát biztosítja, beleértve az összetevő kapcsainak leírását. A következő példa egy félösszeadó egyed bejelentését mutatja be, amelyben két *in* és két *out* kapocs van:

```
-- A halfAdder egyed bejelentése
entity halfAdder is
  port (x, y: in bit; sum, carry : out bit);
end halfAdder;
```

A tárgybejelentésben az egyed maga és az egyes kapcsait ***azonosító***nak (identifier) nevezzük. A VHDL azonosító karakterek sorozata, amely kielégíti a következő követelményeket:

- Szóközt nem tartalmazhat;
- kis- és nagybetű, számjegy vagy aláhúzás szerepelhet a karakterfüzérben;
- az első karakter mindig betű;
- a füzér nem tartalmazhat aláhúzással össze nem vont karakter sorokat, valamint az utolsó karakter nem lehet aláhúzás.

A VHDL azonosítók nem különböztetik meg a kis- és nagybetűket. Az elnevezések esetében megkötés, hogy van néhány fenntartott szó, amely semmilyen felhasználó által kialakított névben nem szerepelhet.

Az építménytest az egyed egy belső képét nyújtja, leírja az összetevő viselkedésmódját vagy a szerkezetét. Az építménytestben van egy eljárás utasítás, amely két jelhozzárendelési utasítást és egy várakozási utasítást tartalmaz. Az első jelértékadó utasítás előírja, hogy a *Sum* nevű kapocs az *X* nevű kapocs és az *Y* kapocs értékének kizáró-vagy kapcsolata lesz egy 5 ns-os késleltetés után. Hasonlóan a *Carry* kapocs az előbbi két kapocs értékének és kapcsolatát veszi fel adott késleltetés után. A nyíl mindig az adatfolyás irányát mutatja meg. A

várakozási utasítás felfüggeszti az eljárást addig, amíg nem történik egy esemény az *X* vagy az *Y* jeleken. Ekkor a végrehajtás újra indul a folyamat elejétől. A *process* utasítás így leírja az adat-átalakítást és az időzítést, amelyek együtt alkotják a félösszeadó viselkedésmódját.

3.7. A különböző modellek összehasonlítása

A viselkedési leírás egy vagy több folyamatot használ, ahol minden egyes folyamat sorrendi jelhozzárendelő utasításokat tartalmaz. Ezzel szemben az adatáramlási leírásban nagyszámú egyidejű jelhozzárendelés található. A sorrendi és az egyidejű jelhozzárendelés között az a különbség, hogy a sorrendi jelhozzárendeléseket a szimulátor szigorúan a modellben szereplő sorrendben hajtja végre, míg az egyidejű jelhozzárendeléseknél (hasonlóan bármely más egyidejű utasításhoz) nincs meghatározva a hozzárendelések végrehajtásának sorrendje. Egy példa az adatáramlási modellre a következő.

```
architecture dtf of pelda is
begin
  q<=a3 and a2 and a1 or c0;
  c0<= not c1;
end;
```

Egy példa a viselkedési modellre az alábbi:

```
architecture bhv of pelda is
begin
  process
  begin
    if sign='0' then q0 <= q0+1;
    else q0 <= q0-1;
    end if;
  end process;
end;
```

Mindkét fenti példa szintetizálható. Azonban gyakran az így írt viselkedési kód csak szimulálható, nem szintetizálható.

Míg a viselkedési szintű leírás abban segíti a tervezőt, hogy ellenőrizze, vajon jól megértette-e a feladatot, addig az adatáramlási leírás arra használható, hogy megtervezze az adatbuszok és a tárolóegységek (regiszterek) szerkezetét. A szerkezeti, viselkedési és adatáramlási leírás közös elemeit a következő táblázat mutatja be.

Utasítás magyar neve	Utasítás angol neve
egyed bejelentés	entity declarations
építménytest	architecture bodies
függvény bejelentés	function declaration
kapcsok	ports
álnevek	aliases
csomag bejelentések	package declarations
csomagtestek	package bodies
típus bejelentések	type declarations

általánosságok	generics
sajátosságok	attributes
állandó bejelentések	constant declarations
altípus bejelentések	subtype declarations
egyidejű követelmények	assertions
jelek	signals
tömbök	blocks

3.8-1. táblázat. A különböző elvonatkoztatási szintű leírások közös elemei

A) A szerkezeti, viselkedési és adatáramlási leírás sajátos elemei

Szerkezeti

- összetevők (components)
- kialakítási előírások (configuration specifications)
- kialakítási bejelentések (configuration declarations)
- általános utasítás (generate statement)

Viselkedési

- regiszter- és buszjelek
- egyidejű hozzárendelések (concurrent assinments)
- védők (guards)
- elválasztás előírás (disconnection specification)
- eljárás bejelentés (procedure decl.)
- sorrendi és egyidejű eljáráshívások (seq. and conc. procedure calls)
- sorrendi utasítások (sequential statements)
- folyamat utasítások (process statements)
- változók (variables)
- hozzárendelések /változó és jel/ (assignments /variable and signal/)
- dinamikus elhelyezés (dynamic allocation)

Adatáramlási

- regiszter és busz jelek
- egyidejű hozzárendelések (concurrent assinments)
- védők (guards)
- elválasztás előírás (disconnection specification)

Látható, hogy az adatáramlási leírásban nincs a viselkedésihez képest új elem, viszont hiányoznak azok az utasítások, amelyekhez nem rendelhető feladatkörrel rendelkező egység.

3.8. A követelés (assert) utasítás

A ***követelmény*** (**assertion**) fogalma, s ezt megvalósító ***követelés*** (**assert**) utasítás a VHDL egy egyedi tulajdonsága, egy érdekes nyelvi szerkezet, amely lehetővé teszi egy modell feladatköreinek és időbeli feltételeinek vizsgálatát egy VHDL összetevőn belül Segítségével a tervező a VHDL modellbe bekódolhatja a modell működési határfeltételeit. A tervező egy

logikai feltételt ad, amelynek ki kell elégülnie a szimuláció során. A feltételt futás közben ellenőrzi a szimulátor, és ha nem teljesül, akkor a követelményben meghatározott, egy bizonyos ***szigorúság***ú (**severity**) üzenetet küld a szabványos kimeneti eszközre, ami rendszerint a *végződés* (terminal) képernyője vagy egy fájl. Így a követelés utasítás akkor fut le, ha egy kifejezés hamissá válik. A szimulátor által küldött üzenetben az is fel van tüntetve, hogy éppen hol tartott a szimulátor a modell végrehajtásában.

Az **assert** használatával lehetővé válik a vizsgálata jelek tiltott együttállásának, egy időbeli feltétel nem teljesülésének vagy annak, hogy vannak-e az összetevőnek be nem kötött bemenetei. A követelmény utasítás a jelölésmódja a következő:

```
assert <feltétel>
  report <üzenet>
    severity <hibaszint>;
```

Ha a követelményben nincs megadva üzenet, akkor az „*Assertion violated*” (Követelmény megsértve) a jelentés. A következő példában, ha a *signalReset* nem egyenlő 1-gyel, akkor a „*Reset aktív!*” üzenetet küldi el a szimulátor. Sem a **report**, sem a **severity** használata nem kötelező.

```
assert signalReset = '1'
  report "Reset aktív!";
```

3.8.1. A jelentés (report) utasítás

A ***jelentés*** (**report**) utasítás jelölésmódja a következő:

```
report_statement ::=
    [ label : ] report expression
        [ severity expression ] ;
```

Egy példa a jelentés használatára:

```
report "Vege a szimulacionak!"
severity failure;
```

Az előbbi követelmény nem teljesülése esetén a szimulátor **failure** (meghibásodás) állapotba kerül, ami beállítástól függ, de általában a szimuláció megállítását jelenti az éppen kiértékelés alatti időpontban. Ha a következő példában szereplő sor a *clkDiv* eljárás megfelelő helyén van, akkor ennek alapján, amikor a *clkDiv* folyamat elindul, küld erről egy üzenetet.

```
report "clkDiv folyamat elindulasa" ;
```

A **severity** a standard csomagban bejelentett **severity_level** típusú, amelynek az értékei: **Note** (jegyzet), **Warning** (figyelmeztetés), **Error** (hiba), **Failure** (meghibásodás). A VHDL szimulátorban általában Error vagy Failure esetén a szimuláció befejeződik.

3.8.2. Hibakezelés

Az üzenet és a hibaszint szokásosan sima szövegként a VHDL szimulátor utasításablakában jelenik meg. A szimulátorban be van állítva az egyes hibaszinteknek megfelelő következmény, pl. az, hogy a szimulátor megszakítja a szimulációt. Az alapértelmezett szigorúsági hibaszint az ***error***.

A követelmény az összetevők külső jeleinek vagy belső viselkedésének igazolására használható, mind időbeli, mind feladatköri igazolásra. Ahogy az **assert** lehet mind sorrendi, mind egyidejű utasítás, elvileg bárhol használható a kódban. A hibakezelési kód csak a szimulációban jut szerephez. Pl.:

```
assert in0 /= 'X' and in1 /= 'X'
  report "in nincs bekötve"
    severity error;
```

A hibakezelési kód ellenőrzi, hogy az *in0* és az *in1* be van-e kötve, vagy pedig nem meghatározott az értékük.

3.9. Elnevezési szokások és tervezési vezérelvek

Az elnevezési szokások minden tervező számára könnyen érthető elnevezéseket biztosítanak, így hibamentes átvitelt tesznek lehetővé a különböző eszközök között:

- Az elnevezésekben használható karakterek: a...z, A...Z, 0...9.
- Az elnevezésekben aláhúzás nem használandó.
- Az elnevezések hosszúsága 3 és 20 karakter közötti legyen.
- A kapcsok nevének hosszúsága 8-nál ne legyen rövidebb, és 12-nél ne legyen hosszabb.
- Az elnevezések ne kezdődjenek számmal.
- Az elnevezések első betűje legyen nagybetű, amelyet kisbetűk kövessenek ugyanazon szóban. Ha az elnevezésben több szó van összeolvasztva, minden szó első betűje legyen nagybetű.
- A foglalt VHDL szavak csupa kis és vastag betűvel írandók.
- A szabványos behúzás két vagy három karakter, tabulátort ne használjunk.
- A csoport logikailag összefüggő bejelentések és utasítások. Érdemes tömböket (block) használni a következő egyidejű utasítások csoportosítására: alprogramok, típus és altípus műveletek, jelek és állandók.
- Használjunk behúzást az egymásba ágyazottság és az alrendszerek bemutatására.
- Használjunk ***fehér szóköz***t (white space) vagy ***üres sorok***at (blank lines) a logikailag elkülönülő kódrészek elválasztására.
- Írjuk egymás fölé a hasonló szavakat és központozási jeleket.
- Használjunk megjegyzéseket a kód nem nyilvánvaló feladatköreinek leírására. A VHDL modellek készítésénél feltétlenül javasolt a megjegyzésekkel való ellátás. Ennek oka az, hogy összetettebb VHDL modellek esetén az utólagos beleszerkesztés komoly hibaforrás lehet (és nem is elegáns, hiszen a VHDL esetén a modelleket elvben mások is szeretnék használni).
- Egyfajta stílus: nagybetűk használata a fenntartott szavakra és kisbetűk a felhasználó által meghatározott azonosítókra. Ebben a könyvben a kulcsszavakra kisbetűket használunk, mert így könnyebben olvasható a szöveg, viszont vastag betűvel szedjük.
- Használjuk a választható címkéket a folyamatokra, egyidejű jelhozzárendelésekre, az egyidejű eljáráshívásokra, stb. azért, hogy magyarázó nevet adhassunk ezeknek

Ha a kimeneti listán a bemenő (gerjesztő) jelvektorok egymás alatt, kettes számrendszerben vannak ábrázolva, ez összetettebb feladat eseten áttekinthetetlen.

Ugyanakkor, lehetséges olyan ***folyamat***okat (process) is készíteni, amelyek szöveges információkat írnak ki az alapértelmezett kimenetre, vagy csak elválasztó jeleket, pl. kötőjelekből álló sort. Ezek önálló folyamatok, amelyeket lehet aktivizálni pl. az érzékenységi listájukon szereplő, egyenesen erre a célra létrehozott jelekkel, és **wait** utasítással lehet felfüggeszteni. Ezeket a próbapadot tartalmazó fájlban érdemes alkalmazni, de lehet pl. a gerjesztő jeleket tartalmazó fájlban is.

4. Könyvtárak és csomagok

4.1. Csomagok

A ***csomag*** (package) megosztott, azaz közös építőelemek tárolására szolgál, segítségével ezek újrafelhasználása egyszerűen megtörténhet. A csomag két részből áll, az egyik a ***bejelentés*** (declaration), a másik a ***test*** (body). A ***csomag bejelentés*** (package declaration) a nyilvános információkat hordozza, beleértve állandók, típusok, altípusok, alprogramok, azaz függvények és eljárások, valamint jelek bejelentéseit.

A ***csomagtest*** (package body) a magán információkat tartalmazza, beleértve a helyi típusokat és az alprogram megvalósításokat. A csomagtestben lévő alprogram megvalósítások nem láthatóak a tervezési egyedek számára.

Bár mindezen információkat be lehet jelenteni kifejezetten egy rendszeren belül minden egyes tervezési egyedben vagy építményben, általában egyszerűbb a rendszerinformációkat egy külön csomagban bejelenteni. Minden egyes tervezési egyed ezután használhatja a rendszer csomagját.

4.1.1. A könyvtár és a csomag láthatóvá tétele

A **library** (könyvtár) záradékot a később hivatkozott könyvtárak logikai nevének felsorolásával azért helyezik el a tervezési egyed után a fájlban, hogy a szimulátor megnyissa azokat. A szimulátor kialakítási állományában van egy logikairól-fizikaira való kiosztás, amely a fájlrendszer direktori szerkezetében azonosítja, hogy hol helyezkednek el fizikailag az egyes könyvtárak. A különböző VHDL tervező eszközöknek saját módszerük van ennek a kiosztásnak az elvégzésére. A **library** utasítás jelölésmódja:

```
library logical_name_list ;
```

A **use** záradékot akkor alkalmazzák, ha bizonyos csomagok tartalmára szükség van. Jelölésmódja a következő:

```
use_clause ::=
use selected name { , selected_name } ;
```

Röviden:

```
use prefix.suffix,prefix.suffix… ;
```

A *work* könyvtárbeli *myPackage* csomag összes eleme egységbe vonható a következő módon:

```
use work.myPackage.all;
```

Ha csak egy függvényt akarunk a terv számára láthatóvá tenni, pl. a *work* könyvtárbeli *myPackage* csomagbeli *myFunction* függvényt, akkor ez a következő sorral érhető el:

```
use work.myPackage.myFunction;
```

4.2. Összetevők a csomagokban

Ha a csomagokban határozzák meg az összetevőket, akkor az építményben nincs szükség összetevő bejelentésre a beültetésnél, amint ezt a következő példa bemutatja:

```
package myPack is
  function minimum (a,b:in bit_vector)
  return bit_vector;
  component c1 port(clk,resetN,dIn:in bit;
                              q1,q2:out bit);
  end component;
  component c2 port(a,b:in bit;
                       q:out bit);
  end component;
end myPack;

package body myPack is
  function minimum (a,b:in bit_vector)
    return bit_vector is
  begin
    if a<b then
      return a;
    else
      return b;
    end if;
  end minimum;
  ...
end myPack;
```

Az összetevő felhasználásához csak a csomagot kell bejelenteni a VHDL kód elején:

```
use work.myPack.all;

entity pelda is
  port(clk,resetN:in bit;
            d1,d2:out bit;
              a,b:in bit_vector(3 downto 0);
         q1,q2,q3:out bit;
               q4:out bit_vector(3 downto 0));
end;

architecture rtl of pelda is
begin
  U1: c1 port map(clk,resetN,d1,q1,q2);
  U2: c2 port map(d1,d2,q3);
  q4<=minimum(a,b);
end;
```

A fenti példában a „*use*” záradék előtt nem kell a „*library work*;” záradékot használni, mivel a VHDL szimulátor a *work* könyvtárt (hasonlóan az *std* könyvtárhoz) alapértelmezésben megnyitja.

4.3. A standard csomag

Az alapvető típusokat a *standard* csomagban határozzák meg, amelyet az *std* nevű könyvtárban szoktak tárolni. Ezt a csomagot a VHDL fordító és szimulátor beolvassa anélkül, hogy ezt külön meg kellene jelölni a VHDL leírásban.

```
package standard is
      type boolean is (false,true);
      type bit is ('0', '1');
      type character is (
            nul, soh, stx, etx, eot, enq, ack, bel,
            bs,  ht,  lf,  vt,  ff,  cr,  so,  si,
            dle, dc1, dc2, dc3, dc4, nak, syn, etb,
            can, em,  sub, esc, fsp, gsp, rsp, usp,
            ' ', '!', '"', '#', '$', '%', '&', ''',
            '(', ')', '*', '+', ',', '-', '.', '/',
            '0', '1', '2', '3', '4', '5', '6', '7',
            '8', '9', ':', ';', '<', '=', '>', '?',
            '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
            'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
            'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
            'X', 'Y', 'Z', '[', '\', ']', '^', '_',
            '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
            'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
            'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
            'x', 'y', 'z', '{', '|', '}', '~', del);
      type severity_level is (note, warning, error, failure);
      type integer is range -2147483648 to 2147483647;
      type real is range -1.0E38 to 1.0E38;
      type time is range -2147483647 to 2147483647
            units
                  fs;
                  ps = 1000 fs;
                  ns = 1000 ps;
                  us = 1000 ns;
                  ms = 1000 us;
                  sec = 1000 ms;
                  min = 60 sec;
                  hr = 60 min;
            end units;
      function now return time;
      subtype natural is integer range 0 to integer'high;
      subtype positive is integer range 1 to integer'high;
      type string is array (positive range <>) of character;
      type bit_vector is array (natural range <>) of bit;
end standard;
```

4.4. Felültöltés

A VHDL erősen típusos nyelv: ha egy függvény pl. *bit_vector* típust vár, nem lehet bemenő paraméternek pl. *std_logic_vector*-t adni. A VHDL-ben lehetséges a ***felültöltés*** (overloading), akár felsorolási szövegelemekre, akár alprogramokra (eljárásokra és

függvényekre). A felültöltés előnye, hogy segítségével a VHDL kódok könnyebben olvashatók és tömörebbek. A felsorolási szövegelemek felültöltésére példa a következő:

```
type wireColor is (green, black, red) -- felhasználói felsorolt típus
type trafficLight is (green, yellow, red, flashing) -- felültöltött
```

4.4.1. A felültöltéssel kapcsolatos hibalehetőségek

Egyes VHDL fordítóknál a következő sor hibát okoz:

```
write (L, "szervusz");
```

Egyes fordítók esetében erre a következő hibaüzenetet kapjuk: „*Subprogram "write" is ambiguous*”, azaz a „write” alprogram nem egyértelmű. Ennek az oka az, hogy abban a szöveget kiíró csomagban (pl. az *std* könyvtárban a *textIo* csomag), amelyben a *write* eljárást meghatározzuk, a *write* eljárást felültöltjük a *string* és a bit_vector típusra a következők szerint:

```
procedure write (l: inout line; value: in bit_vector;
         justified: in side:= right; field: in width:= 0);
procedure write(l: inout line; value: in string;
        justified: in side:= right; field: in width := 0);
```

Ekkor azért keletkezik hiba, mert a „szervusz” argumentumot lehetne füzérként és bit vektorként is értelmezni, a fordítónak viszont nincs joga meghatározni az argumentum típusát ameddig nem ismeri, hogy melyik függvényt hívtuk. A következő eljáráshívás szintén hibát eredményez:

```
write (L, "010101");
```

Ez a hívás még kevésbé egyértelmű, mert a fordító még akkor sem tudná meghatározni, hogy a "010101" argumentumot bit vektorként vagy füzérként értelmezze, ha ez egyébként jogában állna. Két lehetséges megoldás van erre a gondra:

Az első lehetőség, hogy típusleíró ***minősített kifejezés***t (qualified expression) kell használni, mint pl.:

```
write (L, string'("szervusz"));
```

A második megoldás az, hogy nem ***felültöltött*** (overloaded) eljárást kell hívni, pl.:

```
write_string (L, "szervusz");
```

A *write_string* eljárás az értéket egyszerűen *string*-ként határozza meg és hívja a *write* eljárást, így ez egy héjként szolgál a *write* eljárás körül, így megoldja a felültöltési problémát. A *write_string* eljárás nem minden szövegkiíratással foglalkozó csomagban található, ilyenkor az első megoldást kell alkalmazni.

4.4.2. Az std_logic_vector kétféle értelmezése

Az ***ieee*** könyvtár két csomagot tartalmaz, amelyben az *std_logic_vector* felülírt adattípusa: az ***std_logic_unsigned*** és az ***std_logic_signed***. Attól függően, hogy a kettő közül melyik csomagot bejelentik a VHDL kódban, az *std_logic_vector* típusú adatokat előjel nélküliként vagy előjelesként értelmezik. Példák:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
...
architecture rtl of pelda is
begin
  q<=a+b;   -- előjel nélküli összeadás
end;

Library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

architecture rtl of pelda is
begin
  q<=a+b;   -- előjeles összeadás
end;
```

Előjeles és előjel nélküli vektorok ugyanazon építményben, pl.:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
...
architecture rtl of pelda is
begin
  q1<=unsigned(a) + unsigned (b);   -- előjel nélküli összeadás
  q2<=signed(a) + signed (b);   -- előjeles összeadás
end;
```

4.4.3. Alprogramok felültöltése

Egy alprogramot a hívási kód alapján lehet felültölteni. Ehhez ugyanazon alprogramnak többszörös példányait kell bejelenteni. A fordító megvizsgálja az alprogramnak küldött argumentumokat, és ez után eldönti, hogy melyik példányt kell használnia. Ha egy alprogramot nem tud feloldani, akkor hiba keletkezik.

Az *ieee* könyvtárban található felültöltött függvények: „+”, „-”, „*”, „=“, „<“, „>“, „/=“, „<=“, „>=“. Ezek bemenő ill. kimenő adattípusaik az ***std_logic_unsigned*** csomagban: *std_logic*, *std_logic_vector* és *integer*. Példák ezekre:

```
function "=" (L:std_logic_vector; R:integer) return boolean;
function ">" (L:std_logic_vector; R:integer) return boolean;
```

Példa a felültöltött „=“ használatára:

```
architecture viselk of pelda is
  signal a,b,c: std_logic_vector(15 downto 0);
begin
  process(a,b,c,d1,d2,d3)
  begin
    if a=12 or c=11 then q<=d1; -- bal oldalon egy std_logic_vector,
                                -- jobb oldalon egy egész van
```

```
    else if b>5 then q<=d2;
    else q<=d3;
    end if;
  end process;
end;
```

4.5. Típusváltás csomagbeli váltófüggvényekkel

A VHDL-ben nem lehetséges különböző adattípusú jeleket egymáshoz rendelni (pl. hierarchikus tervben). Ennek elkerülésére a tervekben rendszerint végig azonos adattípust használnak. A leggyakoribb műveletek felültölthetők. Ilyen pl. az „=”, amely felhasználható pl. egy *std_logic_vector* és egy *integer* összehasonlítására típusváltás nélkül. Sok esetben azonban szükség van típusváltásra. Ezek különböző beépített függvényeket használnak az alkalmazott könyvtár-csomagtól függően.

Az ***std_logic_1164*** csomag pl. a következő táblázatban bemutatott adattípusok közötti váltásokat támogatja:

std_logic	⇔	**bit**
std_logic_vector	⇔	**bit_vector**
std_ulogic	⇔	**bit**
std_ulogic_vector	⇔	**bit_vector**

4.5-1. táblázat. Az *std_logic_1164* csomagbeli típusváltások

Példa a *to_std_logic_vector* függvény bejelentésre:

```
function to_std_logic_vector(s: bit_vector) return std_logic_vector;
```

Példa a fenti függvény használatára:

```
library ieee;
use ieee.std_logic_1164.all;
entity pelda is
  port (a,b: in   bit_vector(3 downto 0);
           q: out std_logic_vector(3 downto 0));
end;

architecture rtl of pelda is
begin
  q<=to_stdlogicvector(a and b);
end;
```

Az *std_logic_vector* és az *integer* közti váltásra használható váltófüggvények bejelentése:

```
function conv_integer(arg: std_logic_vector)
  return integer;
function conv_std_logic_vector(arg: integer; size: integer)
  return std_logic_vector;
```

A következő példa bemutatja a használatukat:

```
entity pelda is
  port (a,b,c: in integer range 0 to 15;
        q: out std_logic_vector(3 downto 0));
end;

architecture rtl of pelda is
begin
  q<=conv_std_logic_vector(a,4) when convInteger(c) = 8 else
         conv_std_logic_vector(b,4);
end;
```

Ha a bemeneti és a kimeneti adatok is egyneműek, akkor nem lenne szükség típusváltásra. Előfordulhat azonban, hogy a típusváltás a kód olvashatóbbá tételéhez szükséges, ilyen eset látható a következő példában:

```
entity pelda is
  port (a,b,c:  in std_logic_vector(3 downto 0);
            q: out std_logic_vector(3 downto 0));
end;

architecture rtl of pelda is
begin
  q<=a when convInteger(c) = 8 else b;
end;
```

Ha a szintézis eszköz támogatja az *std_logic_vector*-ra és az egészre vonatkozó „=” függvény felülírását, akkor még olvashatóbb kódot lehet létrehozni:

```
entity pelda is
  port (a,b,c:  in std_logic_vector(3 downto 0);
            q: out std_logic_vector(3 downto 0));
end;
architecture rtl of pelda is
begin
  q<=a when c = 8 else b;
end;
```

A szintézis szempontjából nem jelent különbséget a váltófüggvény használata, mert nem használ fel egyetlen kaput sem. Az alkalmazásának csak kényelmi okai vannak a kód írásánál és megértésénél.

4.6. Egy mintacsomag és alkalmazása

Tekintsük a következő csomagot, amely két függvényt tartalmaz: *average* és *sum*. Az *average* függvény visszatér két szám átlagával (lefelé kerekítve). A *sum* függvény a két szám összegével tér vissza. A két függvény mind *integer*-re, mind *std_logic_vector* adattípusra bejelentett, ahogy azt a következő VHDL leírás bemutatja.

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

```
package myPackage is
  function average1(a,b: in integer) return integer;
  function average2(a,b: in std_logic_vector) return std_logic_vector;
  function sum1(a,b: in integer) return integer;
  function sum2(a,b: in std_logic_vector) return std_logic_vector;
end;

package body of myPackage is
  function average1(a,b: in integer) return integer is
  begin
    return (a+b)/2;
  end;

  function average2(a,b: in std_logic_vector)
  return std_logic_vector is
    variable int: std_logic_vector(a'range);
  begin
    int:=a+b;
    return  shr(int,"1");
  end;

  function sum1(a,b: in integer) return integer is
  begin
    return (a+b);
  end;

  function sum2(a,b: in std_logic_vector) return std_logic_vector is
  begin
    return (a+b);
  end;
end;
```

A következő, *c1* tervezési egyed az előző csomagbeli függvényeket (*average* és *sum*) használja. Az egyednek 4 bemenete van, *a*, *b*, *c*, *d*. Az *a* és *b* bemenetek *integer(0 to 127)* típusúak, a *c* és *d* bemeneti jelek típusa pedig std_logic_vector(7 downto 0). Az összetevő kimenetei: *average1*, *average2*, *sum1*, *sum2*, az egyes függvényeknek megfelelően. Az *average1* és a *sum1* integer(0 to 127), az *average2* és *sum2* pedig *std_logic_vector(7 downto 0)* típusúak, ahogy azt a következő leírás bemutatja.

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

entity c1 is
  port (a,b: in integer range 0 to 127;
        c,d: in std_logic_vector(7 downto 0);
      q1,q2: out integer range 0 to 127;
      q3,q4: out std_logic_vector(7 downto 0));
end;

architecture rtl of c1 is
begin
  q1 <= average1(a,b);
```

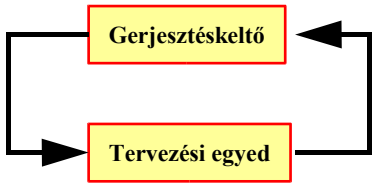
```
  q2 <= sum1(a,b);
  q3 <= average2(c,d);
  q4 <= sum2(c,d);
end;
```

5. Igazolás

Az ***igazolás*** (verification) célja a modell jóságának ellenőrzése. A szimulátor által létrehozott bemeneti gerjesztő jelek használatának hátránya az, hogy szimulátortól függő. A fejlettebb változata a *rendszer szimuláció*, amely a környezet modellezésével hozza létre a gerjesztést. A másik megoldás egy vizsgálati összeállítás (*próbapad*) VHDL modelljének létrehozása, amely bemeneti jeleket nyújt az áramköri modellhez. Ez egyben ellenőrzi az áramkör kimeneti jeleit is. Előnye, hogy platformfüggetlen, hátránya pedig az, hogy ebben a próbapad VHDL modelljében is lehetnek hibák. A megoldást mindkét módszer alkalmazása jelenti.

5.1. A gerjesztéskeltő és a próbapad

A *vizsgáló áramköri összeállítás*, azaz egy ***próbapad*** (test bench) célja az áramkör működésének a valóságnak megfelelő modellezése. A próbapad a vizsgálandó áramkörből, az áramköri modellhez a bemeneti jeleket, azaz a gerjesztést létrehozó modellből, röviden ***gerjesztéskeltő***ből (stimulus generator) és az ezeket összekötő vezetékek modelljeiből áll, ahogy azt a következő ábra bemutatja.



5.1-1. ábra. Egy próbapad tömbvázlata

Egy egyszerű példa a gerjesztéskeltőre:

```
inModel <= '1' after 1 ns,
           '0' after 10 ns,
           '1' after 20 ns,
           '0' after 30 ns + 2 ns;
clk <= not clk after 5 ns;
```

Az *inModel* jel egy négyszögjelet nyújt, míg a *clk* 10 ns-os periódusidővel rezeg.

5.2. Felhúzás/lehúzás

Ha egy tervezés alatti összetevő bemeneti jelét fel- vagy le kell húzni egy ***bemeneti/kimeneti*** (**I**nput/**O**utput, **I/O**) cellában, a ***felhúzás*** (pull up) és a ***lehúzás*** (pull down)

két módon írható le, az összetevőben és a próbapadban. Vegyük a következő összetevőt, amelynek kétirányú *io* kapcsát fel kell húzni:

```
library ieee;
use ieee.std_logic_1164.all;
entity pelda is port(a,b: in std_logic;
                     io: inout std_logic; q: out std_logic);
end;
architecture dtf of pelda is
begin
  q<=a and io;
  io<='1' when a='1' and b='0' else 'Z';
end;
```

A felhúzás modellezésének módja a kétirányú jelhez 'H' (gyenge '1') érték adás. A jelnek *std_logic* típusúnak kell lennie, amely *feloldott* (resolved) adattípus. Ha az *io* jelet folyamatosan 'H'-val meghajtják és néha más értékekkel ('0' vagy '1') egyszerre, a feloldási függvény meghívódik. Ha a kétirányú jelet a próbapad vagy belsőleg egy áramkör meghajtja pl. '0'-ba, a feloldási függvény a 'H' és a '0' bemeneti argumentumokra '0'-t fog adni. Ez megfelel a valóságnak is. Ha a felhúzás modelljét hozzáadjuk az összetevőhöz, akkor a következő építményt kapjuk:

```
architecture dtf of pelda is
begin
  io<='H';
  q<=a and io;
  io<='1' when a='1' and b='0' else 'Z';
end;
```

Ha csak kezdőértékre akarjuk állítani az *io* jelet az összetevőben, akkor a következő egyedet kell létrehozni:

```
library ieee;
use ieee.std_logic_1164.all;
entity pelda is
  port(a,b: in std_logic;
       io: inout std_logic:='H';
        q: out std_logic);
end;
```

Amint az *io* jelhez egy értéket adnak, a kezdeti érték „leválik” a jelről. Fel/lehúzás a próbapaddal:

```
library ieee;
use ieee.std_logic_1164.all;
entity probapad is port(io: inout std_logic; q: out std_logic); end;
architecture dtf of probapad is
begin
  io<='H';    -- Felhúzás
  a<='0', '1' after 100 ns, '0' after 50 ns;
  b<='0', '1' after 150 ns, '0' after 50 ns;
  io<='0', 'Z' after 100 ns, '0' after 50 ns; --io belsőleg meghajtott
end;
```

5.3. Egy egyszerű áramkör modellezése tervezőrendszertől független kiíratással

Az alábbiakban egy egyszerű áramkör különböző modelljeit és egy olyan hozzá tartozó próbapad kerül bemutatásra, amely szimulátortól, ill. tervezőrendszertől függetlenül biztosítja a szimulációs eredmények kijelzését a szabványos kimenetre. A tervezőrendszertől független tervezési stílus előnye, hogy a VHDL modellek, a hozzájuk készített gerjesztéskeltők, valamint a próbapadok csereszabatosak, azaz bármely tervezési környezetben használhatók.

A VHDL-ben egy összetevőt egy tervezési tárggyal, azaz egy tervezési egyeddel képviselnek. A tervezési egyed az egyed bejelentésből (tárgybejelentésből) és az építményből áll. Az egyed bejelentés az összetevő kívülről való láthatóságát biztosítja, beleértve az összetevő kapcsainak leírását. A következő VHDL leírás egy ***nem-és*** (nand) kapu egyed bejelentését mutatja be, amelyben két bemeneti kapcsot és egy kimeneti kapocs van:

```
-- A nandKapu áramkör tárgybejelentése
entity nandKapu is
  port (a,b: in bit; y: out bit);
end nandKapu;
```

A *nandKapu* építménye látható a következő leírásban.

```
-- A nandKapu nevű tervezési tárgyhoz tartozó viselk nevű építmény
architecture viselk of nandKapu is
-- Az építmény bejelentési része (üres)
begin
  process
  begin
    y <= not (a and b) after 1 ns;
    wait on a, b;
  end process;
end viselk;
```

Az építménytestben van egy eljárás utasítás, amely két jelhozzárendelési utasítást és egy várakozási utasítást tartalmaz. Az első jelhozzárendelési utasítás előírja, hogy az *y* nevű kapocs az *a* nevű kapocs és a *b* kapocs értékének kizáró-vagy kapcsolata lesz egy 1 ns-os késleltetés után. A nyíl mindig az adatáramlás irányát mutatja meg. A várakozási utasítás felfüggeszti az eljárást addig, amíg nem történik egy esemény az *a* vagy a *b* jeleken. Ekkor a végrehajtás újra indul a **process** utasítás elejétől. A **process** utasítás így leírja az adat-átalakítást és az időzítést, amelyek együtt alkotják a *nandKapu* viselkedésmódját.

A viselkedési leírás a valóságos működéstől leginkább elvonatkoztatottabb, lényegében a viselkedés matematikai algoritmusát írja le. A benne lévő időzítés csak a szimuláció könnyítése érdekében van, a valóságos működéshez nincs köze. A következő leírás a nandKapu adatáramlási szintű modelljét tartalmazza.

```
-- A nandKapu nevű tervezési tárgyhoz tartozó másik építmény,
-- melynek neve: adataramlas
architecture adataramlas of nandKapu is
  -- Az építmény bejelentési része (üres)
begin
  y <= not (a and b) after 1 ns;
```

```
end adataramlas;
```

Az adatáramlási elvonatkoztatási szintű leírásban nincsen **process** utasítás, az adatutakat közvetlenül írjuk le.

A harmadik elvonatkoztatási szint a szerkezeti leírás, melyben könyvtári cellákból épül fel az áramkör. A szerkezeti leírás valójában az áramköri kapcsolási rajzot helyettesíti. A **nand** esetében látszólag erre nincs szükség, mert a **not** és az **and** függvényeket a VHDL nyelv tartalmazza, de egy adott technológiára tervezés során szükségünk van az adott technológiára jellemző késleltetési, stb. paraméterekre, így jobb a könyvtári elemekből építkezni. Az *orakapuk* nevű csomag tartalmazza a szükséges elemeket. Az *oraKapuk* csomag, az elemek egyed bejelentése és viselkedési leírású építménye megtalálhatók a *Függelék*ben. Felhasználásukkal készíthetünk egy harmadik fajta építményt a *nandKapu* tervezési egyedhez:

```
-- A nandKapu nevű tervezési tárgyhoz tartozó harmadik építmény,
-- melynek neve: szerk
-- Könyvtár bejelentésére nincs szükség, mivel a work könyvtár
-- anélkül is felhasználható
-- Az use utasítással és az all kulcsszóval az oraKapuk csomag
-- minden eleme láthatóvá (így felhasználhatóvá) válik
-- az adott terv számára.
use work.oraKapuk.all;
architecture szerk of nandKapu is  -- A nandKapu szerkezeti leírása
  -- Egy, csak belül használt vezeték bejelentése (itt a signal
  -- második jelentése jön elő, ahol fizikai összeköttetést és nem
  -- elvonatkoztatott jelet jelent).
  signal belso: bit;
  -- Az építménytest
begin -- összetevő utasítások (beültetések, kapcsolódási lista)
  x1: xor2 port map(sum, x, y);
  a1: and2 port map(belso, a, b);
  a2: inv port map(y, belso);
end szerk;
```

A VHDL-ben a szerkezetet a jelek megadásával és az alösszetevők kapcsaihoz való hozzákapcsolásával írják le. A *belso* jel kapcsolja az *and2* kapu kimenetét az *inv* bemenetére. Ezen összeköttetéseket a két összetevő beültetési utasítás határozza meg. Az egyes összetevő beültetési utasítások egy-egy összetevőre vonatkoznak (*a1*, *a2*), amelyeket előzőleg a helyi összetevő bejelentésben megadtunk.

Minden összetevő beültetési utasítás címkével van ellátva, ami egy azonosító (*a1* vagy *a2*) és egy kettőspont. Ezután megnevezünk egy összetevőt (*inv*, vagy *and2*), amelyet a helyi összetevő bejelentésben adtunk meg; majd következik egy összekapcsolási lista (a port map után következő zárójelben álló lista), amely előírja, hogy melyik **aktuális** kapcsolódik melyik helyihez (ezek sorrendben a helyi összetevő bejelentésben szereplő kapcsok). Az **aktuális** lehet jel vagy kapocs is.

```
-- A nandKapu áramkör vizsgálatához készített gerjesztéskeltő
-- áramkör (neve: nandGerj) targybejelentése
-- A könyvtárak felsorolása
entity nandGerj is port (a,b: out bit; y:in bit);
end;
--A nandGerj nevű áramkörhöz tartozó adataramlas nevű építmény
```

```
-- A könyvtárak felsorolása
architecture adataramlas of nandGerj is
  -- Az építmény bejelentési része (üres)
begin  -- Az építménytest
  a <= '0' after 0 ns, '1' after 80 ns;
  b <= '0' after 0 ns, '1' after 40 ns,
       '0' after 80 ns, '1' after 120 ns;
end adataramlas;
```

A következő VHDL modell a *nandKapu* áramkörből és a *nandGerj* nevű gerjesztésből álló vizsgáló elrendezést írja le. Esetében a *nandKapu* három rendelkezésre álló építmény-modelljéből a *viselk* nevűt használtuk, de a másik kettővel is feladatkörileg ugyanazon eredményeket kell kapnunk (késleltetési időkben lehet eltérés).

```
-- A nandKapu-ból és a nandGerj-ből összeépített teljes próbapad
-- (test bench) targymegadása, neve: nandBnc
entity nandBnc is end nandBnc;

-- A nandBnc nevű tárgyhoz tartozó szerk nevű építmény megadása
-- A könyvtárak felsorolása
use std.textio.all;
architecture szerk of nandBnc is
  -- Az építmény bejelentési része
  component nandKapu port (a,b:in bit; y:out bit); end component;
  for all: nandKapu use entity work.nandKapu(viselk);
  component nandGerj port (a,b:out bit; y:in bit); end component;
  for all: nandGerj use entity work.nandGerj(adataramlas);
  signal va,vb,vy: bit; -- a „va” az „a” kapocshoz kötött
  --                       belső vezetéket jelenti.
begin  -- Az építménytest
  Aramkor: nandKapu port map(va,vb,vy);
  Gerjesztes: nandGerj port map(va,vb,vy);

-- A következő két kiíratő folyamat tervezőrendszertől független
-- megjelenítést nyújt
  Fejlec: process  -- Eredmény fejlécének kijelzése
    variable sor: line;
  begin
    write (sor, " TIME va vb vy");
    writeline (output, Sor);
    wait;  -- Ez a folyamat örökre várakozik
  end process;
  -- Eredmények kijelzése
  -- A folyamat akkor aktivizálódik, ha az érzékenységi listáján
  -- szereplő va, vb, vy jelek valamelyikén értékváltozás jelenik meg.
  AdatFigyeles: process (va, vb, vy)
    variable adatSor: line;
  begin
    -- A now változó értéke a mindenkori szimulációs idő.
    write (adatSor, now, right, 7);
    write (adatSor, va, right, 7);
    write (adatSor, vb, right, 7);
    write (adatSor, vy, right, 7);
```

```
    writeline (output, adatSor);
  end process;
end szerk;
```

Az egyed bejelentést és az építményt tartalmazó fájlt gyakran összevonják.

6. Egyidejű modellezés

6.1. Összetevő beültetés

A szerkezeti modell a rendszer hierarchikus felépítését az egyes részrendszerek kapcsolódási szerkezetét írja le. A részrendszerek közötti kapcsolatot az egyes részrendszerek ***kapcsait*** (ports) összekötő ***jelek*** (signals) írják le. Ebben az esetben a jelek valódi vezetékeket modelleznek, ezért mindig kétirányúak.

6.1.1. Összetevő bejelentés

Ahhoz, hogy a VHDL összetevőt be lehessen ültetni, először be kell jelenteni. Az összetevő bejelentés az építmény bejelentési részében történik. Jelölésmód:

```
component <entity_name>
          [generic(<generic-association-list>);]
                port(<port-association-list>);
          end component;
```

Az összetevő bejelentésbeli kapocslistának azonosnak kell lenni az *összetevő egyed bejelentés*ében szereplővel. Az ***általános*** (generic) paramétereket nem kell bejelenteni az összetevő bejelentésben.

A) Összetevő beültetés

A VHDL-ben egy egyed építményének szerkezetét a vezetékeket modellező jelek bejelentésével és ezeknek a jeleknek az egyedet alkotó összetevők kapcsaihoz való hozzákapcsolásával írják le. Ezen összeköttetéseket az összetevőkre vonatkozó ***példányosítási***, más néven ***beültetési*** (instantiation) utasítás határozza meg. Az egyes összetevő beültetési utasítás egy-egy összetevőre vonatkozik, amelyeket előzőleg egy helyi összetevő bejelentésben megadtak.

Minden összetevő beültetési utasítás címkével van ellátva, ami egy azonosító (pl.: U0, U1, U2) és egy kettőspont. Ezután megnevezünk egy összetevőt (pl.: halfAdder, orGate), amelyeket egy helyi összetevő bejelentésben adtunk meg; majd következik egy összekapcsolási lista (a *port map* után következő zárójelben álló lista), amely előírja, hogy melyik aktuális kapcsolódik melyik helyihez (ezek sorrendben a helyi összetevő bejelentésben szereplő kapcsok). Az aktuális lehet jel vagy kapocs is.

```
component_instantiation_statement ::=
instantiation _label :
    instantiated_unit
        [ generic_map_aspect ]
        [ port_map_aspect ] ;
```

Példa:

```
myEnt: entity
   work.myEntity(bhv)
   port map (I1 => S1 , I2 => S2) ;
```

A *myEntity* egyedet közvetlenül beültették. A *myEntity* I1 és I2 jeleit az S1 és S2 helyi jelekhez kötötték be. A következő példában egy *nandComp* összetevő egyed bejelentése és adatáramlási szintű leírása látható:

```
entity nandComp is
  port(a,b: in std_logic; c: out std_logic);
end;
architecture dtf of nandComp is
  signal int: std_logic;  --  Belsőjel bejelentés
begin
  int <= a and b; c <= not int;
end;
```

Az összetevő beültetés:

```
U1: nandComp port map (a => wire1, b => wire2, c => wire3);
```

A *nandComp* összetevőt beültetik, azaz a könyvtári elemet példányosítják. A konkrét példány neve: *U1*. A *nandComp* könyvtári elem minden egyes új példányának egyedi nevet kell adni.

6.1.2. Bekötetlen kimenetek

Előfordulhat beültetésnél, hogy az összetevő nem minden kimeneti kapcsa kerül kiosztásra. Ezt a helyzetet kezeli az ***open*** VHDL kulcsszó. Tegyük fel, hogy az *ex4* összetevő *q2* kimenete bekötetlen a beültetésnél:

```
architecture rtl of topLevel is
  component ex4
    port (a,b: in std_logic; q1,q2: out std_logic);
  end component;
  for U1: ex4 use entity work.ex4(rtl);
begin
  U1: ex4 port map(a=>a, b=>b, q1=>dOut, q2=>open);
end;
```

A **port map** utasításból el is hagyható a bekötetlen kimenet, ilyenkor a következő alakban írható:

```
U1: ex4 port map(a=>a, b=>b, q1=>dOut);
```

A rövidített **port map** felírási módnál a bekötetlen kimenet csak akkor hagyható el, ha az az utolsó; ez a rövidített felírás egy hátránya.

6.1.3. Bekötetlen bemenetek

A bemeneti kapocs nem ***lebeg***het (float) beültetés után. Ha egy összetevő bemenete nem használt, akkor a jelnek vagy VCC-re vagy GND-re kell kapcsolódnia. A VHDL-87 szabványban nem lehetett közvetlenül kiosztani ezekre a kapcsokat a ***port map*** parancsban,

hanem egy belső jelet kellett használni. Ezt a belső jelet hozzá kell rendelni a '0'-hoz vagy az '1'-hez, s a nem használt bementet erre a belső jelre kellett kiosztani:

```
architecture rtl of topLevel is
  component ex1 port (a,b: in std_logic; q1,q2: out std_logic);
  end component;
  for U1: ex1 use entity work.ex4(rtl);
  signal gnd: std_logic;
begin
  gnd<='0';
  U1: ex1 port map(a=>gnd, b=>b, q1=>dout, q2=>d2);
end;
```

Egy hibás példa:

```
architecture rossz of topLevel is
  component ex1
    port (a,b: in std_logic; q1,q2: out std_logic);
  end component;
  for U1: ex1 use entity work.ex4(rtl);
begin
  U1: ex1 port map(a=>open, b=>b, q1=>dout, q2=>d2); -- Ez a sor hibás
end;
```

Megengedett viszont a kapcsok közvetlen kiosztása '0'-ra vagy '1'-re:

```
architecture jo of topLevel is
  component ex1
    port (a,b: in std_logic; q1,q2: out std_logic);
  end component;
  for U1: ex1 use entity work.ex4(rtl);
begin
  U1: ex1 port map(a=>'0', b=>b, q1=>dout, q2=>d2);
end;
```

6.1.4. Általános kiosztás (generic map) utasítás

Ha általános összetevőt írnak elő a beültetendő összetevőben, akkor az értékei a **generic map** paranccsal módosíthatók a beültetés során. Jelölésmódja:

```
generic map(<generic_értékek>);
```

Az általánosságok használatával paraméterezhető összetevők hozhatók létre. A következő példa általánosságokat használ mind az összetevő késleltetésre, mind a bemenetek számára. Ha az összetevőt beültetik, az összetevő késleltetését és a bemenetek számát meghatározzák:

```
entity andComp is
  generic (tdelay:time:=10 ns; n:positive:=2);
  port (a: in bit_vector(n-1 downto 0); c: out bit);
end;
architecture bhv of andComp is
begin
  P0: process(a)
    variable int:bit;
```

```
  begin
    int:='1';
    for i in a'length-1 downto 0 loop
      if a(i)='0' then
        int:='0';
      end if;
    end loop;
    c<=int after tdelay;
  end process;
end;
```

Az előző példában meghatározott összetevő felhasználásával, ha egy tervben egy három-bemenetű *andComp* összetevő szükséges 12 ns késleltetéssel és egy két-bemenetű 8 ns késleltetéssel, akkor a jelölésmód a következő:

```
entity pelda is port (d1,d2,d3,d4,d5: in bit; q1,q2: out bit); end;

architecture str of pelda is
  component andComp generic(tdelay:time; n:positive);
    port (a: in bit_vector(n-1 downto 0); c: out bit);
  end component;
  for U1,U2: andComp use entity work.and_comp(bhv)
begin
  U1: andComp generic map(n=>2, tdelay=>8 ns)
              port map(a(0)=>d1, a(1)=>d2, c=>q1);
  U2: andComp generic map(n=>3, tdelay=>12 ns)
              port map(a(0)=>d3, a(1)=>d4, a(2)=>d5, c=>q2);
end;
```

6.1.5. A létrehozó (generate) utasítás

Ha azonos összetevőt sokszor kell beültetni ugyanazon építménybe, hatékonyabb a **port map** utasítást hurokba helyezni. Tegyük fel, hogy a *c1* összetevőt hatszor kell beültetni a *top* összetevőbe. Ez a **generate** utasítással tehető meg:

```
entity top is
  port(a,b: in std_logic_vector(4 downto 0);
         q:out std_logic_vector(4 downto 0));
end;
architecture rtl of top is
  component c1 port(a,b: in std_logic; q:out std_logic);
  end component;
  for U1: c1 use entity work.c1(rtl);
begin
  cGen: for i in 0 to 5 generate
    U: c1 port map(a(i), b(i), q(i));
  end generate cGen;
end;
```

6.1.6. Közvetlen beültetés

A közvetlen beültetés azt jelenti, hogy sem az összetevő bejelentés, sem a kialakítás nem szükséges egy összetevő beültetéséhez. A következő példa bemutatja a kétféle beültetés közti különbséget. A hagyományos leírás:

```
architecture rtl of topLevel is
  component c1
     port(a,b: in std:logic; q: out std_logic);
  end component;
for U1: c1 use entity work.c1(rtl);
begin
  U1: c1 port map(a,b,q);
end;
```

Az egyszerűsített leírás:

```
architecture rtl of topLevel is
begin
  U1: entity work.c1(rtl) port map(a,b,q);
end;
```

6.2. A when utasítás

Jelölésmód:

```
<cél> <= <kifejezés> [after <kifejezés>] when <feltétel> else
              <kifejezés> [after <kifejezés>] when <feltétel> else
             ...
              <kifejezés>;
```

Lehetséges a többszörös **when else** sorok használata is. A <*cél*> jelet hozzá kell rendelni egy jelhez, függetlenül a <*kifejezés*> értékétől. Ez azt jelenti, hogy az utasításnak egy **else** <*kifejezés*> szerkezettel kell zárulnia. Egy példa:

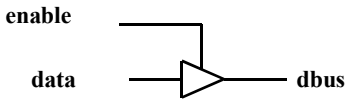
```
entity pelda is port (a,b,c: in std_logic;
  data: in std_logic_vector (1 downto 0); q: out std_logic);
end;
architecture rtl of pelda is
begin
  q<=a when data="00" else b when data="11" else c;
end;
```

6.2.1. Háromállapotú buffer és az others utasítás

A **when** utasítás nagyon hasznos, pl. háromállapotú buffer tervezése során:

```
dBus <= data when enable = '1' else 'Z';
```

A következő ábra bemutatja a szintézis eredményét:



6.2-1. ábra. A szintetizált háromállapotú buffer

Ha a *dbus* és a *data* jeleket vektorra változtatják a bejelentésben, akkor az egész *bus* vektor kimenetenként egy-egy háromállapotú buffert fog kapni. Ennek ellenére az építményben a kód alig változik:

```
dBus <= data when enable = '1' else (others =>'Z');
```

Az **(others** =>'Z') utasítás használatával az egész vektorhoz, tekintet nélkül a hosszúságára, hozzá lesz rendelve a 'Z' érték. Az egész vektorhoz egy érték hozzárendelésének ez a kezelése nagyon hatékony és a kódot karbantarthatóbbá teszi. Ha a vektor hossza változik, csak a vektor bejelentését kell módosítani és nem az építménybeli kódot.

Megengedhető számos különböző jel használata a <*feltétel*> kifejezésben, ami az utasítást hajlékonnyá és a terv számára hasznosabbá teszi. Pl.:

```
q <= a when en='0' else
     b when data="11" else
     c when enable='1' else d;
```

Megjegyzendő, hogy a feltételek olyan sorrendben kerülnek ellenőrzésre, ahogyan azok felsorolásra kerültek. Így sorról sorra kerülnek kiértékelésre a feltételek, amíg nem lesz egy feltétel igaz. Ez azt jelenti, hogy ha pl. a második sor (*data*="11") és a harmadik (*enable*='1') felcserélődnek, és ha mind a két feltétel egyidejűleg teljesül, az eredmény más lesz. Tegyük fel, hogy *en*='0', *data*="11" és *enable*='1'. Ez azt jelenti, hogy *q* megkapja a *b* jel értékét a fenti példában. Ha azonban a második és a harmadik sor felcserélődik, a *q* a *c* jel értékét fogja megkapni.

Fontos, hogy csak az egyidejű utasítások között nem számít a sorrend. Egy egyidejű utasításon belül, pl. a **when else**-ben a sorrend már fontos.

6.3. A with utasítás

Jelölésmód:

```
with <kifejezés> select
    <cél> <= <kifejezés> when <választás>;
```

Az összes lehetséges <*választás*> lehetőséget fel kell sorolni. Ha az összes maradék <*választás*>-t össze akarjuk gyűjteni egy sorba, akkor a **when others** szerkezetet lehet használni. Ebben az esetben az **others** kulcsszónak az utolsó <*választás*> lehetőségnek kell lennie. Példa:

```
entity pelda is
  port (a,b,c: in std_logic;
        data: in std_logic_vector (1 downto 0); q: out std_logic);
end;
architecture rtl of pelda is
begin
  with data select
    q <= a when "00",
         b when "11",
         c when others;
end;
```

Összehasonlítva a **when else** utasítással a **with select** utasítás nem olyan hajlékony, mivel csak egy <*kifejezés*> lehetséges. De az eredményül kapott kód olvashatóbb és szerkezetesebb.

Egy ***nyaláboló*** (multiplexor) két alábbi adatáramlási modellje egyenértékű. Az első építmény a **when** utasítást használja, a második pedig a **with** utasítást. Mindkét modell szintetizálható, mellőzve a 10 ns késleltetést.

```
entity mux is
  port (sel0,a,b: in std_logic; c: out std_logic);
end;

architecture dtf1 of mux is
begin
  c<=a after 10 ns when sel0='0' else
     b after 10 ns;
end;

architecture dtf2 of mux is
begin
  with sel0 select
    c <= a after 10 ns when '0',
         b after 10 ns when others;
end;
```

A fenti példa is mutatja, hogy a VHDL számos különböző szerkezetet nyújt ugyanazon előírás modellezésére.

6.4. Egyidejű követelmény

Ha a követelményt a VHDL kód egyidejű részében használják, az utasítás csak akkor hajtódik végre, ha egy esemény az „érzékenységi listáján”, bekövetkezik. Ha az **assert** csak az időt ellenőrzi a **now** változóval, esemény nem fog történni és az **assert** soha nem hajtódik végre. Így a követelményt egy folyamatba kell elhelyezni, amely minden órajel élnél végrehajtódik. Amikor a követelményt az egyidejű részben használják, az egyedben is használható a következőképpen:

```
entity pelda is
  port (a,b: in std_logic; q: out std_logic);
begin
  assert a/='1' or b/='1'
  report "a='1' és b='1' egyidejűleg"
  severity warning;
end;
architecture   ...
```

A követelmény használata az áramköri modellen kívül célszerű az áramkör válaszának igazolására szolgáló próbapadban is.

6.5. A tömb (block) utasítás

A tömb jelölésmódja a következő:

```
block_statement ::=
block_label :
block [ (guard _expression) ] [ is ]
    block_header
    block_declarative_part
    begin
        block_statement_part
end block [ block _label ] ;
```

A következő példában egy címkével ellátott egyszerű tömb látható, amely egy késleltetett jelhozzárendelést tartalmaz.

```
b: block
begin
  s <= '1' after 2 ns ;
end block b;
```

Az ***őrzött tömb*** (guarded block) egy olyan logikai kifejezést tartalmazó tömb, amely lehetővé vagy lehetetlenné teszi a tömbbeli meghajtót. Az őrzött tömb nem szintetizálható. A következő leírás egy őrzött tömböt mutat be.

```
architecture dtf of guardedLatch is
begin
  latch: block(clk = '1')
  begin
    q <= guarded d after 3ns;
    qn <= guarded not(d) after 5;
  end block latch;
end;
```

7. Sorrendi modellezés

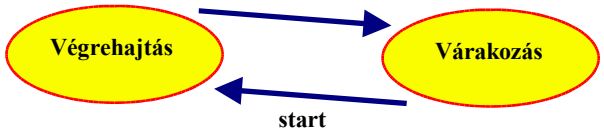
7.1. A folyamat (process) működése

A ***folyamat***ok (process) önmagukban sorrendi, de egymáshoz képest egyidejű programrészek. A folyamat fogalma a szoftvertechnológiából származik és hasonlít egy sorrendi programhoz. Ha több folyamat van egy építményben, akkor azok egyidejűleg hajtódnak végre.

Egy folyamat működése bármikor felfüggeszthető, akár valamilyen feltétel bekövetkezéséig, akár végtelen hosszú ideig. Ugyanakkor egy folyamat nem hívható kétszer, mint egy alprogram. Minden folyamat elvben a szimuláció elejétől indul és végig „él", legfeljebb a végrehajtása van felfüggesztve.

Minden **process** utasítás meghatároz egy bizonyos cselekményt vagy viselkedést, amelyet el kell végezni, amikor az érzékenységi jelei egyikének értéke megváltozik. Ezt a cselekményt a folyamatban a sorrendileg rendezett végrehajtási utasítások határozzák meg. A viselkedés eredménye a kimenti jelekre kerül alkalmazásra és esetleg más folyamatok olvassák. Ez a viselkedés történik mindannyiszor, amikor új információ érkezik a folyamat érzékenységi listájára. Amikor a folyamat utolsó utasítása végrehajtódik, a végrehajtás visszatér a folyamat első utasítására, és ez folytatódik. A végrehajtás a **wait** (vár) utasítás hatására függesztődhet fel.

A folyamat ***várakozás***i (waiting) vagy ***végrehajtás***i (executing) állapotban lehet.



7.1-1. ábra. A folyamat végrehajtásának állapotábrája

A végrehajtás alatti folyamat ***cselekvő*** (active), egyébként a folyamat felfüggesztett. Minden modellbeli folyamat lehet cselekvő egy adott időpontban, és az összes cselekvő folyamat végrehajtható a szimulációs időben számítva egyidejűleg. Azokat a **process** utasításokat, amelyek nem rendelnek hozzá értékeket a jelekhez, ***tétlen***nek (passive) hívják.

7.1.1. A folyamat jelölésmódja

Az alábbiakban a **process** utasítás jelölésmódja látható:

```
process_statement ::= [ process_label : ]
    [ postponed ] process [ (sensitivity_list) ]
        process_declarative_part
    begin
        process_statement_part
    end [ postponed ] process
        [ process_label ] ;
```

Ahhoz, hogy a folyamat ne kerüljön végtelen ciklusba, vagy *érzékenységi listá*t kell alkalmazni, vagy *várakozási* (wait) *utasítás*t, vagy pedig egy *eljáráshívás*nak (procedure call) kell lennie a folyamatban, amelynek meghatározásban szükséges egy várakozási utasítás.

A következő példa egy flip-flop leírását adja. A pozitív óraélt a **wait** utasítás észleli, és ha a clock='1' feltétel teljesül, az óraélt követő 2 ns múlva a bemenő *d* értéke a *q* kimeneten lesz.

```
process
begin
  if (clock = '1') then q <= d after 2 ns;
  end if;
  wait on clock;
end process;
```

Amikor egy folyamat elindul, akkor *D* idő (a szimulátor minimális felbontása) szükséges ahhoz, hogy visszamenjen várakozási állapotba. Ez azt jelenti, hogy a folyamat végrehajtása nem igényel időt. Egy folyamat tekinthető egy végtelen huroknak a **begin** és az **end process** utasítás között. A *D* időt arra használják, hogy a szimulátor kezelje az egyidejűséget. A valóságban a *D* idő nullával egyenlő.

7.1.2. Diszkrét esemény idő modell

A VHDL időzítési modellje azon alapul, hogy a modellezett rendszer a bemeneteire adott gerjesztésre meghatározott módon válaszol, majd a gerjesztés további változásaira várakozik. A szimuláció során az egyes események időpontjai mindig szimulációs időben mértek, azaz függetlenek a fizikai időtől. A szimulátor diszkrét időegységekben lépkedve dolgozza fel a szimulációs időpontokra ütemezett eseményeket, amelyeket egyrészt a gerjesztés, másrészt a modellezett rendszernek a gerjesztésre adott válasza határoz meg.

Esemény alatt egy jel értékének megváltozását értjük. Egy adott szimulációs időpontban az események feldolgozása egy vagy több ún. szimulációs ciklusban történik. A szimulációs ciklusnak két üteme van. Az első ütemben a jeleknek az értéke változik meg, amelyek változása az adott szimulációs időpontra van előírva. A második ütemben lefutnak azok a folyamatok, amelyeket az első ütemben megváltozott jelek tesznek tevékennyé. A folyamatok újabb jelek értékváltozásait írhatják elő, így előfordulhat, hogy az adott szimulációs időpontban további szimulációs ciklusokat is végre kell hajtani.

Egy szimulációs időpontban több szimulációs ciklus egymás utáni végrehajtásakor gondoskodni kell arról, hogy a ciklusok ne zérus idő alatt fussanak le, mert különben a jelváltozások között nem biztosítható az ok-okozati viszony. Ezért minden szimulációs ciklushoz egy elméleti ***delta*** késleltetést rendel a VHDL szimulátor, ami a szimulációs időt nem befolyásolja.

Egy adott szimulációs időpontban a szimulációs ciklusok addig ismétlődnek, amíg a folyamatokon végig nem gyűrűzik a kezdeti jelváltozások hatása. Ha az adott szimulációs időpontra nincs több jelváltozás előírva, akkor a szimulátor a következő szimulációs időpontra ütemezett eseményeket kezdi feldolgozni.

Amint egy építőelem szerkezetét és viselkedését meghatároztuk, lehetséges szimulálni az építőelemet a viselkedési leírásának végrehajtásával. Ez úgy történik, hogy az adott időtartamot diszkrét lépésekben szimulálják. Valamely időpillanatban az áramköri építőelem bemenetén egy bemeneti kapocs értékének változását szimuláljuk. Ezt egy jelen lévő *jelváltoztatás* ütemezésének hívjuk. Ha az új érték különbözik az előzőtől, akkor egy *esemény* jelenik meg, és a jelre a bemeneti kapcsaikkal kapcsolódó más építőelemek tevékennyé válhatnak.

A szimuláció egy *kezdőérték beállítási ütemmel* kezdődik, majd egy két-fokozatú *szimulációs ciklus* ismétlésével folytatódik. A kezdőérték beállítási ütemben az összes jel a kezdeti értéket megkapja, a szimulációs idő zérusra lesz beállítva, és minden egyes építőelem viselkedési programja végrehajtódik. Ennek eredménye rendszerint egy bizonyos idő múlva a kimeneti jeleken való jelváltoztatás.

Egy szimulációs ciklus első fokozatában a szimulált idő arra a legkorábbi időpontra áll be, amelynél az ütemezés szerint fellép egy jelváltoztatás. Erre az időre ütemezett összes jelváltoztatás végrehajtódik, ami jelváltozási események fellépését jelentheti egyes jeleken.

A második fokozatban az összes építőelem, amelyre hatnak az első fokozatbeli események, végrehajtják a viselkedési programjukat. Ezek a programok rendszerint további jelváltoztatásokat ütemeznek a kimeneti jeleiken. Mikor az összes viselkedési program végrehajtódott, a szimulációs ciklus megismétlődik. Ha nincsen több ütemezett jelváltoztatás, az egész szimuláció befejeződött.

A szimuláció célja az időtartam alatt a rendszerben bekövetkezett változásokról való információk összegyűjtése. Ez úgy történhet, ha a szimulációt a *szimulációs monitor* ellenőrzése mellett futtatjuk. A monitor lehetővé teszi a jelek és más állapot információk figyelését vagy tárolását egy nyomvonal fájlban későbbi elemzés számára. Szintén lehetővé teszi a szimuláció interaktív lépésenkénti végrehajtását úgy, mint egy interaktív program-hibakereső.

A ***delta idő***t a VHDL-ben a sorrendi események sorba állítására használják. A két sorrendi esemény közötti időt *delta késleltetés*nek nevezik. Egy delta időnek nincs valósidőbeli egyenértéke, hanem miközben telik (folyik valamilyen végrehajtás), a szimulációs idő nem halad. A jelhozzárendeléskor az érték nem rendelődik közvetlenül a jelhez hozzá, hanem legkorábban egy delta után, ahogy ez a következő példán is látszik:

```
b<=a;    -- A „b” jelhez hozzárendeli az „a” jel értékét
         -- egy delta késleltetés múlva.
```

Egy kombinációs logikai tömbben, ahol az összes elemnek 0 ns késleltetése van, az összes hozzárendelés 0 ns-nál fog bekövetkezni, de sok delta késleltetést tartalmazhat. A VHDL szimulátor kiszámolja, hogy hány delta idő kell, amíg az összes jel stabillá nem válik. Ha a VHDL kód helytelen, akkor van rá esély, hogy a terv a végtelenségig fog rezegni. Azért, hogy a szimulátort megvédjük az elszállástól, beállítható, hogy hány delta idő után álljon le, jellemző érték erre az 1000 delta idő. A következő példa helyes VHDL kód, de végtelenségig rezgő tervet hoz létre:

```
        q<= not q;
```

A *q* jel egy delta idő után frissülni fog saját maga fordítottjával. Ez azt okozza, hogy a sor ismét végrehajtásra fog kerülni, és a *q* újra frissülni fog egy delta idő után, stb. Ezt a gondot könnyű megoldani egy késleltetés beillesztésével:

```
        q<= not q after 10 ns;
```

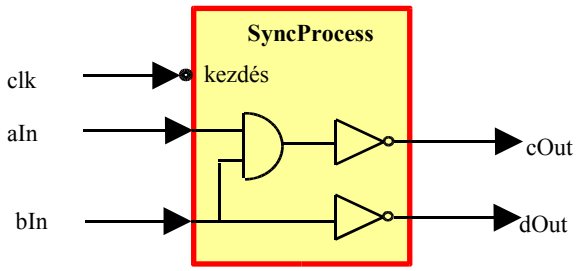
Megjegyzendő, hogy ha a fenti kódot szintetizálják, akkor ez egy aszinkron visszacsatolást eredményez, amely rendszerint nem kívánatos a tervben.

7.1.3. Egy példa a folyamat működésére

Az alábbi folyamat akkor kezdődik, amikor a *clk* jel lemegy alacsony értékre. Ezután elintéz két utasítást és vár a következő élre. A programozónak nem kell hurkot adni a folyamathoz, mivel az mindenképpen újra fog indulni a kezdetétől.

```
SyncProcess: process
begin
  wait until clk='0';
  cOut<= not (aIn and bIn);
  dOut <= not bIn;
end process;
```

A modellbeli két utasítás a szimulátor felbontásával megegyező D időn belül fog végrehajtódni.



7.1-2. ábra. Egy folyamat hardver megvalósítása

7.1.4. A folyamatbeli késleltetés modellezése

A gyakorlatban az utasítások megvalósítása különböző hosszú időt igényelnek. Ez a késleltetés a folyamatban a következők szerint modellezhető:

```
cOut<= not (aIn and bIn) after 20 ns;
dOut<= not aIn after 10 ns;
```

A fentiek azt jelentik,, hogy a *cOut* a folyamat indulása után 20ns-mal, a *dOut* pedig 10 ns-mal később lesz érintve. A szimulátor a következő táblázatban megadott eseménysorban fogja a *cOut* és *dOut* számára az eredményt előállítani.

Szimulációs idő	Esemény a szimulátorban
0 ns	aIn = 1
0 ns	bIn = 0
x + 20 ns	cOut = 1
x + 10 ns	dOut = 1

7.1-1. táblázat. Az időbeli eseménysor a szimulátorban.

Ha *aIn* vagy *bIn* megváltozik és a *clk*-n egy eső él megjelenik, akkor egy másik esemény fog bekapcsolódni a sorba az előírt késleltetéssel ahhoz az időhöz képest, amikor a változás bekövetkezett.

7.1.5. A folyamat típusai

Kétfajta folyamat van a VHDL-ben: a ***kombinációs*** (combinational) és az ***órajeles*** (clocked) folyamat. A kombinációs folyamatot a hardverbeli kombinációs logika tervezésére használják. Az órajeles folyamatok flip-flopokat és esetleg kombinációs logikát eredményeznek.

A) A kombinációs folyamat

A kombinációs folyamatban az összes bemeneti jelet tartalmaznia kell a folyamat érzékenységi listájának. Ezek a jelek azok, amelyek a <= művelet jobb oldalán vannak, vagy az *if/case* utasításban szerepelnek. Ha egy jel kimarad, a folyamat a szintézis után a hardverben nem kombinációs logikaként fog viselkedni. A valódi hardverben ugyanis a kimeneti jelek csak akkor változhatnak, ha a kombinációs logika egy vagy több bemenő jele megváltozott.

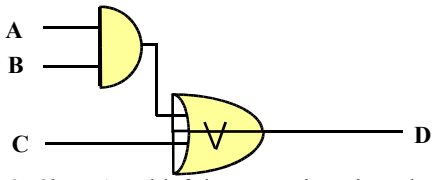
Ha ezeknek a jeleknek egyikét mellőzik az érzékenységi listából, akkor a folyamat nem válik tevékennyé, amikor a mellőzött jel értéke megváltozik és nem fog új érték hozzárendelődni a folyamat kimeneti jeleihez. A VHDL szabvány megengedi a jelek mellőzését az érzékenységi listából.

Ha egy jel kimarad egy szintetizálandó VHDL tervben az érzékenységi listából, akkor a VHDL szimuláció és a szintetizált hardver eltérően fog viselkedni.

Egy példa a kombinációs folyamatra:

```
process (a,b,c)
  begin
    d<= (a and b) or c;
end process;
```

A szintézis eredménye:



7.1-3. ábra. A példafolyamat szintézisének eredménye.

B) Nemteljes kombinációs folyamat

Olyan terv esetében, ahol kombinációs folyamatok vannak, a folyamat összes kimeneti jeléhez hozzá kell rendelni egy értéket mindannyiszor, amikor a folyamat végrehajtódik. Ha ez a feltétel nem teljesül, a jel megtartja az értékét. Ezt észleli a szintézis eszköz, és a helyzetet egy latch bevonásával oldja meg arra a kimenetre nézve, amelyhez nem rendeltek értéket a folyamatban. A latch zárva lesz, amikor a jelnek a régi értéket kell tartania. Működésében a VHDL kód és a hardver azonos lesz. De egy kombinációs folyamat célja kombinációs logika létrehozása. Ha latch-eket vonnak be, a megnőtt számú kapuk lerontják az időzítést. Ami még nagyobb baj, a latch rendszerint megtöri az önműködő vizsgáló vektorok létrehozására való vizsgálati szabályokat. Azokat a folyamatokat, amelyek tévedésből latch-eket eredményeznek, ***nemteljes kombinációs folyamat***oknak (incomplete combinational process) hívják. A megoldás az, hogy a folyamatban olvasott összes jelet bele kell foglalni az érzékenységi listába.

C) A halasztott (postponed) folyamat

A ***halasztott*** (**postponed**) folyamatot a VHDL-93 szabvány vezette be. A halasztott folyamat használatával lehetséges egy adott szimulációs időpontban egy folyamatot az utolsó delta időciklusban végrehajtani.

Mint a többi folyamatot, a halasztott folyamatot is egy érzékenységi listával vagy egy várakozási utasítással lehet tevékenyíteni. A különbség az, hogy egy halaszott folyamat nem kezdi el a végrehajtást annál a delta eseménynél, amely tevékenyíti a folyamatot, hanem *vár addig, amíg az összes jel stabillá nem válik*. Ez azt jelenti, hogy abban az időben már nincs több delta sorbaállítva, így a halasztott folyamat kerül egy adott szimulációs időpontban utoljára végrehajtásra. Ha több halasztott folyamatot használnak, a VHDL-93 szabvány nem határozza meg, hogy melyik kerül először végrehajtásra. Ezért egynél több halasztott folyamat alkalmazása kerülendő.

A halasztott folyamatokat nem támogatják a szintézis során. Szerepe a szimulációban van, mivel biztosítja, hogy a jelek az értéküket a legutolsó delta ciklusban kapják meg, ami könnyebbé teszi bizonyos szimulációs modellek létrehozását.

A halasztott folyamat nem növelhető további delta késleltetésekkel. Ez azt jelenti, hogy a halasztott folyamatbeli összes értékadásnak **after**-rel megadott késleltetéssel kell rendelkeznie. Példa halasztott folyamatokra:

```
architecture bhv of pelda is
begin
  process
  begin
```

```
      ...
  end process;
  postponed process (a,b)
  begin
    if a>b then
      q<='1' after 5 ns;
    else
      q<='0' after 5 ns;
    end if;
  end process;
end;
```

7.2. Az if-then-else utasítás

Az **if-then-else** utasítás jelölésmódja a következő:

```
if_statement::=[if_label:]
       if condition then
              sequence_of_statements
       { elsif condition then
              sequence_of_statements }
       [ else sequence_of_statements ]
       end if [ if_label ] ;
```

Több **elsif** is megengedett, de csak egy **else**. A következő példában ha az *a* értéke nagyobb mint 0, akkor b-hez a értékét rendelik hozzá, egyébként **abs**(*a*+1)-et.

```
if a > 0 then
  b:= a;
else
  b:= abs(a+1);
end if;
```

Tekintsük a következő leírást, amely egy ***nyaláboló*** (multiplexor) áramkört modellez.

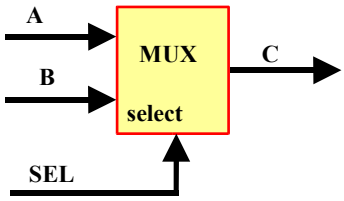
```
entity pelda is port (a, b,sel :in bit; c:out bit); end;

architecture bhv of pelda is
begin
  process (a, b, sel)
  begin
    if sel = '1' then
      c <= b;
    else
      c <= a;
    end if;
  end process;
end bhv;
```

A hozzá tartozó gerjesztés modellje:

```
entity peldaStm is
  port (a,b,cel: out bit; c: in bit);
end;
```

```
architecture dtf of peldaStm is
begin
  a<='0' after 0 ns, '1' after 10 ns, '1' after 20 ns;
  b<='0' after 0 ns, '1' after 5 ns, '0' after 15 ns;
  sel<= '0' after 1 ns, '1' after 10 ns;
end;
```

A fenti eljárást szintézis bemeneti adataként használva egy kétbemenetű ***nyalábol***t (**mul**tiplexor, **MUX**) kapunk, amelynek bemenetei: *a* és *b*, a választó bemenet: *sel* és a kimenet: *c*. A következő ábrán látható a szintézis eredménye.



7.2-1. ábra. Egy if utasítás alapján szintetizált nyaláboló

7.3. Az eset (case) utasítás

Az ***eset*** (**case**) utasítás jelölésmódja a következő:

```
case_statement ::=
[ case_label : ]
    case expression is
        case_statement_alternative
        { case_statement_alternative }
    end case [ case_label ];
```

A következő példában az *a* bit értékét ellenőrzik. Ha az 0, akkor *q* a "0000" értéket kapja 2 ns múlva, másként az "1111" értéket, szintén 2 ns múlva.

```
case a is
   when '0' => q <= "0000" after 2 ns ;
   when '1' => q <= "1111" after 2 ns ;
end case;
```

7.3.1. Az others használata a case utasításban

A **case** utasításban az összes választást fel kell sorolni. A választásoknak nem szabad átlapolniuk. Ha a **case** kifejezés sok értéket tartalmaz, amelyekhez ugyanaz a választás tartozik, az **others** is használható, ahogy azt a következő példa mutatja.

```
entity cir is
```

```
  port (a: in integer range 0 to 30; q: out integer range 0 to 6);
end;
architecture bhv of cir is
begin
  P1: process(a)
  begin
    case a is
      when 0 =>            q<=3;
      when 1 | 2 =>        q<=2;
      when others =>       q<=0;
    end case;
  end process;
end;
```

A gerjesztési vektor létrehozása:

```
entity stm is
  port (a: out integer range 0 to 30; q: in integer range 0 to 6);
end;

architecture dtf of stm is
begin
  a<=0 after 10 ns,
     1 after 20 ns,
     6 after 30 ns,
     2 after 40 ns,
     0 after 50 ns,
     10 after 60 ns;
 end;
```

A próbapad:

```
entity bnc is end;
use std.textio.all;
architecture str of bnc is
  -- Az épitmény bejelentési része, benne a helyi összetevő bejelentés
  component cir port (a:in integer range 0 to 30;
                      q:out integer range 0 to 6); end component;
  component stm port (a:out integer range 0 to 30;
                      q:in integer range 0 to 6); end component;
  -- A cir nevű tervezési targyhoz egy bhv nevű építmény
  -- található a work könyvtárban.
  for all:cir use entity work.cir(bhv);
  for all:stm use entity work.stm(dtf);
  signal a,q:integer:=0;
  signal aid:boolean;
begin
  C1:cir port map(a,q);
  C2:stm port map(a,q);
  Fejlec:process
    variable dline:line;
  begin
    write (dline,string'("   time     a   q   aid"));
    writeline (output,dline);
```

```
    write (dline,string'("========================"));
    writeline (output,dline);
    aid<=True;
    wait;
  end process;
  AdatFigyeles:process (a,q,aid)
    variable dline:line;
  begin
    if aid=True then
      write (dline,now,right,7);
      write (dline,a,right,6);
      write (dline,q,right,4);
      write (dline,aid,right,6);
      writeline (output,dline);
    end if;
  end process;
end;
```

Szimulációs eredmények:

```
#     time      a    q    aid
# ========================
#     0 ns      0    3   TRUE
#    20 ns      1    3   TRUE
#    20 ns      1    2   TRUE
#    30 ns      6    2   TRUE
#    30 ns      6    0   TRUE
#    40 ns      2    0   TRUE
#    40 ns      2    2   TRUE
#    50 ns      0    2   TRUE
#    50 ns      0    3   TRUE
#    60 ns     10    3   TRUE
#    60 ns     10    0   TRUE
```

7.3.2. Összehasonlító tervezése

Az ***összehasonlító***k (comparators) a vektorok méretének összehasonlítására szolgálnak. Az alábbi összetevő összehasonlít két hárombites vektort, és létre hozza a *comp* kimeneti jelet. A *comp* kimeneti jel jelentését meghatározza a *selF* bemeneti jel:

selF	comp
00	'1', ha a=b
01	'1', ha a<b
10	'1', ha a>b
11	'0'

7.3-1. táblázat. Az összehasonlító kimeneti jelének jelentése a bemenettől függően

A következő leírás bemutatja egy ilyen összehasonlító modelljét.

```
library ieee;
```

```
use ieee.std_logic_1164.all; use ieee.std_logic_unsigned.all;

entity comp is port(a,b: in std_logic_vector(2 downto 0);
                    selF: in std_logic_vector(1 downto 0);
                       q: out boolean);
end;

architecture viselk of comp is
begin
  process(selF,a,b)
  begin
    case selF is    when "00"    =>q<=a=b;
                    when "01"    =>q<=a<b;
                    when "10"    =>q<=a>b;
                    when others  =>q<=false;
    end case;
  end process;
end;
```

7.3.3. Értéktartomány használata

Lehetséges meghatározni ***értéktartomány***t (range) egy választási listában. Ez a **to** és a **downto** kulcsszóval tehető, ahogy azt a következő példa bemutatja.

```
entity pelda is
    port (a: in integer range 0 to 30;
          q: out integer range 0 to 6);
end;
architecture viselk of pelda is
begin
  P1: process(a)
  begin
    case a is
      when 0 =>            q<=3;
      when 1 to 17 =>      q<=2;
      when 23 downto 18 =>q<=6;
      when others =>       q<=0;
    end case;
  end process;
end;
```

Értéktartománnyal kapcsolatos gyakori tervezési hiba

Nem lehet értéktartományt meghatározni egy vektorhoz, ha a vektornak nincs értéktartománya, pl. a következő megoldás hibás:

```
entity pelda is      -- Ez a példa  H I B Á S !
port (a: in std_logic_vector(4 downto 0);
      q: out std_logic_vector(2 downto 0));
end;
architecture viselk of pelda is  begin
  P1: process(a)
```

```
  begin
    case a is
      when "00000"            =>q<="011";
      when "00001" to "11110" =>q<="010"; -- Hiba
      when others             =>q<="000";
    end case;
  end process;
end;
```

Ha mégis szükséges tartomány megadása, akkor az *std_logic_vector* típust először egy egésszé kell átváltani. Ez úgy történik, hogy egy egész típusú változót bejelentenek a folyamatban, és ezután átváltják a vektort egy átváltó függvény (a példában a *conv_integer*) használatával, ahogy ezt a következő példa bemutatja.

```
entity pelda is            -- egy jó megoldás
port (a: in std_logic_vector(4 downto 0);
      q: out std_logic_vector(2 downto 0));
end;
architecture viselk of pelda is  begin
  P1: process(a)
  variable int: integer range 0 to 31;
  begin
    int:=conv_integer(a);
    case int is
      when 0 =>q<="011";
      when 1 to 30 =>q<="010"; -- Jó!
      when others =>q<="000";
    end case;
  end process;
end;
```

7.3.4. Az összefűzés használata

Nem lehetséges az ***összefűzés*** (concatenation) használata arra, hogy összekapcsoljanak vektorokat egy választássá a következő egy rossz megoldás, mivel a **case**-beli kifejezésnek változhatatlannak kell lennie:

```
entity pelda is            -- H I B Á S!
port (a,b: in std_logic_vector(2 downto 0);
        q: out std_logic_vector(2 downto 0));
end;
architecture viselk of pelda is
begin
  P1: process(a,b)
  begin
    case a & b is                  -- Hibás
      when "000000" =>q<="011";
      when "001110" =>q<="010";
      when others =>b<="000";
    end case;
  end process;
end;
```

A megoldás egy változó bevezetése a folyamatban, amelyhez az a & b értéket hozzá lehet rendelni, vagy az altípusokhoz használatos ***minősítő*** (qualifier) alkalmazása. Az alábbiakban egy példa látható a változó alkalmazására az összefűzéshez.

```
entity pelda is
port (a,b: in std_logic_vector(2 downto 0);
        q: out std_logic_vector(2 downto 0));
end;

architecture viselk of pelda is
begin
  P1: process(a,b)
    variable int: std_logic_vector (5 downto 0);
  begin
    int := a & b;
    case int is
      when "000000" =>q<="011";
      when "001110" =>q<="010";
      when others =>b<="000";
    end case;
  end process;
end;
```

A következő példa a minősítőnek az összefűzéssel való használatát mutatja be:

```
entity pelda is
port (a,b: in std_logic_vector(2 downto 0);
        q: out std_logic_vector(2 downto 0));
end;

architecture viselk of pelda is
begin
  P1: process(a,b)
    subtype myType is std_logic_vector (5 downto 0);
  begin
    case myType'(a & b) is
      when "000000" =>q<="011";
      when "001110" =>q<="010";
      when others =>b<="000";
    end case;
  end process;
end;
```

7.3.5. Többszörös értékadás (hozzárendelés)

A VHDL egyidejű részében minden egyes jelhozzárendeléshez külön meghajtó kell, ami szokásosan nem kívánatos. A VHDL sorrendi részében lehetséges ugyanazon jelnek többször értéket adni azonos folyamaton belül anélkül, hogy több meghajtó kellene ugyanazon jelhez. A jelértékadásnak ez a módja használható a jelekhez egy alapértelmezett érték hozzárendelésére a folyamatban. Ezt az értéket azután felülírja más jelhozzárendelés. A következő két példa azonos a szimuláció és a szintézis eredménye szempontjából is. Az első példa:

```
architecture viselk of pelda is
begin
  P1: process (a)
  begin
    case a is
      when "00" =>q1<='1';   q2<='0'; q3<='0';
      when "10" =>q1<='0';   q2<='1'; q3<='1';
      when others =>q1<='0'; q2<='0'; q3<='1';
    end case;
  end process;
end;
```

A második példa:

```
architecture viselk of pelda is
begin
  P1: process (a)
  begin
    q1<='0'; q2<='0'; q3<='0';
    case a is
      when "00" =>q1<='1';
      when "10" =>q2<='1'; q3<='1';
      when others =>q3<='1';
    end case;
  end process;
end;
```

Ha összehasonlítjuk a két példát, látható, hogy kevesebb jelhozzárendelés kell a második példában. A második példabeli megoldás magyarázata az, hogy a szimulációs idő nem kerül be a folyamatba, azaz a jelértékadások csak felülírják egymást a sorrendi VHDL kódban. Természetesen, ha ugyanazon jelhez történik hozzárendelés különböző folyamatokban, a jelnek több meghajtója lesz.

7.3.6. A null utasítás

A **null** utasítás jelölésmódja a következő:

```
null_statement::=
[ label : ] null;
```

A **null** utasítás kifejezetten véd bármely véghezviendő cselekvéstől. Ez az utasítás azt jelenti, hogy: „*ne tégy semmit*”. Ez az utasítás pl. akkor használható, ha az alapértelmezett jelértékadást egy folyamatban használjuk, és egy választási lehetőség a **case** utasításban ne változtasson ugyanazon értékek esetén. A következő példa mutat egy ilyen esetet:

```
architecture bhv of pelda is
begin
  P1: process (a)
  begin
    q1<='0';
    q2<='0';
    q3<='0';
    case a is
      when "00" =>q1<='1';
      when "10" =>q2<='1';
                  q3<='1';
      when others => null;
    end case;
  end process;
end;
```

A fenti példában a **null** nem hagyható ki. Más esetekben csak a kód olvashatósága növekszik, ha a **null** utasítás bele van foglalva. Ha a **null**-t mellőzzük, az a veszély lép fel, hogy valaki más olvassa a VHDL kódot esetleg nem lesz biztos abban, hogy a VHDL tervező nem felejtett-e el egy jelhozzárendelést.

7.4. Hurokképző utasítások

A ***hurok*** (loop) hasznos egy- vagy kétdimenziós vektorokkal való tervezéskor. Két különböző hurokképző utasítás van a sorrendi VHDL-ben, az egyszerű hurok, a **for** hurok és a **while** hurok. Jellemző rájuk, hogy nem lehet a hurok indexének értékét a folyamatban megváltoztatni.

7.4.1. Hurok for vagy while nélkül

Egy példa a **for** vagy **while** nélküli hurokra:

```
loop
   wait until clock = '1' ;
   q <= d after 1 ns;
end loop;
```

Ez egy végnélküli hurok, amelyben a *q* jelhez a *d* értéke van hozzárendelve 1 ns késleltetés után, ha a *clock*='1' feltétel teljesült.

7.4.2. For hurokképző utasítás

A **for** utasítás jelölésmódja az alábbi:

```
[ loop_label: ] for <identifier> in <discrete_range> loop
    sequence_of_statement
    end loop [ loop_label ] ;
```

A **for** utasítás esetén a hurok tartományát rögzíteni kell, ahogy ezt a következő példa bemutatja:

```
entity pelda is
  port (a,b,c: in std_logic_vector (4 downto 0);
             q: out std_logic_vector (4 downto 0));
end;
architectue bhv of pelda is
begin
  process (a,b,c)
  begin
    for i in 0 to 4 loop -- Az i hurokindex nem jelenthető be
      if a(i)=i then
        q(i)<=b(i)
      else
        q(i)<=c(i);
      end if;
    end loop;
  end process;
end;
```

7.4.3. While hurokképző utasítás

A **while** utasítás jelölésmódja a következő:

```
[loop_label:] while <condition> loop
      sequence_of_statement
end loop [loop_label];
```

Csak a fejlett szintézis eszközök támogatják a **while** hurkot. Az alábbiakban bemutatásra kerülő példabeli eset is csak szimulálható, de nem szintetizálható. Ahogy a példa bemutatja, a hurok megszakítható egy belső **exit** (kilépés) utasítással.

```
process (a,b,resetN)
  variable i:integer range 0 to 31;
begin
  if resetN='0' then
    i:=0;
    q<=(others=>'0');
  else
    while q<12 loop
      if i=31 then
        exit;
      else
        i<=i+1;
        q<=a(i)+b(i)+q;
      end if;
    end loop;
  end if;
end process;
```

7.4.4. A következő (next) utasítás

A ***következő*** **(next)** utasítás megszakítja az éppen futó hurkot és előreugrik a hurok következő iterációjára, jelölésmódja az alábbi:

```
next_statement ::=
[ label : ] next [ loop _label ] [ when condition ] ;
```

A következő példa bemutatja a ***feltételes*** (conditional) **next** utasítás használatát. Ebben a **next** utáni utasításokat figyelmen kívül hagyja a szimulátor, ha a *value*=3.

```
while value > 0 loop
   next when value = 3 ;
   tab(value) := value rem 2 ;
   value := value / 2 ;
end loop;
```

7.4.5. A kilépés (exit) utasítás

A ***kilépés*** **(exit)** utasítás jelölésmódja a következő:

```
exit_statement ::=
[ label : ]
exit [ loop _label ] [ when condition ] ;
```

A következő példában a hurkot elhagyja a szimulátor, ha a *value*=0.

```
loop
   exit when value = 0 ;
   tab(value) := value rem 2 ;
   value := value / 2 ;
end loop;
```

7.5. A vár (wait) utasítás

A **wait** (***vár***) utasítás jelölésmódja az alábbi:

```
wait_statement::=
     [ label : ] wait [ sensitivity_clause ]
    [ condition_clause ]
    [ timeout_clause ] ;
```

A **wait** utasítás sorrendi, nem lehet függvényben használni, de eljárásban és folyamatban alkalmazható.

7.5.1. Feltétel nélküli várakozás

A következő példának az eredményeképpen a folyamat végrehajtása tartósan abbamarad:

```
wait;
```

7.5.2. Időtartamra várakozás

A soron következő példában a folyamat 1 ms-ra megszakad. A szintézis során, ha az alap órajel pl. 1μs-os, akkor egy számlálót ültetnek be, amely létrehozza 1 ms-os órajelet.

```
wait for 1 ms ;
```

Lehetséges a **wait** utasítás következőképpen való használata is:

```
constant period:time:=10 ns;
wait for 2*period;
```

7.5.3. Értékre várakozás

Az alábbi példában a folyamat megszakad, amíg a *clk* értéke nem lesz '1'.

```
entity cir is port (a,clk: in bit; y: out bit); end;

architecture bhv of cir is
begin
  process
  begin
    y <= a;
    wait until clk='1';
  end process;
end;
```

A fenti példa vizsgálatához szükséges gerjesztéskeltő a következő:

```
entity stm is port (a,clk: out bit; y: in bit); end;

architecture dtf of stm is
begin
  a<='1' after 20 ns,       '0' after 25 ns,
     '1' after 45 ns,        '0' after 58 ns;
  clk<='1' after 10 ns,   '0' after 30 ns,
       '1' after 50 ns,       '0' after 70 ns;
end;
```

A példa vizsgálatára való ***próbapad*** (test bench) az eredmények beépített kijelzésével:

```
entity bnc is end;

use std.textio.all;
architecture str of bnc is
  component cir port (a,clk:in bit;y:out bit); end component;
  component stm port (a,clk:out bit;y:in bit); end component;
  for all:cir use entity work.cir(bhv);
  for all:stm use entity work.stm(dtf);
  signal a,clk,y:bit;
begin
  C1:cir port map(a,clk,y);
  C2:stm port map(a,clk,y);

  Fejlec:process
    variable dline:line;
  begin
    write (sdline,string'("  IDO     a  clk  y "));
    writeline (output,dline);
    wait;
  end process;
```

```
  AdatFigyeles:process (a,clk,y)
    variable dline:line;
  begin
    write (dline,now,right,7);
    write (dline,a,right,6);
    write (dline,clk,right,4);
    write (dline,y,right,4);
    writeline (output,dline);
  end process;
end;
```

A szimulátor futási eredménye:

```
#     IDŐ       a   clk   y
#    10 ns      0    1    0
#    20 ns      1    1    0
#    25 ns      0    1    0
#    30 ns      0    0    0
#    45 ns      1    0    0
#    50 ns      1    1    0
#    50 ns      1    1    1
#    58 ns      0    1    1
#    70 ns      0    0    1
```

Egy másik példamodell a következő:

```
wait until clk = '1';
d <= '1';
```

Ez utóbbi esetben várni kell egy jelre, mielőtt egy bizonyos érték a *d* kimenetre kerülne. Ennek megvalósítása egy flip-flopot igényel.

7.5.4. Értékváltozásra várakozás

A soron következő példában a folyamat megszakad, amíg az *a* és a *b* jelek valamelyike nem változik.

```
wait on a, b;
```

A fenti példával egyenértékű a folyamatbeli érzékenységi lista. Ezt mutatja be az alábbi példában lévő két folyamat:

```
P0: process (a, b)
begin
  if a>b then
    q<='1';
  else
    q<='0';
  end if;
end process;

P1: process
begin
  if a>b then
```

```
    q<='1';
  else
    q<='0';
  end if;
  wait on a,b;
end process;
```

Az első folyamatban a folyamat végrehajtását az *a* vagy a *b* jel változása váltja ki. A jeleken megjelenő változást a'**event** or b'**event** alakkal is ki lehet fejezni. Ahhoz, hogy a második példa egyenértékű legyen az elsővel a **wait on** utasítást a folyamat végére kellett elhelyezni, mivel az összes folyamat végrehajtódik a szimuláció elindulásánál, amíg el nem érik az első várakozási utasításukat, de azok a folyamatok is végrehajtódnak egyszer, amelyeknek érzékenységi listájuk van és annak szereplőinek nem változott az értéke. Ha a **wait on** utasítást bárhol máshol helyeznénk el a folyamatban, a kimeneti jel különbözne a szimuláció elindulásakor.

Ha érzékenységi listát használunk egy folyamatnál, nem engedélyezett a **wait** utasítás használata a folyamaton belül. Több **wait** utasítást azonban szabad ugyanazon folyamaton belül alkalmazni. A **wait** használati lehetőségei össze is kapcsolhatók, pl.:

```
wait on a until b='1' for 10 ns;
```

A fenti példában a **wait** feltétele kielégül, amikor *a* megváltozik, *b* egyenlővé válik '1'-gyel vagy 10 ns várakozás letelik.

7.5.5. A folyamat várakoztatására vonatkozó megoldások összehasonlítása

A következő példa több várakoztatási lehetőséget hasonlít össze.

```
type a: in bit; c1, c2, c3, c4, c5: out bit;
```

1. eset:

```
process (a)
begin
  c1<= not a;
end process;
```

2. eset:

```
process
begin
  c2<= not a;
  wait on a;
end process;
```

3. eset:

```
process
begin
  wait on a;
  c3<= not a;
end process;
```

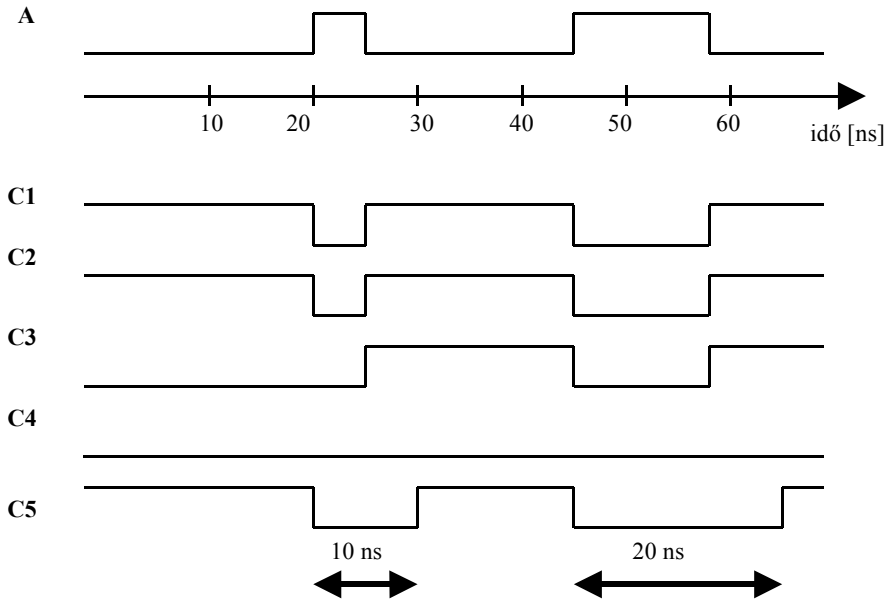
4. eset:

```
process
begin
  wait until a='1';
  c4<= not a;
end process;
```

5. eset:

```
process
begin
  c5<= not a;
  wait until a='1' for 10 ns;
end process;
```

A szimulációs eredmények a következő ábrán láthatók.



7.5-1. ábra. Példák a wait utasításra (szimulációs eredmények)

Egyes szintézis eszközök csak a folyamatbeli érzékenységi listát támogatja, így az 1. eset szintetizálható biztosan. Néhány szintézis eszköz a 2. esetbeli leírást is lefogadja, de a 3. eset biztosan nem szintetizálható. A 4. eset egy órajeles folyamat és egy D flip-flopot eredményez a szintézise. Az 5. eset is csak a szimulációra alkalmazható.

7.6. A kimenti jelek vizsgálata követelménnyel

A követelmény utasítás használatára példa a következő RS flip-flop modell.

```
entity rsff is port (s,r: in bit; q,qBar: out bit); end;

architecture bhv of rsff is
begin
  process (s,r)
    variable Internal1, Internal2: Bit:='0';
    variable LastState: Bit:='0';
  begin
    assert not (S='1' and R='1')
      report "S és R mindketten egyenlők '1'-gyel"
        severity error;
    if (S='0' and R='0') then LastState:=LastState;
    elsif (S='0' and R='1') then LastState:='0';
    else LastState:='1';
    end if;
    Q   <=LastState     after 2 ns;
    QBar<=not LastState after 2 ns;
  end process;
end;
```

Az alábbi kódnál pl. a szimulátor megáll a 900 ns-t elérve.

```
process (clk)
begin
  assert now < 900 ns
    report "a szimulátor megállítása (max. szimulációs idő 900 ns)"
      severity Failure;
end process;
```

A következő esetben, ha a kimeneti és a bementi jelek megegyeznek, a szimulátor megáll és egy hibakódot irat ki. Az alábbi példában egy **if** utasítás határozza meg, hogy a követelés végrehajtásra kerüljön-e vagy sem. Ebben az esetben a követelés feltétel mindig hamis (false).

```
process (outModel, inModel)
begin
  if outModel='1' and inModel='1' then
    assert outModel/='1' or inModel/='1'
      report "A ki- és bemeneti jelek egyszerre vannak '1'-ben"
        severity Error;
  end if;
end process;
```

7.7. Sorrendi jelentés követelmény nélkül

Ha a **report** utasítást a VHDL sorrendi részében használják, az **assert** utasítás elhagyható, ahogy azt az alábbi példák bemutatják.

```
process (a,b)
```

```
begin
  if  a='1' and b='1' then
    assert false
      report "a='1' and b='1'";
  end if;
...
end process;
```

A következő felírás is megfelelő:

```
process (a,b)
begin
  if  a='1' and b='1' then
    report "a='1' and b='1'";
  end if;
...
end process;
```

A követelés használata nélkül pl. egy erre a célra kijelölt jelet használnak az ellenőrzés eredményének jelzésére:

```
process (outModel, inModel)
begin
  if outModel='1' and inModel='1' then
    testOk<='0';
  end if;
end process;
```

7.8. *A* now *(most) változó*

A **now** változót a VHDL szabvány határozta meg. Ez a szimulátor belső abszolút szimulációs idejét hordozza. Az **assert** utasítással együtt ellenőrizhető, hogy a szimulátor ideje nem haladta-e meg, pl. az 1000 ns-t, ahogy azt a következő példa bemutatja.

```
process (clk)
begin
  assert now < 1000 ns
    report "Szimulátor megállítása (max. szimulációs idő 1000 ns)"
      severity failure;
end process;
```

Az előbbi példában az **assert** egy sorrendi követelmény utasítás.

8. Kialakítás

A ***kialakítás*** (configuration) egy építményt köt össze egy tervezési egyeddel. Sok fejlesztőeszközben kihagyják, vagy az egybeépített tervező eszköz végzi el. A következő példában VHDL nyelven kerül leírásra a kialakítás. A *mux* egyednek két különböző építménye van, az *rtl* és a *bhv*. A kialakítás szabja meg, hogy melyiket kell szimulálni. A *muxBhv* kialakítás a *bhv* nevű építményt alkalmazza, a *muxRtl* pedig az *rtl*-t, ahogy ezt a következő példa is mutatja:

```
entity mux is
  port(a,b,c,d: in std_logic_vector(3 downto 0);
            sel: in std_logic_vector(1 downto 0);
              q: in std_logic_vector(1 downto 0));
end;

architecture bhv of mux is
begin
  process(a,b,c,d,sel)
    variable int: std_logic_vector(3 downto 0);
  begin
    case sel is
       when "00"           =>     int<=a;
       when "01"           =>     int<=b;
       when "10"           =>     int<=c;
       when "11"           =>     int<=d;
       when others         =>     int<=(others =>'X');
    end case;
    q<=int after 10 ns;
  end process;
end;

architecture rtl of mux is
begin
  process(a,b,c,d,sel)
  begin
    case sel is
       when "00"           =>     q<=a;
       when "01"           =>     q<=b;
       when "10"           =>     q<=c;
       when others         =>     q<=d;
    end case;
  end process;
end;

-- Kialakítás a bhv építmény használatához
configuration muxBhv of mux is
  for bhv
```

```
  end for;
end muxBhv;

-- Kialakítás az rtl építmény használatához
configuration muxRtl of mux is
  for rtl
  end for;
end muxRtl;
```

A felhasznált építmény attól függ, hogy melyik kialakítást fordítják le. Az összetevő előírása a kialakítás egy egyszerűsített alakja. A kialakítást elsősorban szimulálható modellek tervezésekor használják, a hardvertervezésnél a tervező eszközök rendszerint elkerülik az alkalmazását.

9. Tervezési fogások és módszerek példákon bemutatva

A következőkben a VHDL alapú rendszertervezés módszereit tervezési példákon keresztül mutatjuk be.

9.1. Igényes leíráshoz szükséges nyelvi elemek használata

9.1.1. Rácsrend bejelentése és indexek használata

Az adatáramlási leírás:

```
-- Rácsrend indexek
entity arrayIn2 is  port (
      a: in bit_vector(0 to 7);
      index: in integer range 0 to 7;
      output: out Bit);
end arrayIn2;

-- Belső szerkezet
architecture adataramlas of arrayIn2 is
begin
  output <=a(index);
end adataramlas;
```

A vizsgálati gerjesztés:

```
entity arrayIn2Stm is port (a: out bit_vector(0 to 7);
   index: out integer range 0 to 7; output:in bit);
end;

architecture adataramlas of arrayIn2Stm is
begin
  a <= "00000001" after   0 ns,
       "00010001" after  10 ns,
       "10110110" after  20 ns,
       "00100000" after 100 ns;
  index <= 0 after   0 ns,
           3 after  12 ns,
           1 after  25 ns,
           2 after  30 ns,
           3 after  35 ns,
           4 after  40 ns,
```

```
            5 after  45 ns,
            6 after  50 ns,
            7 after  55 ns,
            4 after 100 ns;
end;
```

9.1.2. Vektorszorzás

A következő leírás egy példa két vektor összeszorzására és a felültöltött * operátor használatára.

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

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

architecture rtl of vSzorz is
begin
  c<=a*b;
end;
```

9.1.3. Kimeneti jel újraolvasása

Amikor egy kimeneti jel értékét újra kell olvasni, három lehetőség kínálkozik:

- Az egyed bejelentésnél a jel ***buffer***ként való bejelentése
- Belső ***látszatos*** (dummy) jel használata az építményben
- A ***'driving_value***, jelre vonatkozó ***jelzőérték*** (attribute) alkalmazása

Az alábbiakban mind a háromra látható egy-egy példa:

```
library ieee;
use ieee.std_logic_1164.all;
entity peldaBuffer is
  port(clk,resetN,dIn1: in     std_logic;
                    q1: buffer std_logic;
                    q2: out    std_logic);
end;

architecture bhv of peldaBuffer is
begin
  process(clk,resetN)
  begin
    if resetN='0' then q1<='0'; q2<='0';
    elsif clk'event and clk='1' then
      q1<=dIn1;
      q2<=q1; -- A q1 jel olvasása
    end if;
  end process;
```

```
end;
```

Látszatos (dummy) jel használata:

```
library ieee;
use ieee.std_logic_1164.all;
entity peldaDummy is
  port(clk,resetN,dIn1: in  std_logic;
                   q1: out std_logic;
                   q2: out std_logic);
end;

architecture dtf of peldaDummy is
signal q1Dummy: std_logic;  -- A dummy jel
begin
  q1<=q1Dummy;
  process(clk,resetN)
  begin
    if resetN='0' then
      q1Dummy<='0';
      q2<='0';
    elsif clk'event and clk='1' then
      q1Dummy<=dIn1;
      q2<=q1Dummy; -- A q1Dummy jel olvasása
    end if;
  end process;
end;
```

A **'driving_value** használata:

```
library ieee;
use ieee.std_logic_1164.all;
entity peldaDriving is
  port(clk,resetN,dIn1: in  std_logic;
                   q1: out std_logic;
                   q2: out std_logic);
end;

architecture bhv of peldaDriving is
begin
  process(clk,resetN)
  begin
    if resetN='0' then q1<='0'; q2<='0';
    elsif clk'event and clk='1' then
      q1<=dIn1;
      q2<=q1'driving_value;
    end if;
  end process;
end;
```

A három példa közül az első használata gondot okozhat akkor, ha szerkezeti modellben egy összetevőt be akarnak ültetni. Ilyenkor szükség lenne egy **buffer** módú kapocs hozzárendelésére egy **out** módú kapocshoz, ami tilos. Ilyen helyzet akkor jön létre, ha az összetevő kapcsa **buffer** módú, s ezt kellene kivezetni az összetevőt tartalmazó egyed **out**

módú kapcsára. Ilyenkor nem jó megoldás a magasabb szintű egyed szóban forgó kapcsát is **buffer** módúnak bejelenteni, mert azon a szinten az már tiszta **out** módban működik. A **buffer** mód helyett az **inout** használata szintén nem jó megoldás, mivel ebben az esetben mások is tudnak az összetevő **inout** módú kapcsára írni. Az **inout** mód csak kétirányú jeleknél vagy huzalozott-logikánál (wired-or, stb.) használandó.

9.1.4. Fájl olvasása

Az alábbi leírás bemutat egy fájlt olvasó VHDL modellt.

```
library ieee;
use ieee.std_logic_1164.all; use ieee.std_logic_textio.all;
use std.textio.all;

entity olvaso is
  port (enable: in std_logic);
end;

architecture bhv of olvaso is
begin
  process(enable)
    type characterFile is file of character;
    file cFile: characterFile;
    variable c: character;
    variable charCount: integer:= 0;
  begin
    if enable = '1' then
      file_open(cFile, "proba.txt", read_mode);
      while not endFile(cFile) loop
        read (cFile, C);
        charCount:= charCount + 1;
      end loop;
      file_close(cFile);
    end if;
  end process;
end;
```

9.2. Nyalábolók és visszakódolók

9.2.1. Kettőből-egy nyaláboló

Bár többféle módon le lehet írni egy kettőből-egy ***nyalábolót*** (mulitplexor), de a **when else** összeállítás az ajánlott, mivel így egyetlen sorban elfér a VHDL kódban:

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

entity mux2 is
  port(a,b,sel: in std_logic; q: out std_logic);
end;
```

```
architecture rtl of mux2 is
begin
   c<=a when sel='0' else b;
end;
```

9.2.2. Négyből-egy nyalábolóló

Itt érdemes a **when else** mellett a **case** utasítást is használni, mert így áttekinthetőbb VHDL kód kapható (szintézis szempontjából nincs különbség):

```
library ieee;
use ieee.std_logic_1164.all;
entity mux4 is port(a: in std_logic_vector(3 downto 0);
        sel: in std_logic_vector(1 downto 0); q: out std_logic);
end;

architecture viselk of mux4 is
begin
  process(a,sel)
  begin
    case sel is     when   "00"   =>q<=a(3);
                    when   "01"   =>q<=a(2);
                    when   "10"   =>q<=a(1);
                    when others   =>q<=a(0);
   end case;
  end process;
end;
```

9.2.3. Háromból-nyolc visszakódoló

A következőkben egy háromból-nyolc ***visszakódoló*** (decoder) áramkör VHDL leírását láthatjuk.

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

entity decoder is port(a,b,c,g1,g2_n: in std_logic;
             y_n: out std_logic_vector(7 downto 0));
end;

architecture viselk of decoder is
begin
  process(a,b,c,g1,g2_n)
  begin
    y_n<=(others=>'1');
    if g1='1' and g2_n='0' then
      y_n(convInteger(c&b&a))<='0';
    end if;
  end process;
end;
```

9.3. Összeadók

9.3.1. Egész számokra használható összeadó

A következőkben egy egész számok összeadását végző tervezési egyedet és adatáramlási építményét láthatjuk.

```
entity sumArith is
  port (a,b: in integer range 0 to 15; c: out integer range 0 to 15);
end sumArith;

architecture adataramlas of sumArith is
begin
  c <= a + b;
end adataramlas;
```

A fenti egyedhez tartozó gerjesztés létrehozásának modelljét mutatja be a következő VHDL leírás.

```
entity sumArithStm is
 port (a,b: out integer range 0 to 15; c: in integer range 0 to 15);
end sumArithStm;

architecture adataramlas of sumArithStm is
begin
 a <=  0 after  0 ns, 10 after 50 ns;
 b <=  0 after  0 ns,   5 after 10 ns;
end adataramlas;
```

9.3.2. A félösszeadó szerkezeti leírása

A kétbemenetű éskapu viselkedési leírása:

```
entity and2 is port (a,b: in bit; y: out bit); end;

architecture viselk of and2 is
begin
  y <= a and b  after 1 ns;
end viselk;
```

A kétbemenetű kizáróvagy-kapu viselkedési leírása:

```
-- Kizáróvagy-kapu
entity xor2 is port (a: in bit; b: in bit; y: out bit); end;

architecture viselk of xor2 is
begin
  process(a, b)
  begin
    if a = b then
      y <= '0' after 1 ns;
    else y <= '1' after 1 ns;
    end if;
```

```
  end process;
end viselk;
```

A félösszeadó tárgybejelentése és szerkezeti leírása:

```
-- Félösszeadó szerkezeti modell
-- Külső kapcsok
entity halfAdd is port (x,y: in bit; sum,carry: out bit); end;

-- A félösszeadó belső szerkezeti leírása
architecture szerk of halfAdd is
  -- Felhasználásra kerülő összetevők
  component xor2 port (a,b:in bit; y:out bit); end component;
  component and2 port (a,b:in bit; y: out bit); end component;
begin
  X1:xor2 port map(x, y, sum);
  A1:and2 port map(x, y, carry);
end szerk;
```

A félösszeadó vizsgálati gerjesztése:

```
entity halfAddStm is port (x, y:out bit; sum, carry:in bit); end;

architecture dtf of halfAddStm is
begin
  x <= '0',
       '1' after  50 ns,
       '0' after 150 ns;
  y <= '0',
       '1' after 100 ns,
       '0' after 200 ns;
end dtf;
```

A félösszeadó próbapadja:

```
-- Próbapad a félösszeadóhoz
-- Ez a próbapad a halfAdd(szerk) tervet használja
entity halfAddBnc is end halfAddBnc;

architecture szerk of halfAddBnc is
  component halfAddStm port (x,y:out bit; sum,carry:in bit);
  end component;
  component halfAdd port (x,y:in bit; sum,carry:out bit);
  end component;
  signal x,y,sum,carry: bit;
begin
  Gerjesztes:halfAddStm port map(x,y,sum,carry);
  Aramkor:halfAdd port map(x,y,sum,carry);
end;
```

9.3.3. A teljes összeadó leírása

A kétbemenetű vagykapu tárgybejelentése és építménye:

```
entity or2 is port (a,b: in bit; y: out bit); end;
```

```
architecture adataramlas of or2 is
begin
  y <= a or b  after 1 ns;
end adataramlas;
```

A vagykapu vizsgálati gerjesztése:

```
entity or2Stm is port (a,b:out bit; y:in bit); end;

architecture adataramlas of or2Stm is
begin
  a <= '1' after 0 ns,
       '1' after 10 ns,
       '0' after 20 ns,
       '1' after 30 ns;
  b <= '0' after  0 ns,
       '1' after 10 ns,
       '1' after 20 ns,
       '1' after 30 ns;
end adataramlas;
```

A vagykapu próbapadjának leírása:

```
entity or2Bnc is end;

architecture szerk of or2Bnc is
  component or2Stm port (a,b:out bit; y:in bit); end component;
  component or2 port (a,b:in bit; y:out bit); end component;
  signal a,b,y: bit;
begin
  Aramkor:or2 port map(a,b,y);
  Gerjesztes:or2Stm port map(a,b,y);
end szerk;
```

A következő példa a teljes összeadó felépítését mutatja be két félösszeadóból és egy vagykapuból. A *tempSum* jel kapcsolja az első félösszeadó összeg kimenetét a második félösszeadó bemenetére. A *tempCarry1* jel az első félösszeadó átviteli kimenetét kapcsolja a vagykapu bemenetére; a *tempCarry2* jel a második félösszeadó átviteli kimenetét kapcsolja a vagykapu másik bemenetére. A teljes összeadó szerkezeti leírása:

```
-- külső kapcsok
entity fullAdd is
  port (a, b, carryIn: in bit; ab,carryOut: out bit);
end fullAdd;

-- belső szerkezet
architecture szerk of fullAdd is
  -- belső jelek bejelentése
  signal temp_sum, tempCarry1, tempCarry2: bit;
  -- felhasználásra kerülő osszetevők
  component halfAdd port (x,y: in bit; sum,carry: out bit);
  end component;
  component or2 port (a,b:in bit; y: out bit);
```

```
  end component;
begin
  -- részegységek beültetése
  C0: halfAdd port map (a,b,tempSum,tempCarry1);
  C1: halfAdd port map (tempSum, CarryIn, AB, TempCarry2);
  C2: or2 port map (TempCarry1, TempCarry2, CarryOut);
end szerk;
```

A teljes összeadó vizsgálati gerjesztése:

```
entity fullAddStm is
  port (A, B, CarryIn:out bit; AB, CarryOut:in bit);
end fullAddStm;

architecture adataramlas of fullAddStm is
begin
  A <= '0',
       '1' after  50 ns,
       '0' after 100 ns,
       '1' after 200 ns;
  B <= '1',
       '0' after  25 ns,
       '1' after  50 ns,
       '0' after  75 ns,
       '1' after 100 ns;
  CarryIn <= '0',
             '1' after  25 ns,
             '0' after  50 ns,
             '1' after  75 ns,
             '1' after 100 ns;
end adataramlas;
```

A teljes összeadó próbapadja, azaz vizsgáló áramköre:

```
entity fullAddBnc is end;

architecture szerk of fullAddBnc is
  component fullAddStm port (a,b,carryIn:out bit; ab,carryOut:in bit);
  end component;
  component fullAdd port (a,b,carryIn:in bit; ab,carryOut:out bit);
  end component;
  signal a,b,carryIn,ab,carryOut: bit;
begin
  Aramkor:fullAdd port map(a,b,carryIn,ab,carryOut);
  Gerjesztes:fullAddStm port map(a,b,carryIn,ab,carryOut);
end szerk;
```

A teljes összeadó szimulációs eredményeit a következő ábra mutatja be.

```
  ns a b  carryIn ab   carryOut
   0 0 0        0  0         0
   0 0 1        0  0         0
   2 0 1        0  1         0
  25 0 0        1  1         0
  26 0 0        1  0         0
```

```
  27 0 0         1  1          1
  28 0 0         1  1          0
  50 1 1         0  1          0
  51 1 1         0  0          0
  52 1 1         0  0          1
  75 1 0         1  0          1
  76 1 0         1  1          1
  77 1 0         1  0          0
  78 1 0         1  0          1
 100 0 1         1  0          1
 200 1 1         1  0          1
 202 1 1         1  1          1
```

9.3.4. Egybites összeadó átvitel bemenettel

Az alábbi példában az összeadás művelet eredményének bitszáma a legnagyobb bitszámú argumentumának bitszámával fog megegyezni

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

entity add1 is port(a,b,cIn: in std_logic; cOut,sum: out std_logic);
end;

architecture rtl of add1 is
  signal s: std_logic_vector(1 downto 0);
begin
  s<=('0' & a)+b+cIn;
  sum<=s(0);
  cOut<=s(1);
end;
```

9.3.5. Nyolcbites összeadó átvitel bemenettel

Az alábbi példa egy nyolcbites teljes összeadót mutat be.

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

entity add8 is port(a,b:std_logic_vector(7 downto 0);
                    cIn: in std_logic;
                    sum: out std_logic_vector(7 downto 0),
                   cOut: out std_logic);
end;

architecture rtl of add8 is
  signal s: std_logic_vector(8 downto 0);
begin
  s<=('0' & a)+b+cIn;
  sum<=s(7 downto 0);
  cOut<=s(8);
end;
```

9.3.6. Általános összeadó átvitel bemenettel

A következő VHDL kód két általános, *N* hosszúságú vektort ad össze. Az *N* értékét beültetésnél határozzák meg.

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

entity gAdd is
  generic (N:positive:=4);
  port(a,b: in std_logic_vector(N-1 downto 0);
       cIn: in std_logic;
       sum: out std_logic_vector(N-1 downto 0);
      cOut: out std_logic);
end;

architecture rtl of gAdd is
  signal s: std_logic_vector(N-1 downto 0);
begin
  s<=('0' & a)+b+cIn; sum<=s(N-1 downto 0);
  cOut<=s(N);
end;
```

9.3.7. Négybites összeadó/kivonó

Az alábbi négybites tervezési egyed összeadó és kivonó feladatkörrel rendelkezik.

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

entity addSub is
  port(addSubN:in std_logic;
            a,b:in std_logic_vector(3 downto 0);
              q:out std_logic_vector(4 downto 0));
end;

architecture rtl of addSub is
begin
  process(a,b,addSubN)
  begin
    if  addSubN='1' then q<=('0' & a)+b;
    else q<=('0' & a) -b;
    end if;
  end process;
end;
```

9.4. Példák folyamatokra

9.4.1. Nem teljesen meghatározott kombinációs folyamat

Egy kombinációs folyamatban (nem egy tároló elemben) minden kimeneti jelnek értéket kell adni, azaz nem szabad előfordulnia, hogy egy folyamat lefut úgy, hogy legalább egyszer nem adott értéket minden kimeneti jelnek. A következő példa bemutat egy nem teljes folyamatot:

```
architecture ROSSZ of pelda is
begin
  process(a,b)
  begin
    if a>b then a<='0';
    elsif a<b then a<='1';
    end if;
  end process;
end;
```

Ezt az építményt szintetizálva, a program egy latch-et fog elhelyezni a kimenetre. Az előző példa helyes leírása a következő:

```
architecture HELYES of pelda is
begin
  process(a,b)
  begin
    if a>b then a<='0';
    else a<='1';
    end if;
  end process;
end;
```

Ezt az építményt szintetizálva tiszta kombinációs hálózat lesz az eredmény.

9.4.2. Órázott folyamat

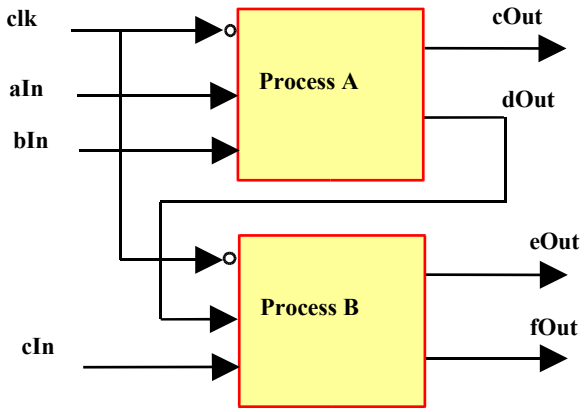
Az órázott folyamatok szinkron áramköröket modelleznek, és sok ilyen folyamat kapcsolódhat ugyanazon órához. Az alábbi sor azt biztosítja, hogy a folyamat nem kezdődhet el, amíg a *clk* órajel eső éle meg nem jelenik.

```
wait until clk='0';
```

A következő példában az *A* folyamat *dOut* kimeneti jele kapcsolódik a *B* folyamat bemenetére. Ezért az adatoknak stabilaknak kell lenniük, amikor az órajel elindítja a folyamatokat, és a következő értéknek ki kell alakulnia, mielőtt a folyamatok megint elindulnak.

Így a *dOut* jelnek stabilnak kell lennie mielőtt az órajel elindítja a folyamatokat. Az a leghosszabb idő, ami alatt a *B* folyamat az órajel általi elindítása után a *dOut* jelnek stabil értéket ad, meghatározza az órajel periódusának lehetséges legrövidebb értékét. A következő példában ez az idő 10 ns.

```
A: process
begin
  wait until clk='0';
  cOut<= not (aIn and bIn);
  dOut<= not bIn after 10 ns;
 end process;

B: process
begin
  wait until clk='0';
  eOut<= not (dOut and cIn);
  fOut<= not cIn;
 end process;
```



9.4-1. ábra. Órázott folyamat szintézisének eredménye.

A példa összetevő egyed bejelentése és építményének szerkezete látható az alábbiakban.

```
entity pelda is
port (clk,aIn,bIn,cIn: in std_logic;
  cOut,dOut,eOut,fOut: out std_logic);
end;

architecture viselk of pelda is
--  belsőjel bejelentés
...
begin
   A: process
   begin
      ...
   end process;
   B: process
   begin
      ...
   end process;
```

```
end;
```

Órajel előállítás

Ha a *clk* nem *bit* típusú, hanem pl. *std_logic*, akkor gondot jelent, hogy kezdőértékre állításnál (inicializálásnál) az *std_logic* az 'U' értéket fogja kapni. Az 'U' negáltja pedig szintén 'U'. Ez a gond a *clk*-nak a bejelentésekor '0' kezdő értékre való beállításával oldható meg:

```
signal clk:std_logic:='0';
```

9.4.3. Élvezérelt impulzuskeltő

Az ***élvezérelt impulzuskeltő*** (edge-controlled pulse generator) összetevő feladata impulzus létrehozása az órajelre, amikor a *dIn* bemenetnek emelkedő vagy eső éle van. Az emelkedő vagy eső él kiválasztását a *pozNegV* jel vezérli, ahogy azt a következő leírás bemutatja.

```
entity pulsG is
  port(dIn,pozNegV,clk,resetN: in std_logic; puls: out std_logic);
end;

architecture viselk of pulsG is
  signal dIn2: std_logic;
begin
  process(clk,resetN)
  begin
    if resetN='0' then
      dIn2<='0'; puls<='0';
    elsif clk'event and clk='1' then
      dIn2<=dIn;
      if dIn=pozNegV and dIn2= not pozNegV then
        puls<='1';
      else
        puls<='0';
      end if;
    end if;
  end process;
end;
```

9.4.4. Élészlelő

A következő, aszinkron törléses ***élészlelő*** (edge detector) észleli egy jel ***eső*** (falling) vagy ***emelkedő*** (rising), azaz ***felfutó*** élét és létrehoz egy impulzust, amely kitart az órajel egy időközében.

```
entity edgeDetector is
port (reset, clk, a: in std_logic; q: out std_logic);
end edgeDetector;

architecture viselk of edgeDetector is
begin -- Az építménytest
  process (clk, reset)
    variable edgeState : std_logic;
```

```
  begin -- a folyamattest
    if reset = '0' then -- aszinkron törlés (aktív alacsony)
      edgeState:= '0';
      q<= '0';
    elsif clk'event and clk = '1' then -- felfutó óra él
      if edgeState = A then
        q <= '0';
      else
        q <= '1';
      end if;
      edgeState := a;
    end if;
  end process;
end viselk;
```

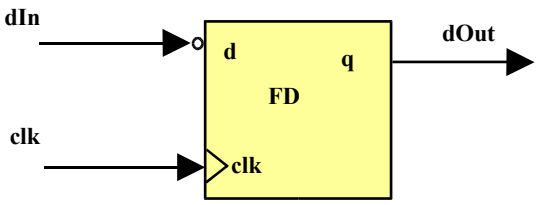
9.5. Flip-flop modellezése és szintézise

9.5.1. Flip-flop szintézis órázott jelekkel

Az órajeles folyamatok ahhoz vezetnek, hogy a folyamaton belüli hozzárendelt jelek flip-flopot eredményeznek a szintézis során. A következő példa azt mutatja be, hogy egy órázott folyamat hogyan valósítható meg a szintézis során. Az alábbi VHDL modell 1 ns alatt átviszi a *dIn*-t a *dOut*-ba. A szintézis után az átvitel egyenlővé válik a flip-flop előírásával.

```
Pelda: process
begin
  wait until clk='1';
  qOut<= qIn after 1 ns;
end process;
```

Az alábbi ábrán a szintetizált áramkör látható.



9.5-1. ábra. Flip-flop szintézis órázott jelekkel

9.5.2. Flip-flop változókkal modellezve

Az órázott folyamatbeli változók szintén vezethetnek flip-flophoz. Ha egy változót hamarabb olvasnak, mielőtt egy értéket rendeltek volna hozzá, ez egy flip-flophoz vezet. A következő példában a *count* változóval való szintézis három flip-flopot eredményez: egy a *q* jel,

kettő pedig a count változó miatt szükséges. Ennek az oka az, hogy a *count* változót a *count:=count+1;* értékadás során hamarabb olvassa a folyamat, mielőtt értéket adott volna neki.

```
process
  variable count: std_logic_vector (1 downto 0);
begin
  wait until clk='1';
  count:=count + 1;
  if count="11" then
    q<='1';
  else
    q<='0';
  end if;
end process;
```

9.5.3. Vizsgálható flip-flop szinkron engedélyezéssel

A következő két, hasonló példa szinkron engedélyezésű flip-flop szintézisére vezet.

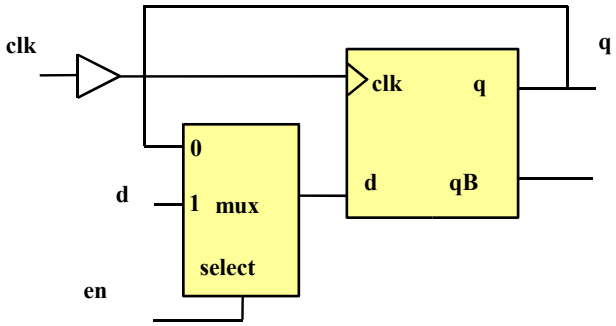
Ha egy jel nem kap értéket egy órázott folyamatban, a jel meg fogja tartani a korábbi értékét. A szintézis a jel visszacsatolását fogja eredményezni egy flip-flopból jövő kimeneti jelről saját magával való nyalábolón keresztül. Az ilyen terv kedvező a vizsgálhatóság szempontjából.

```
entity syncEnaFF is
  port (d, clk, en: in std_logic; q: out std_logic);
end;
architecture viselk of syncEnaFF is
begin
  process
  begin
    wait until clk='1';
    if en='1' then
      q<=d;
    end if;
  end process;
end;
```

Az alábbi leírás az előző feladatkörű modellt érzékenységi listás folyamattal írja le (csak az előzőtől eltérő rész, a folyamat kerül bemutatásra).

```
  process (clk)
  begin
    if clk'event and clk = '1' then -- felfutó órajel
      if en = '1' then
        q <= dat;
      end if;
    end if;
  end process;
```

Mindkét leírás a következő áramkör szintézisére vezet.



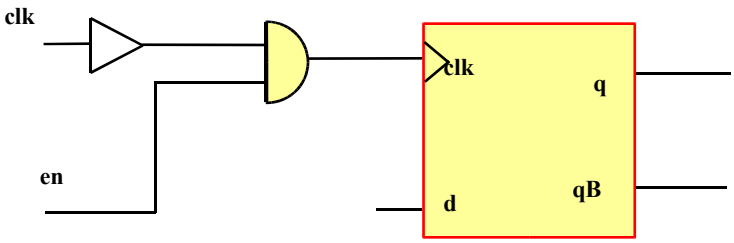
9.5-2. ábra. A vizsgálható, szinkron engedélyezésű flip-flop szintézise

9.5.4. Flip-flop szintézis kapuzott órával

A következő modellben az órajel kapuzva van az *en* jellel. Ez a terv nem jó a vizsgálhatóság szempontjából.

```
clk2<=clk and en;
Pelda2: process
begin
  wait until clk2='1';
  q<=d;
end process;
```

A következő ábra az előállított kapcsolást jeleníti meg.



9.5-3. ábra. Flip-flop szintézis kapuzott órával

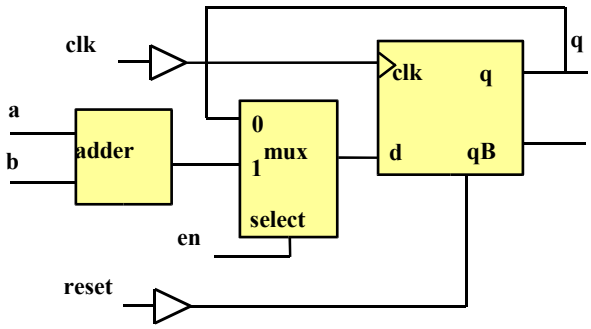
9.5.5. Összeadó és flip-flop szintézise

A következő példa bemutat egy összetettebb leírást.

```
Pelda: process (clk, reset)
begin
  if reset='1' then
    q<=(others=>'0');
```

```
  elsif clk'event and clk='1' then
    if en='1' then
      q<=a+b;
    end if;
  end if;
end process;
```

Az órázott folyamatban a jelhozzárendelés által okozott logika a flip-flop bal oldalán, a flip-flop bemenete előtt fog megjelenni a szintézis után, ahogy azt a következő ábra bemutatja.



9.5-4. ábra. Összeadó és flip-flop szintézise

9.5.6. Flip-flop regiszter aszinkron törléssel

Az alábbiakban egy aszinkron ***törlés***es (reset) flip-flop regisztert mutatunk be.

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

entity dMem is port(clk,resetN,dIn: in std_logic;
                                dOut: out std_logic);
end;

architecture rtl of dMem is
begin
  process(clk,resetN)
  begin
    if resetN='0' then dOut<='0';
    elsif clk'event and clk='1' then dOut<=dIn;
    end if;
  end process;
end;
```

9.5.7. Flip-flop regiszter szinkron törléssel

Az alábbiakban egy flip-flop regisztert kerül bemutatásra, amely szinkron ***törlés***sel (reset) rendelkezik.

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

entity dMem is port(clk,resetN,dIn: in std_logic;
                                dOut: out std_logic);
end;

architecture rtl of dMem is
begin
  process(clk)
  begin
    if clk'event and clk='1' then
      if resetN='0' then dOut<='0';
      else dOut<=dIn;
      end if;
    end if;
  end process;
end;
```

9.5.8. Flip-flop regiszter aszinkron törléssel és beállítással

Az alábbiakban egy flip-flop regisztert kerül bemutatásra, amelynek aszinkron ***törlés***e (reset) és ***beállítás***a (set) van.

```
library ieee;
use ieee.std_logic_1164.all;
entity dMem is port(clk,resetN,presetN,dIn: in std_logic;
                                        dOut: out std_logic);
end;

architecture rtl of dMem is
begin
  process(clk,resetN,presetN)
  begin
    if resetN='0' then dOut<='0';
    elsif presetN='0' then dOut<='1';
    elsif clk'event and clk='1' then dOut<=dIn;
    end if;
  end process;
end;
```

9.5.9. Nyolcbites regiszter engedélyezéssel és aszinkron törléssel

A következőkben egy engedélyezéssel és aszinkron törléssel rendelkező nyolcbites regiszter kerül ismertetésre.

```
library ieee;
use ieee.std_logic_1164.all;
entity dMem is port(clk,resetN,en: in std_logic;
                               dIn: in std_logic_vector(7 downto 0);
```

```
                              dOut: out std_logic_vector(7 downto 0));
end;
architecture rtl of dMem is
begin
  process(clk,resetN)
  begin
    if resetN='0' then dOut<=(others=>'0');
    elsif clk'event and clk='1' then
      if en='1' then dOut<= dIn;
      end if;
    end if;
  end process;
end;
```

9.6. Léptető egységek

9.6.1. Léptetés műveletek

A) VHDL szabványban meghatározott léptetés műveletek

Léptetés műveletből 6 különböző van a VHDL szabványban, melyeket a következő táblázat bemutat.

Művelet jele	Magyar név	Angol név	Magyarázat
sll	léptetés balra	shift left	jobb oldali utolsó bit a '0'-t veszi fel
srl	léptetés jobbra	shift right	bal oldali első bit a '0'-t veszi fel
rol	átforgatás balra	roll over left	
ror	átforgatás jobbra	roll over right	
sla	léptetés balra, megtartva a jobboldali végértéket	shift left, and keep value 'right	
sra	léptetés jobbra, megtartva a baloldali végértéket	shift right, and keep value 'left	

9.6-1. táblázat. A VHDL szabványban rögzített léptetés műveletek

Az alábbi VHDL leírás bemutatja a fenti műveletek használatát.

```
architecture viselk of pelda is
begin
  a <= "01101";
  q1 <= a sll 1;     -- q1 = "11010"
  q2 <= a srl 3;     -- q2 = "00001"
  q3 <= a rol 2;     -- q3 = "10101"
  q4 <= a ror 1;     -- q4 = "10110"
  q5 <= a sla 2;     -- q5 = "10111"
  q6 <= a sra 1;     -- q6 = "00110"
```

```
end;
```

B) Léptetés műveletek csomagbeli függvényekkel

Ha a szintézis eszköz nem támogatja a szabványbeli léptetés műveletet, akkor csomagban meghatározott függvénnyel lehet megvalósítani. Pl. az *ieee* könyvtárbeli *std_logic_unsigned* csomagban a következő függvények találhatók:

```
function shl (arg: std_logic_vector; count: std_logic_vector)
return std_logic_vector;
function shr (arg: std_logic_vector; count: std_logic_vector)
return std_logic_vector;
```

Egy alkalmazási példa:

```
q1 <= shl(data,'1');              -- Egyet léptet balra
q2 <= shr(data,"101");            -- Ötöt léptet jobbra
q3 <= shr(data, count);           -- Count számút léptet jobbra
```

C) Léptetés művelet elkerülése

Készíthető VHDL kód léptetési művelet alkalmazása nélkül is, pl.:

```
architecture rtl of pelda is;
signal data: std_logic_vector(7 downto 0);
begin
  process(clk, resetN)
  begin
    if resetN='0' then q1<=(others=>'1'); q2<=(others=>'1');
    elsif clk'event and clk='1' then
      q1(6 downto 0)<=q1(7 downto 1);   -- Egyet léptet jobbra
      q1(7)<=dIn;
      q2(7 downto 1)<=q2(6 downto 0);   -- Egyet léptet balra
      q2(0)<=dIn;
    end if;
  end process;
end;
```

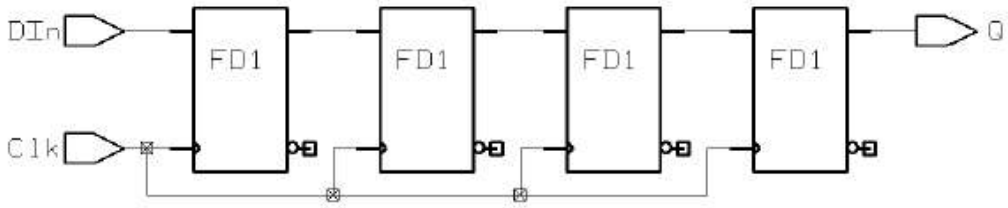
9.6.2. Egyszerű léptető regiszter

Az alábbiakban egy egyszerű léptető regiszter VHDL modellje látható.

```
entity shiftRegister is
  port (dIn, clk : in std_logic; q: out std_logic);
end;
architecture viselk of shiftRegister is
begin
  process (clk, dIn)
    variable dTmp : std_logic_vector(3 downto 0);
  begin
    if clk'event and clk = '1' then -- felfutó órajel
      dTmp := dTmp(2 downto 0) & dIn;
      q <= dTmp(3);
    end if;
```

```
  end process;
end;
```

A következő ábra bemutatja a ***léptető*** (shift) ***regiszter*** szintetizált alakját.



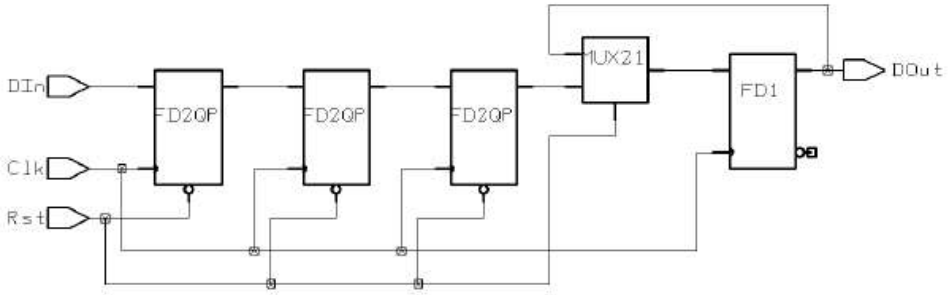
9.6-1. ábra. A szintetizált léptető regiszter.

9.6.3. Léptető regiszter aszinkron törléssel

A következő VHDL leírás egy összefűzés művelettel létrehozott, aszinkron törléssel ellátott léptető regisztert mutat be.

```
entity pelda is
  generic (nBits : integer := 4);
  port (dIn, clk, rst: in std_logic; dOut: out std_logic);
end;
architecture viselk of pelda is
  begin
  process (clk, rst)
    variable reg: std_logic_vector(nBits-1 downto 0);
  begin
    if rst = '0' then -- aszinkron törlés (aktív alacsony szint)
      reg:= (others => '0');
    elsif clk'event and clk = '1' then -- felfutó órajel
      reg:= reg(nBits-2 downto 0) & dIn;
      dOut <= reg(nBits-1);
    end if;
  end process;
end;
```

A következő ábrán a léptető regiszter szintetizált alakja látható, az áramkörhöz aszinkron törlés tartozik.



9.6-2. ábra. Az aszinkron törléses szintetizált léptető regiszter.

9.7. Számlálók

9.7.1. Egy 2-bites számláló

Ebben a részben egy példát látunk egy kétbites számláló VHDL leírására. A leírást a *count2* egyed bejelentésével kezdjük, meghatározva a külső határfelületét, amely magában foglalja a kapcsainak megadását. Így a számláló egyedet a következőképpen írhatjuk le:

```
entity count2 is
  generic (propDelay : time := 10 ns);
  port (clock: in bit; q1, q0 : out bit);
end;
```

Ez meghatározza, hogy a *count2* egyed két bemenettel és egy kimenettel rendelkezik, mindegyike bit értékű, azaz, '0' vagy '1' értéket vehetnek fel. Szintén meghatároz egy *propDelay* nevű általános állandót is, amelyet az egyed működésének (ebben az esetben a késleltetési idő) szabályozására használnak. Ha kifejezetten nem adnak meg értéket, amikor ezt az egyedet egy tervben használják, a 10 ns alapértelmezési érték kerül felhasználásra.

Az egyed egy megvalósítását egy építménytestben írják le. Ugyanazon egyed bejelentéshez több építménytest is tartozhat, melyek az egyed különböző elvonatkoztatási szintű modelljeit képviselik. Pl. a számláló egy viselkedési leírása a következő lehet:

```
architecture viselk of count2 is
begin
  CountUp: process (clock)
    variable countValue : natural := 0;
  begin
    if clock = '1' then
      count_value := (countValue + 1) mod 4;
      q0 <= bit'val(countValue mod 2) after propDelay;
      q1 <= bit'val(countValue / 2) after propDelay;
    end if;
  end process countUp;
end;
```

Ebben számláló ebben a leírásában a viselkedést egy *countUp* folyamattal valósítjuk meg, amely érzékeny a *clock* bemenetre. Ennek a folyamatnak van egy *countValue* nevű változója a számláló mindenkori állapotának tárolására. A változó a szimuláció kezdetén zérusra van beállítva, és a folyamat két tevékenyítése között megtartja az értékét. Amikor a *clock* bemenet változik '0'-ról '1'-re, az állapot változó eggyel megnövekszik, és a kimeneteken az új értéken alapuló jelváltoztatás ütemezve jelentkezik. A hozzárendelések használják a *propDelay* általános állandót az órajel változása után a jelváltoztatás ütemezése idejének meghatározására. Amikor a szimulációs szabályozás eléri a folyamattest végét, a folyamat felfüggesztődik, amíg újabb változás nem jelenik meg a *clock* jelen.

A kétbites számlálót két *T-flipflop*ból és egy inverterből álló áramkörként is le lehet írni. VHDL nyelven ez a következőképpen modellezhető:

```
architecture szerk of count2 is
  component tFlipFlop port (ck : in bit; q : out bit); end component;
  component inverter port (a : in bit; y : out bit); end component;
  signal ff0, ff1, inv_ff0 : bit;
begin
  Bit0: tFlipFlop port map (ck => clock, q => ff0);
  Inv: inverter port map (a => ff0, y => inv_ff0);
  Bit1: tFlipFlop port map (ck => inv_ff0, q => ff1);
  q0 <= ff0;
  q1 <= ff1;
end szerk;
```

Ebben az építményben kétféle összetevő van az elején bejelentve: *tFlipFlop* és *inverter*. Ezeken kívül az építmény bejelentési részében szerepel három belső jel bejelentése is. Az összetevők mindegyikére van később hivatkozás és a hivatkozások kapcsai hozzá vannak rendelve az egyed belső jeleihez és a kapcsaihoz. Pl., *bit0* a *tFlipFlop* összetevő egy hivatkozása, amelynek a *ck* kapcsa a *count2* egyed *clock* kapcsához van kötve, és a *q* kapocs pedig az *ff0* belső jelhez kapcsolódik. Az utolsó két jelhozzárendelés frissíti az egyed kapcsait amikor csak a belső jelek változnak. Erre azért van szükség, mert a kimenetnek bejelentett *q0* és *q1* jeleket nem lehet olvasni, csak írni.

9.7.2. Egy 8 bites számláló leírása

A következő leírás jó példa arra, hogy szigorúan kötött az egyes jelek típus bejelentése. Az *intCnt* jelre csak azért van szükség, mert a *cnt* **out**-nak, kimeneti jelnek van bejelentve, így annak nem adhatnánk értékeket.

```
--   8 bites számláló adatáramlási leírása
entity counter8 is
port (clk, load, rst: in bit;
    data: in bit_vector(7 downto 0);
    cnt: out bit_vector(7 downto 0));
end counter8;
architecture adataramlas of counter8 is
  signal intCnt: bit_vector(7 downto 0);
begin
  intCnt <= "00000000" when Rst='0'
       else data when clk='1' and (not clk'stable) and Load='1'
```

```
      else intCnt+'1' after 1 ns when clk='1' and not clk'stable
      else IntCnt after 2 ns;
  cnt <= intCnt after 2 ns;
end;
```

A következő leírásban található vizsgáló gerjesztés a számláló lehetséges működési állapotainak csak egy részét vizsgálja meg. Lehetne úgy is módosítani a vizsgáló gerjesztést, hogy a *preset* tulajdonságot is ellenőrizhessük (tehát azt, hogy egy adott értéktől indul a számolás).

```
-- A counter8(adataramlas) tervre vonatkozó próbapad
use std.textio.all;
entity counterBnc is end counterBnc;

architecture szerk of counterBnc is
  -- jel bejelentések
  signal clk, load, rst: bit;
  signal data,cnt: bit_vector(7 downto 0);
   -- összetevő bejelentések és a kialakítás leírása
  component counter8
    port (clk, load, rst: in bit;
                  data: in bit_vector(7 downto 0);
                   cnt: out bit_vector(7 downto 0));
  end component;
  for all: counter8 use entity work.counter8(adataramlas);
begin
  --osszetevo utasitasok
  -- beultetesek, kapcsolodasi lista
  H1: counter8 port map (clk, load, rst, data, cnt);

  -- Vizsgálati gerjesztés
  rst <= 'X' after 0 ns,
         '0' after 2 ns,
         '1' after 10 ns;
  load <= '0' after 0 ns;
  data <= "00000000" after 0 ns;
  clk  <=   'X' after 0 ns,
            '0' after 1 ns,
            '1' after 6 ns,
            '0' after 11 ns,
            '1' after 16 ns,
            '0' after 21 ns,
            '1' after 26 ns,
            '0' after 31 ns,
            'X' after 32 ns;

  -- Eredmény kijelzés fejléce
  FejlecFolyamat: process
    variable L: line;
  begin
    write (L, "Ido  clk  Load   Rst     Data    Cnt");
    writeline (output, L);
    wait;
```

```
  end process;

  -- Eredmények kijelzése
  FigyeloFolyamat: process (clk, Load, Rst, Data, Cnt)
    variable dline: line;
  begin
    write (dline, NOW, right, 2);
    write (dline, clk, right, 5);
    write (dline, Load, right, 3);
    write (dline, Rst, right, 6);
    write (dline, Data, right, 15);
    write (dline, Cnt, right, 15);
    writeline (output, dline);
  end process;
end szerk;
```

9.7.3. Hárombites számláló engedélyezéssel és átvitel kimentettel

Ha az *ieee.std_logic_unsigned* csomagot használjuk, elegendő a „+” vagy „-” függvényt használni, amikor egy számláló értéket növelni vagy csökkenteni kell. Ha a számláló bemenete vektorként van bejelentve, a számláló önműködően fog váltani, amikor az összes bit elérte az '1'-t. Ha a számlálónak meg kell állnia egy határnál, akkor szükség van egy korlát ellenőrzésre. A következők két építmény közül az elsőben (*viselk1*) nincs korlát ellenőrzés, a másodikban (*viselk2*) van. Az áramkörszintézis mind a kettőnél hasonló eredményt ad, mivel az **if** utasítás, amely ellenőrzi a korlátot, nem ad új feladatkört. Dokumentációs szempontból a másodikat könnyebb olvasni, de több sorból áll.

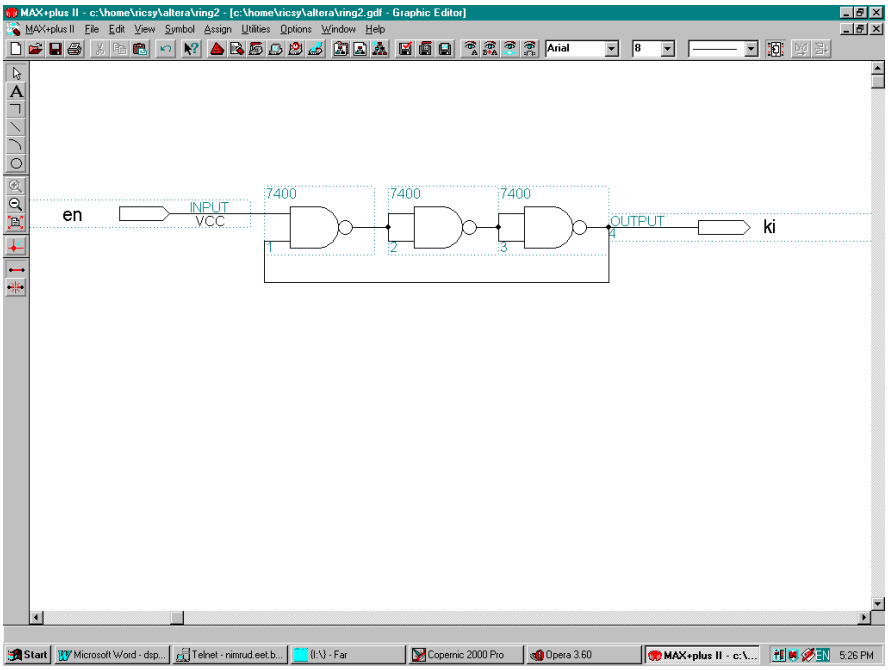
Ha a számláló egésznek van bejelentve, mindig kell korlát ellenőrzés, egyébként hiba lép fel, amikor a *count*=7 és a „+1” kerül végrehajtásra. A következőkben bemutatásra kerül a számláló modellje.

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

entity pelda is
  port(clk,resetN,countEn: in std_logic;
                      sum: out std_logic_vector(2 downto 0);
                     cOut: out std_logic);
end;

architecture viselk1 of pelda is
  signal count:std_logic_vector(2 downto 0);
begin
  process(clk,resetN)
  begin
    if resetN='0' then count<=(others=>'0');
    elsif clk'event and clk='1' then
      if countEn='1' then
        count<=count+1;
      end if;
```

```
    end if;
  end process;
  sum<=count;
  cOut<='1' when count=7 and countEn='1' else '0';
end;

architecture viselk2 of pelda is
  signal count:std_logic_vector(2 downto 0);
begin
  process(clk,resetN)
  begin
    if resetN='0' then count<=(others=>'0');
    elsif clk'event and clk='1' then
      if countEn='1' then
        if count/=7 then
          count<=count+1;
        else
          count<=(others=>'0');
        end if;
      end if;
    end if;
  end process;
  sum<=count;
  cOut<='1' when count=7 and countEn='1' else '0';
end;
```

9.8. Gyűrűs oszcillátor

Az alábbiakban látható egy 4 inverterből és egy NAND kapuból álló ***gyűrűs*** (ring) oszcillátor kapcsolási rajza.



9.8-1. ábra. A grafikusan megtervezett gyűrűs oszcillátor

Az előbbi ábrán látható gyűrűs oszcillátort alkotó inverter egyed bejelentése és modellje látható az alábbiakban.

```
library ieee; use ieee.std_logic_1164.all;
entity myNand is
       port(a, b: in std_logic; c: out std_logic);
end myNand;

library ieee; use ieee.std_logic_1164.all;
architecture viselk of myNand is
begin
  process(a, b)
  begin
       c <= not(a and b) after 50 ns;
       c <= a;
  end process;
end;
```

A gyűrűs oszcillátor szerkezetei modellje látható az alábbi VHDL leírásban.

```
library ieee; use ieee.std_logic_1164.all;
entity myRing is port(en: in std_logic; ki: out std_logic);
end;

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

```
architecture szerk of myRing is
  signal sa, sb, sc : std_logic;
  component myNand
    port(a, b : in std_logic; c : out std_logic);
  end component;
begin
  N1:myNand port map(a => en, b => sa, c => sb);
  N2:myNand port map(a => sb, b => sb, c => sc);
  N3:myNand port map(a => sc, b => sc, c => sa);
  ki <= sa;
end;
```

9.9. Állapotgépes modellezés

Az ***állapotgép***et (state machine) a CPU-val összehasonlítva megállapítható, hogy az állapotgépek használatával hatékonyan lehet szabályozási feladatokat megvalósítani. A Neumann-féle szerkezetben (CPU), pl. olyan műveletek kellenek, mint az ***elhozási*** (fetch) és ***végrehajtási*** ütem, adat útvonalak, ALU regiszterek, stb. Egy állapotgép teljesítménye sokkal jobb, mint egy CPU-é, mivel a viselkedést leíró kód a Bool-algebrát megvalósító kapuhálózatban jelenik meg, az állapot pedig flip-flopokban van tárolva. A következő kód megvalósítható CPU-ban vagy VHDL nyelven leírt állapotgépben:

```
if a>37 and c<7 then
  state <= alarm; outA <= '0'; outB <= '0'; outAnalog <= a+b;
else
  state <= running;
end if;
```

Ha a kódot egy CPU-ban valósítják meg, akkor kb. 10-20 gépi utasítást igényel. A végrehajtása különböző számú assembler utasítást igényel attól függően, hogy az **if** utasítás melyik útvonalat választja.

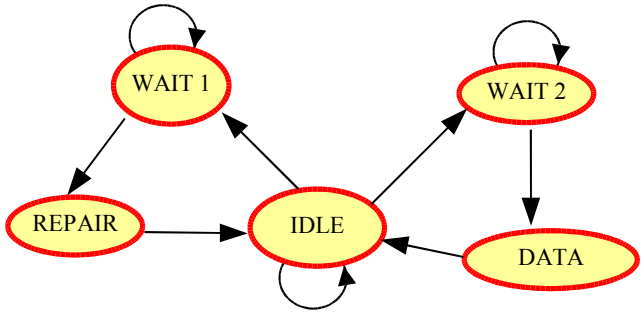
A CPU-nál tehát nem határozható meg egy pontos végrehajtási idő, helyette minimum és maximum végrehajtási időtartamot kell használni. Ha ugyanezt a VHDL kódot kapukkal és flip-flopokkal valósítják meg, akkor a végrehajtás egy órajel ciklusig tart. Következésképpen az állapotgépes megvalósítás teljesítménye és időbeli meghatározottsága lényegesen jobb, mint a CPU-belié.

Az állapotgép két ütemben működik, az első ütemben az új állapotot kiszámítja, a második ütemben pedig az új állapotot betölti egy regiszterbe. A következő állapot kiszámításának időigénye határozza meg az állapotgép sebességét. Minden egyes órajelnél frissítik a regisztereket, majd elkezdődik a következő állapot kiszámítása, aminek be kell fejeződnie, mielőtt a következő órajel megérkezik, hogy amikor a regisztereket frissíteni kell, a belső jelek ismét stabil állapotban legyenek.

9.9.1. Az állapotgépek típusai

Két alapvető típusa van, a ***Mealy*** és a ***Moore gép***. A ***Mealy gép*** kimenete a jelenlegi állapotának és az összes bemenet függvénye, mindig egy órajel ciklus alatt működik, és azonnal változtatja a kimeneteit, amint változnak a bemenetek. A ***Moore gép*** kimenete csak a jelenlegi

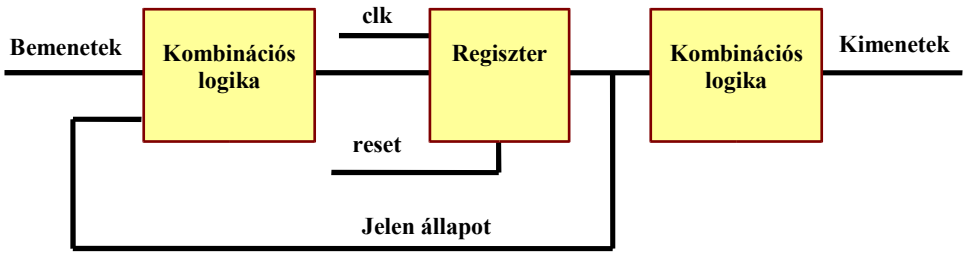
állapotának függvénye és először az állapotát kell megváltoztatnia, azaz várnia egy órajel ciklust, mielőtt a kimeneti értékei megváltoznak. A következő ábra egy Mealy gép állapotábráját mutatja be.



9.9-1. ábra. Egy Mealy gép állapotábrája

9.9.2. A Moore gép egyszerű modellje

A Moore gép tömbvázlata:



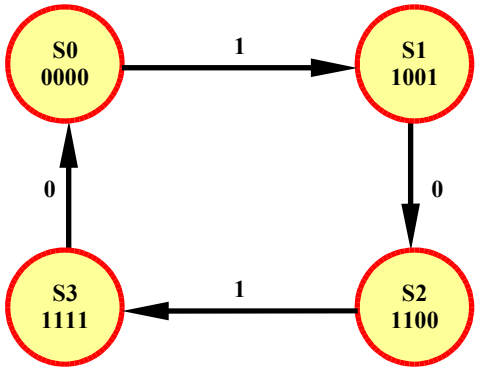
9.9-2. ábra. A Moore gép tömbvázlata

Példa: legyen négy állapot: *S0*, *S1*, *S2* és *S3*, a hozzájuk tartozó kimeneti értékek rendre: "0000", "1001", "1100" és "1111", a VHDL kód:

```
entity pelda is port(clk,in1,reset: in std_logic;
                     out1: out std_logic);
end pelda;

architecture moore of pelda is
  type stateType is (s0,s1,s2,s3);      -- Állapot bejelentés
  signal state: stateType;
begin
  DemoF: process(clk,reset)       -- Órázott folyamat
```

```
  begin
    if reset='1' then state<=s0;  -- Állapot törlése
    elsif clk'event and clk='1' then
      case state is when s0=> if in1='1' then state<=s1; end if;
                    when s1=> if in1='0' then state<=s2; end if;
                    when s2=> if in1='1' then state<=s3; end if;
                    when s3=> if in1='0' then state<=s0; end if;
      end case;
    end if;
  end process;

  KimenetiF: process(state)       -- Kombinációs folyamat
  begin
    case state is  when s0=> out1<="0000";
                   when s1=> out1<="1001";
                   when s2=> out1<="1100";
                   when s3=> out1<="1111";
    end case;
  end process;
end moore;
```

Ahogy a fenti leírásból is látszik, az állapotgép három részből áll, amelyek a ***bejelentés*** (declaration), az ***órázott folyamat*** (clocked process) és a ***kombinációs folyamat*** (combinational process). A Moore gép állapotábrája látható a következő ábrán.



9.9-3. ábra. A Moore gép állapotábrája

A) Bejelentés

A ***bejelentés*** (declaration) az állapotot jelenti be. Bármilyen név választható az állapot számára, amely felsorolt típusú. Egy célszerű választás:

```
type stateType is (startState, runState, errorState);
signal state: stateType;
```

B) Órázott folyamat

Az órázott folyamat dönti el, hogy mikor kell az állapotgépnek állapotot váltania. Ezt a folyamatot az állapotgép órája tevékenyíti. A jelen állapottól és a bemeneti jelek állapotától függően az állapotgép minden felfutó órajel változásra változtathatja az állapotot.

C) Kombinációs folyamat

A kombinációs folyamat a jelen állapottól függően rendeli hozzá a kimeneti jelekhez az értékeket. Az előző példában látható, hogy a *KimenetiF* folyamat érzékenységi listájában csak az állapot vektorok vannak felsorolva, ami megfelel a Moore gép kimeneti jeleire vonatkozó meghatározásnak.

9.9.3. Moore gép modellje három folyamattal

Az előző, általános VHDL modellel szemben van olyan modell, amely három folyamatból áll, melyek feladata a következő:

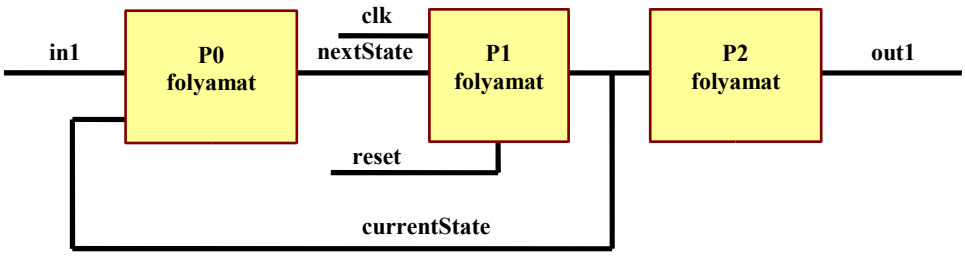
- egy a következő állapotot (*nextState*) dekódolja;
- egy hozzárendeli az állapotvektort a jelen állapothoz (*currentState*);
- egy a kimeneti jelek értékeit állítja be.

Az így kialakított modell pontosan a Moore gép tömbvázlatának felel meg. Némileg hosszabb VHDL kódra vezet és kicsit lassabb a szimulációja, de dokumentációs szempontból előnyösebb. A későbbi szintézis szempontjából nincs különbség a két modell között:

```
entity pelda is port(clk,in1,reset: in std_logic;
                     out1: out std_logic);
end pelda;

architecture moore of pelda is
  type stateType is (s0,s1,s2,s3);      -- Állapot bejelentés
  signal state, currentState, nextState: stateType;
begin
  P0: process(currentState,in1)         -- Kombinációs folyamat
  begin
    case currentState is
                when s0=> if in1='1' then nextState<=s1; end if;
                when s1=> if in1='0' then nextState<=s2; end if;
                when s2=> if in1='1' then nextState<=s3; end if;
                when s3=> if in1='0' then nextState<=s0; end if;
    end case;
  end if;
end process;

  P1: process(clk,reset)          -- Órázott folyamat
    begin
      if reset='1' then currentState<=s0;      -- Állapot törlése
      elsif clk'event and clk='1' then currentState<=nextState;
      end if;
  end process;
```

```
  P2: process(currentState)              -- Kombinációs folyamat
    begin
      case currentState is when s0=> out1<="0000";
                           when s1=> out1<="1001";
                           when s2=> out1<="1100";
                           when s3=> out1<="1111";
      end case;
  end process;
end moore;
```

A következő ábra a Moore gép három folyamattal létrehozott modelljének tömbvázlatát mutatja be.



9.9-4. ábra. A Moore gép tömbvázlata három folyamattal

9.10. RAM és ROM építése

Mind RAM, mind ROM beépíthető ASIC vagy FPGA/EPLD tervbe. A következőkben a szükséges VHDL leírás megtervezéséről lesz szó. Két mód van ROM meghatározására VHDL-ben: tömb állandó használata vagy technológiára jellemző ROM beültetése.

9.10.1.ROM meghatározása tömb állandóval

Ezzel a módszerrel a ROM meghatározása akkor terület-hatékony, ha a ROM viszonylag kicsi. Ha a ROM nagy, akkor a terület alapköltsége sokkal nagyobb, mint egy technológiára jellemző ROM beültetésénél. A tömb állandóval való ROM bejelentést érdemes a csomagba helyezni, mert így újrafelhasználhatóvá válik. Példa egy 4x8 bites ROM-ra:

```
library ieee;
use ieee.std_logic_1164.all;
package rom is
  constant romWidth:integer:=8; constant romLength:integer:=4;
  subtype romWord is std_logic_vector (romWidth-1 downto 0);
  type romTable is array (0 to romLength-1) of romWord;
  constant rom: romTable:= romTable'("00101111", "11010000",
                                     "01101010","11101101");
end;
```

Az előbb meghatározott ROM alkalmazása:

```
library ieee;
use ieee.std_logic_1164.all; use ieee.std_logic_unsigned.all;
use work.rom.all;

entity romCir1 is
  port(addr:in std_logic_vector(1 downto 0); q:out romWord);
end;

architecture dtf of romCir1 is
begin
  q<=rom(convInteger(addr));
  -- Adatot olvas a ROM tömb addr nevű címéről
end;
```

A szintézis során kombinációs logikai hálózattal fog létrejönni a ROM, mivel a felhasznált ROM mező címét előzetesen nem lehet kiszámítani. Ha a cím számítható, akkor az előző példabeli kimeneti *q* jel állandó, így a szintézis során nem kombinációs hálózattal jön létre a ROM, hanem az adott ROM mezőhöz tartozó egyes biteket közvetlenül a V_{CC} tápfeszültségre vagy a földre kötik. A kiszámolható ROM cím egy példája:

```
architecture dtf of romCir2 is
begin
  q<=rom(2);  -- Adatot olvas a ROM tömb 2-es címéről
end;
```

Technológiára jellemző ROM beültetése nagyobb ROM-ok esetén célszerű. Előnye a jobb területkihasználtság, mivel az adott technológiára optimalizált, hátrányai, hogy a VHDL kód technológiafüggővé válik és gondot jelent a szimuláció, ha nincs VHDL modell a ROM-ra. Ilyenkor esetleg egy kevert módú (VHDL- és kapu-szintű) szimulátort lehet alkalmazni.

9.10.2.RAM létrehozása

Két lehetőség van RAM feladatkör megvalósítására, az egyik regiszterek használatával történik, a másik pedig RAM beültetésével.

A) RAM meghatározása regiszterekkel

A regiszterek nagyon kis RAM-okhoz használhatók, nagyobb RAM-ok esetén a területi alapköltség túl nagy. A következő példa egy órázott folyamat használatával modellezi a regisztereket:

```
process(clk,resetN)
begin
  if resetN='0' then
      q<=(others=>'0');
  elsif clk'event and clk='1' then
    if wr='1' then
        q<=data;
    end if;
  end if;
end process;
```

B) RAM létrehozása beültetéssel

Nem lehetséges jó szintézis eredményt elérni RAM beültetése nélkül. Hátrány (hasonlóan a ROM-hoz), hogy a VHDL kód technológiafüggővé válik. A szintézis során a RAM-ot egy írható-olvasható tömbbel helyettesítik. A RAM beültetéséhez a *generate* parancs alkalmazható, pl. a következő módon:

```
architecture rtl of RAM4 is
  component RAM4x1 port(d,a0,a1,we: in std_logic; q:out std_logic);
  end component;
  -- A következő sor csak a szimulációhoz szükséges
  for U1:RAM4x1 use entity work.RAM4x1(rtl);
begin
  for i in 0 to 3 generate
     RAM_b: RAM4x1 port map (dIn(i),a0,a1,write,dOut(i));
  end generate;
end;
```

Tekintsük a következő példát: meg kell tervezni egy *c1* összetevőt, amely beültet egy 4x8 bites RAM-ot. A könyvtár csak a következő RAM-ot tartalmazza:

```
entity RAM4x1 is
  port(d,a0,a1,we: in std_logic; q: out std_logic);
end;
```

A megoldást a következő modell jelentheti:

```
library ieee; use ieee.std_logic_1164.all;
entity c1 is
  port(write, a0, a1: in std_logic;
                 dIn: in std_logic_vector(7 downto 0);
                dOut: out std_logic_vector(7 downto 0));
end;

architecture rtl of RAM4 is
  component RAM4x1  port(d,a0,a1,we: in std_logic; q: out std_logic);
  end component;
  -- A következő sor csak a szimulációhoz szükséges
  for U1:RAM4x1 use entity work.RAM4x1(rtl);
begin
  for i in 0 to 7 generate
       RAM_block: RAM4x1 port map(dIn(i),a0,a1,write,dOut(i));
  end generate;
end;
```

9.11. Egy számtani-logikai egység terve

A ***számtani-logikai egység*** (**A**rithmetic-**L**ogic **U**nit, **ALU**) tervezési egyede és viselkedési modellje:

```
library ieee; use ieee.std_logic_1164.all;
entity alu is
  port (a, b: in std_logic_vector(1 downto 0);
```

```
      opcode: in std_logic_vector(2 downto 0);
         en: in std_logic;
        outp: out std_logic_vector(1 downto 0));
end alu;

architecture bhv of alu is
  signal outBelso: std_logic_vector(1 downto 0);
begin
  Engedelyezo: process(outBelso,en)
  begin
    if (en = '1') then outp <= outBelso;
    else outp <= "00";
    end if;
  end process;

  Szamitas: process(opcode,a,b)
  begin
    if    (opcode = "111") then outBelso <= a and  b;
    elsif (opcode = "011") then outBelso <= a or   b;
    elsif (opcode = "001") then outBelso <= a xor  b;
    elsif (opcode = "010") then outBelso <= a nor  b;
    elsif (opcode = "101") then outBelso <= a nand b;
    else  outBelso <= "00";
    end if;
  end process;

end bhv;
```

Az ALU gerjesztését létrehozó modell tervezési egyede és *adatáramlási* szintű modellje:

```
library ieee; use ieee.std_logic_1164.all; use std.textio.all;
entity aluStm is
  port (a, b: out std_logic_vector(1 downto 0);
      opcode: out std_logic_vector(2 downto 0);
          en: out std_logic;
        outp: in std_logic_vector(1 downto 0));
end aluStm;

architecture dtf of aluStm is
  signal opcodeTemp: std_logic_vector(2 downto 0);
begin
  en <= '1' after 0 ns;
  a  <= "00" after 0 ns, "11" after 60 ns;
  b  <= "00" after 0 ns, "10" after 5 ns;
  opcodeTemp <= "000" after 10 ns, "111" after 15 ns,
                "011" after 20 ns, "001" after 25 ns;
  opcode <= opcodeTemp;

  Muveletek: process (opcodeTemp)
    variable L: line;
  begin
   if    (opcodeTemp="111") then write (L, string'("and kapcsolat"));
   elsif (opcodeTemp="011") then write (L, string'("or kapcsolat"));
```

```
   elsif (opcodeTemp="001") then write (L, string'("xor kapcsolat"));
   elsif (opcodeTemp="010") then write (L, string'("nor kapcsolat"));
   elsif (opcodeTemp="101") then write (L, string'("nand kapcsolat"));
   else  write (L, string'("nincs muvelet"));
  end if
end process;

end dtf;
```

9.12. Szintézisre optimalizált tervezés erőforrás megosztással

A fejlett szintézis eszközök magas szintű optimalizációt végeznek. Ez többek között azt jelenti, hogy a szintézis eszköz megoszthatja az erőforrásokat, pl. egy összeadót különböző vektor összeadások között. Ehhez megfelelőnek kell lenni a VHDL tervnek. Az erőforrás megosztás feltétele az, hogy a különböző vektorokat soha nem használhatja az összeadó egyszerre. A következő példában két összeadás van ugyanabban a folyamatban, de mivel egy **if-then-else** utasításban vannak, soha nem hajtandók végre egyidejűleg. Így a szintézis eszköz erőforrás-megosztással optimalizálhat. A következő példa egy lehetséges erőforrás-megosztást mutat be:

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

entity share is
  port(a,b,c,d: in std_logic_vector(4 downto 0);
            sel: in std_logic;
              q: out std_logic_vector(4 downto 0));
end;

architecture viselk of share is
begin
  process(a,b,c,d,sel)
  begin
    if sel='0' then q<=a+b;
    else  q<=c+d;
    end if;
  end process;
end;
```

A következő példa esetén az összeadó erőforrás megosztása nem lehetséges.

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

entity share is
       port(a,b,c,d: in std_logic_vector(4 downto 0);
              q1,q2: out std_logic_vector(4 downto 0));
end;

architecture rtl of share is
```

```
begin
   q1<=a+b; q2<=c+d;
end;
```

Ebben az esetben a két összeadó használatát nem lehet elkerülni kombinációs hálózat alkalmazásával.

10. Irodalomjegyzék

[1] S. Sjoholm, L. Lindh: VHDL for designers, *Prentice Hall*, London, 1997.

[2] Synopsys honlap, *http://www.synopsys.com*

[3] Mentor Graphics honlap, *http://www.mentor.com*

[4] A. A. Jerraya, H. Ding, P. Kission, M. Rahmouni: Behavioral synthesis and component reuse with VHDL, *Kluwer Academic Publishers*, Boston, 1997.

[5] IEEE Standard VHDL Language Reference Manual (IEEE Std 1076-1987). New York, *Institute of Electrical and Electronics Engineers, Inc.*, 1988.

[6] IEEE Standard VHDL Language Reference Manual (IEEE Std 1076-1993). New York, *Institute of Electrical and Electronics Engineers, Inc.*, 1994.

11. Függelék

Az alábbiakban *work* könyvtárbeli *oraKapuk* nevű felhasználói csomag forráskódja látható.

```
--!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--                    Az oraKapuk csomag
--!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

package oraKapuk is

  component clk       -- Óragenerátor
    port (y:out bit; reset:in bit);
  end component;

  component and2
    port (ki: out bit; be1: in bit; be2: in bit);
  end component;

  component and3
    port (ki: out bit; be1: in bit; be2: in bit; be3: in bit);
  end component;

  component nand2
    port (ki: out bit; be1: in bit; be2: in bit);
  end component;

  component nand3
    port (ki: out bit; be1: in bit; be2: in bit;
          be3: in bit);
  end component;

  component or2
    port (ki: out bit; be1: in bit; be2: in bit);
  end component;

  component nor2
    port (ki: out bit; be1: in bit; be2: in bit);
  end component;

  component xor2
    port (ki: out bit; be1: in bit; be2: in bit);
  end component;

  component inv
    port (ki: out bit; be: in bit);
  end component;
```

```
end oraKapuk;

--!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--              Egyed bejelentések
--!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

entity clk is
  port (y:out bit; reset:in bit);
end clk;

entity and2 is
  port (ki: out bit; be1: in bit; be2: in bit);
end and2;

entity and3 is
  port (ki: out bit; be1: in bit; be2: in bit; be3: in bit);
end and3;

entity nand2 is
  port (ki: out bit; be1: in bit; be2: in bit);
end nand2;

entity nand3 is
  port (ki: out bit; be1: in bit; be2: in bit; be3: in bit);
end nand3;

entity or2 is
  port (ki: out bit; be1: in bit; be2: in bit);
end or2;

entity nor2 is
  port (ki: out bit; be1: in bit; be2: in bit);
end nor2;

entity xor2 is
  port (ki: out bit; be1: in bit; be2: in bit);
end xor2;

entity inv is
  port (ki: out bit; be: in bit);
end INV;

--!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--   Az oraKapuk csomagbeli elemek építménye
--!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

architecture viselk of clk is
  signal belso: bit;
begin
  process
  begin
    if reset='1' then
      if belso='1' then
```

```
        belso <='0';
      else belso <='1';
      end if;
    else belso <='1';
    end if;
    y <= belso;
    wait for 2 ns;
  end process;
end viselk;

architecture viselk of and2 is
begin
  process (be1, be2)
    variable aktualisEredmeny: bit;
  begin
    aktualisEredmeny := be1 and be2;
    ki <= aktualisEredmeny after 2 ns;
  end process;
end viselk;

architecture viselk of and3 is
begin
  process (be1, be2, be3)
    variable aktualisEredmeny: bit;
  begin
    aktualisEredmeny := be1 and be2 and be3;
    ki <= aktualisEredmeny after 2 ns;
  end process;
end viselk;

architecture viselk of nand2 is
begin
  process (be1, be2)
    variable aktualisEredmeny: bit;
  begin
    aktualisEredmeny := be1 nand be2;
    ki <= aktualisEredmeny after 2 ns;
  end process;
end viselk;

architecture viselk of nand3 is
begin
  process (be1, be2, be3)
    variable aktualisEredmeny: bit;
  begin
    aktualisEredmeny := not(be1 and be2 and be3);
    ki <= aktualisEredmeny after 2 ns;
  end process;
end viselk;

architecture viselk of or2 is
begin
```

```
  process (be1, be2)
    variable aktualisEredmeny: bit;
  begin
    aktualisEredmeny := be1 or be2;
    ki <= aktualisEredmeny after 2 ns;
  end process;
end viselk;

architecture viselk of nor2 is
begin
  process (be1, be2)
    variable aktualisEredmeny: bit;
  begin
    aktualisEredmeny := be1 nor be2;
    ki <= aktualisEredmeny after 2 ns;
  end process;
end viselk;

architecture viselk of xor2 is
begin
  process (be1, be2)
    variable aktualisEredmeny: bit;
  begin
    aktualisEredmeny := be1 xor be2;
    ki <= aktualisEredmeny after 2 ns;
  end process;
end viselk;

architecture viselk of inv is
begin
  process (be)
    variable ertek: bit;
  begin
    ertek := not(be);
    ki <= ertek after 2 ns;
  end process;
end viselk;
```