VHDL

VHSIC HARDWARE DESCRIPTION LANGUAGE

SEGÉDLET
(kézirat)

Készítették: dr. Horváth Tamás
Harangozó Gábor

BME Irányítástechnika és Informatika Tanszék

Bevezetés

A VHDL (VHSIC Hardware Description Language) egy olyan általános hardver leíró nyelv, amely elsõsorban digitális áramkörök különbözõ absztrakciós szinteken történõ, egységes leírására alkalmas. A nyelv elsõ változata az Amerikai Egyesült Államok kormányának ösztönzésére 1981-ben jelent meg. A VHDL 1987 decembere óta nemzetközi szabvány (IEEE–1076), ami nagy mértékben hozzájárul egyre szélesebb körû használatához. Kiemelkedõ tulajdonsága, hogy technológia-független, így nem kötõdik szorosan szimulátorhoz, és a tervezõ számára is rugalmas felhasználást tesz lehetõvé. A tervezõ maga választhatja ki a tervezési módszert (pl. top-down vagy könyvtáralapú tervezés, stb.) és a tervezési technológiát (pl. szinkron vagy aszinkron áramkör, PLA vagy random logikai áramkör, stb.). A VHDL az áramköri technológiától is független, az áramköri technológia (pl. CMOS, nMOS, GaAs, stb.) kiválasztása szintén a tervezõ feladata.

Egy tetszõleges digitális áramkör VHDL nyelven leírt modelljét három egymástól független – mégis szétválaszthatatlan – modell alkotja: a *viselkedési modell*, a *struktúrális modell* és az *idõzítési modell*.

A *viselkedési modell* egy rendszer vagy részrendszer funkcionális leírását tartalmazza. A viselkedési modell alapját a folyamatok (process-ek) alkotják, amelyek között az információt speciális globális változók, az ún. jelek (signal-ok) közvetítik. A folyamatok önmagukban szekvenciális, de egymáshoz képest konkurrens programrészek. Egy folyamat mûködése bármikor felfüggeszthetõ, akár valamilyen feltétel bekövetkezéséig, akár végtelen hosszú ideig. A VHDL tehát konkurrens programozási nyelv, ezért kifejezetten alkalmas igen nagy komplexitású digitális rendszerek viselkedésének leírására.

A *struktúrális modell* a rendszer hierarchikus felépítését, az egyes részrendszerek kapcsolódási struktúráját írja le. A részrendszerek közötti kapcsolatot – a viselkedési modellhez hasonlóan – itt is speciális globális változók, az ún. portok teremtik meg. A portok definiálása egyben irányított jelek definiálását is jelenti. A portok között az információáramlás általában egyirányú: az *out* porttól az *in* port felé történik. Speciális esetekben (*inout* portok) az információáramlás a portok között kétirányú is lehet.
Az *idõzítési modell* azon alapul, hogy a modellezett rendszer a bemeneteire adott gerjesztésre meghatározott módon reagál, majd a gerjesztés további változásaira várakozik. A szimuláció során az egyes események idõpontjai mindig szimulációs idõpontok, tehát függetlenek a fizikai idõtõl. A szimulátor diszkrét idõegységekben lépkedve dolgozza fel a szimulációs idõpontokra ütemezett eseményeket, amelyeket egyrészt a gerjesztés, másrészt a modellezett rendszer gerjesztésre adott válasza határoz meg. Esemény alatt egy jel értékének megváltozását értjük. Egy adott szimulációs idõpontban az események feldolgozása egy vagy több ún. szimulációs ciklusban történik. A szimulációs ciklusnak két fázisa van. Az elsõ fázisban azoknak a jeleknek az értéke változik meg, amelyek változása az adott szimulációs idõpontra van elõírva. A második fázisban lefutnak azok a folyamatok, amelyeket az elsõ fázisban megváltozott jelek aktivizáltak. A folyamatok újabb jelek értékváltozásait írhatják elõ, így elõfordulhat, hogy az adott szimulációs idõpontban további szimulációs ciklusokat is végre kell hajtani. Egy szimulációs idõpontban több szimulációs ciklus egymás utáni végrehajtásakor gondoskodni kell arról, hogy a ciklusok ne zérus idõ alatt fussanak le, mert különben a jelváltozások között nem biztosítható az ok-okozati viszony. Ezért minden szimulációs ciklushoz egy elméleti *delta* késleltetést rendelük, ami a szimulációs idõt nem befolyásolja. Egy adott szimulációs idõpontban a szimulációs ciklusok addig ismétlõdnek, amíg a folyamatokon végig nem gyûrûzik a kezdeti jelváltozások hatása. Ha az adott szimulációs idõpontra nincs több jelváltozás elõírva, akkor a szimulátor a következõ szimulációs idõpontra ütmezett eseményeket kezdi el feldolgozni.

Adatszerkezetek

Lexikai elemek

Commentek

A VHDL nyelvben a commentek két egymást követõ kötõjellel ("--") kezdõdnek, amelyek hatása a sor végéig tart.

Pl: -- Ez egy comment sor

Azonosítók

Az azonosítók a VHDL által lefoglalt kulcsszavak valamint a programozó által definiált nevek. Az azonosítók betûkbõl és számokból állhatnak, de mindig betûvel kell kezdõdniük.

azonosító ::= betû { [aláhúzás] betû_vagy_szám }

A szögletes zárójel az opcionális elemeket jelöli, a kapcsos zárójel pedig a megismételhetõket.

Az azonosítók tartalmazhatnak aláhúzás jelet ("_") is. A kis- és nagybetûk között a nyelv nem tesz különbséget.

Pl: This_Name, this_name -- a két azonosító megegyezik

Számok

A számokat a 2 és a 16 között bármilyen számrendszerben felírhatjuk. Ha egy szám tizedes pontot tartalmaz, akkor az valós számkot jelent, egyébként egész számként van értelmezve. Tízes számrendszerben a számok formátuma a következõ:

decimális_szám ::= egész_szám [. egész_szám] [kitevõ]
egész_szám ::= számjegy { [aláhúzás] számjegy }
kitevõ ::= E [+] egész_szám | E - egész_szám

Pl: 0 1 123_456_789 987E6 -- egész számok
0.0 1.5 2.718_28 12.4E-9 -- valós számok

Nem tízes számrendszerben a számok formátuma a következõ:

based_literal ::= base # based_integer [. based_integer] # [exponent]
base ::= integer
based_integer ::= extended_digit { [underline] extended_digit }
extended_digit ::= digit | letter

A számrendszer alapját és a kitevõt decimális formában kell megadni.

Pl: 2#1100_0100# 16#C4# 4#301#E1
2#1.1111_1111_111#E+11 16#F.FF#E2

Karakterek

Karakternek számít az ASCII táblázat valamennyi eleme. A kerektereket rövid idézõjelek közé kell tenni.

Pl: 'a' 'A' '35' ' '

Stringek

A stringek karaktersorozatok, amelyek hossszú idézõjelek között állnak.

Pl: "A string" "A stringben egy újabb ""string""."

Bitvektorok

A bitvektor bit tipusú változókból (0-kból és 1-ekbõl) képzett sorozat. A bitvektort megadhatjuk bináris (B), oktális (O) és hexadecimális (X) szám formájában. A bitvektorok szintaktikája a következõ:

bitvektor ::= alap " számérték "
alap ::= B | O | X
számérték ::= számjegy { [aláhúzás] számjegy }

Pl: B"1010110" O"126" X"56"

Adattípusok

A VHDL-ben az adattípusokat két nagy csoportra oszthatjuk; elemi (vagy skalár) és összetett típusokra. Az elemi típusokhoz tartoznak az egész (Integer) és lebegõpontos (Real) számok, a fizikai mennyiségek, a felsorolás típus és még néhány elõre definiált standard elemi adattípus. Az összetett típust a tömbök és a rekordok alkotják. A két nagy csoporton kívül a VHDL-ben megtalálható még a mutató típus és a file típus is. Az elõredefiniált elemi adattípusok értékkészlete a következõ:

```
-- egész típus:
type Integer is range  -2147483648  to  2147483648 ;

-- lebegõpontos típus (legfeljebb hat tizedes pontossággal) :
type Real is range  -16#0.7FFFFFF8#+32  to  16#0.7FFFFFF8#+32 ;

-- felsorolás típusok:
type Boolean is (False, True) ;
type Bit is ('0', '1') ;
type Severity_level is (Note, Warning, Error, Failure) ;
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);

-- fizikai típus
type Time is range  -(2**31-1)  to  (2**31-1)
  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 ;
```

A tömbök lehetnek egy- és többdimenziósak. A tömb elemszámát nem kötelezõ magadni, ilyenkor a tömbnek tetszõleges számú eleme lehet. Tömb típust a következõképpen deklarálhatunk:

```
tömb_típus_definíció ::=
        korlátlan_tömb_definíció | korlátozott_tömb_definíció
```

korlátlan_tömb_definíció ::=
array (index_definíció { , index_definíció }) **of** elemtípus_megjelölés

korlátozott_tömb_definíció ::=
array (index_korlátozás) **of** elemtípus_megjelölés

index_definíció ::= típusjelzés **range** <>
index_korlátozás ::= (diszkrét_tartomány { , diszkrét_tartomány })
diszkrét_tartomány ::= diszkrét_típus_megjelölés | tartomány
tartomány ::= kifejezés irány kifejezés
irány ::= **to** | **downto**

Pl: **type** word **is array** (31 **downto** 0) **of** Bit;
type memory **is array** (1 **to** 4, 1 **to** 4) **of** Real;
type register_bank **is array** (byte **range** 0 **to** 132) **of** Integer;

type vector **is array** (integer **range** <>) **of** Real;

A VHDL-ben a következõ két tömb elõredefiniált adattípus:

type String **is array** (positive **range** <>) **of** Character;
type Bit_vector **is array** (natural **range** <>) **of** Bit;

A *positive* és a *natural* típus az *Integer* típusból származtatott altípus.

Egy tömb elemeire a tömb neve után írt indexeléssel hivatkozhatunk. Egydimenziós tömbök esetén arra is lehetõség van, hogy ne csak egy elemre, hanem egy összefüggõ tartományra hivatkozzunk. Ebben az esetben az indexben a kiválasztott tartományt kell megadni.

Pl: a (1) ; b (1,1) ; c (2 **to** 5) ; d (10 **downto** 6) ;

Értékadáskor egy tömb elemeit megadhatjuk elemenkénti hozzárendeléssel, de úgy is, hogy a tömb elemeit poziciójukkal együtt felsoroljuk. Ez utóbbi esetben a megegyezõ elemeket nem kell külön pozicionálni, mivel az others kulcsszó hatására a maradék poziciókhoz hozzárendelhetõk az azonos elemek.

Pl: **type** a **is array** (1 **to** 4) **of** Character;
a(1) := 'f';
a(2) := 'o';
a(3) := 'o';
a(4) := 'd';

type a **is array** (1 **to** 4) **of** Character;
a := ('f', 4 => 'd', **others** => 'o');

A rekordok különbözõ típusú adatokból összeállított tömbök. Egy rekord deklarációja a következõ szintaktika szerint történik:

```
rekord_típus_azonosító ::=
        record
                elem_deklaráció
                {elem_deklaráció}
        end record ;

elem_deklaráció ::= aznosító_lista  :  elemtípus_definíció ;
azonosító_lista ::= azonosító { ,  azonosító }
elemtípus_definíció ::= elemtípus_megjelölés
```

Pl:
```
type instruction is
        record
                op_code : processor_op;
                address_mode : mode;
                operand1, operand2 : integer range 0 to 15;
        end record;
```

A rekord egy mezõjére úgy hivatkozhatunk, hogy ponttal elválasztva a rekord azonosítója után írjuk a mezõ azonosítóját.

Pl:
```
instruction.op_code    -- az istruction rekord op_code mezõje
```

Altípusok

Az altípusok használata lehetõvé teszi, hogy az elemi típusok értékkészletét a kívánt részhalmazra szûkítsük le. Az altípusok deklarációja a következõ szintaktika szerint történik:

```
altípus_deklaráció ::=
        subtype  azonsító  is  altípus_megjelölés ;
altípus_megjelölés ::= [ resolution_function_név ]  típus_jelzés  [ korlátozás ]
típus_jelzés ::= típus_név  |  altípus_név
megszorítás ::= tartomány_korlátozás  |  index_korlátozás
```

Az altípusoknak két fajtája van. Egyes altípusok egy skalár típus értékkészletének a leszûkítését jelentik, más altípusok viszont egy korlátlan tömb elemszámának a specifikálását.

Pl:
```
subtype digits is Character range '0' to '9' ;
subtype word is Bit_vector (31 downto 0);
```

A következõ két altípus elõredefiniált adattípus:

```
subtype natural is integer range 0 to highest_integer
subtype positive is integer range 1 to highest_integer
```

Konstansok, változók és jelek deklarációja

A konstansok olyan objektumok, amelyek értéke a kezdeti értékadás után nem módosítható. Konstansok deklarálása az alábbi szintaktika szerint történik:

konstans_deklaráció ::=
constant azonosító_lista : típus_megjelölés [:= kifejezés] ;

A kezdeti értékadás el is maradhat, ilyenkor vagy az adott programban késõbb, vagy egy másik programban kap értéket a konstans.

Pl: **constant** e : Real := 2.71828;
constant delay : Time := 5 ns;

A változók olyan objektumok, amelyek a kezdeti értékadás után megváltoztathatják értéküket. Változók deklarációját a következõ szintaktika írja le:

változó_deklaráció ::=
variable azonosító_lista : típus_megjelölés [:= kifejezés] ;

Ha egy változónak kezdeti értéket adunk, akkor az azonnal kiértékelõdik. Ha nem adunk neki kezdeti értéket, akkor a kezdeti érték egy default érték lesz, ami általában az adott típus legkisebb értéke, tahát felsorolás típusnál a lista elsõ eleme, növekvõ sorrendbe rendezett tartományok esetén az értékkészlet legkisebb eleme, csökkenõ sorrendbe rendezett tartományok esetén pedig az értékkészlet legnagyobb eleme. Ha a változó összetett adattípusú, akkor annak default értéke az egyes elemek default értékébõl tevõdik össze.

Pl: **variable** count : natural := 0;
variable instuction : Bit_vector (31 **to** 0);

A jelek olyan objektumok, amelyek változóként viselkednek, de szerepük az egyes programrészek közötti információtovábbításban van. Jeleket a következõ szintaktika szerint deklarálhatunk:

jel_deklaráció ::=
signal azonosító_lista : típus_megjelölés [:= kifejezés] ;

Pl: **signal** data : Bit_vector (15 downto 0);

A VHDL-ben lehetõség van arra is, hogy egy objektumnak, vagy egy objektum egy részének alternatív nevet adjunk. Ennek szintaktikája a következõ:

alias_deklaráció ::=
alias azonosító : típus_megjelölés **is** objektum_név

Pl: **variable** instuction : Bit_vector (31 **to** 0);
alias op_code : Bit_vector (7 **downto** 0) **is** instruction (31 **downto** 24);

A példaként bemutatott változó deklaráció után az *instruction* változó felsõ nyolc bitjére *op_code* néven is hivatkozhatunk.

Attribútumok

Az attribútum egy olyan névvel ellátott jellemzõ, amely a jellemzett nyelvi eszközrõl (adattípusról, objektumról, eljárásról, függvényrõl, strukturális egységrõl, stb.) kiegészítõ információt ad. Az attribútumot aposztróffal választjuk el attól a nyelvi elemtõl, amelyikre vonatkozik.

```
Pl:   A'high      -- az A skalár típusú változó értékkészletének
                  -- legnagyobb eleme
```

A VHDL nyelvben elõredefiniált attribútumok a Függelékben találhatók. Lehetõség van arra is, hogy a felhasználó maga definiáljon attribútumokat (User-Defined Attributes).

Kifejezések és operátorok

A VHDL nyelvben a kifejezések – akárcsak más programozási nyelvekben – elemi egységekbõl (objektumokból, függvényhívásokból, zárójeles kifejezésekbõl) és operátorkból állnak. A VHDL-ben elõredefiniált operátorokat az 1. táblázat tartalmazza.

Az összeadás és a kivonás két operandusa bármilyen megegyezõ numerikus (egész, lebegõpontos, fizikai) típusú lehet. Szorzásnál és osztásnál mindkét operandusnak vagy egész vagy lebegõpontos számnak kell lennie. A modulus és a maradékképzés operandusai csak egész számok lehetnek. Exponenciális kifejezés alapja bármilyen egész vagy lebegõpontos szám lehet, a kitevõ viszont csak egész szám lehet. Ha a kitevõ negatív egész, akkor az alap kizárólag lebegõpontos szám lehet. Az egyoperandusú aritmetikai operátorok után bármilyen numerikus típusú kifejezés állhat.

A relációs mûveletek mindig Boolean típusú eredményt szolgáltatnak. Az '=' és a '/=' jel bármilyen megegyezõ típusú kifejezés között állhat, a file típust kivéve. Az összes többi relációs operátor csak skalár típusú számok, egydimenziós egészekbõl álló tömbök, és felsorolás típusú tömbök között értelmezhetõ.

A logikai operátorok operandusai kizárólag Bit vagy Boolean típusú kifejezések lehetnek. A logikai mûveletek az operandusok típusával megegyezõ típusú eredményt szolgáltatnak. A logikai kifejezések kiértékelése balról jobbra történik, és csak addig tart, amíg biztosan el nem dõl az eredmény logikai értéke.

Az összefûzõ (concatenation) operátor két egydimenziós tömböt fûz össze egy egydimenziós tömbbé. Ezt az operátort tipikusan stringmûveletek esetén használjuk.

Csoport	Szimbólum	Funkció
aritmetikai (kétoperandusú)	+	összeadás
	–	kivonás
	*	szorzás
	/	osztás
	mod	modulus
	rem	maradékképzés
	**	hatványozás
aritmetikai (egyoperandusú)	+	plusz elõjel
	–	mínusz elõjel
	abs	abszolút érték
relációs	=	egyenlõ
	/=	nem egyenlõ
	<	kisebb
	>	nagyobb
	<=	kisebb vagy egyenlõ
	>=	nagyobb vagy egyenlõ
logikai (kétoperandusú)	**and**	logikai ÉS
	or	logikai VAGY
	nand	logikai NEM ÉS
	nor	logikai NEM VAGY
	xor	logikai kizáró VAGY
logikai (egyoperandusú)	**not**	logikai negálás
összefûzõ	**&**	összefûzés

1. táblázat

Az operátorok prioritását csökkenõ sorrendben a 2. táblázat mutatja.

legnagyobb prioritás: **	**abs**	**not**				
	*	/	**mod**	**rem**		
	+ (elõjel)		– (elõjel)			
	+	–	**&**			
	=	/=	<	<=	>	>=
legkisebb prioritás:	**and**	**or**	**nand**	**nor**	**xor**	

2. táblázat

Viselkedési leírás

Szekvenciális vezérlési szerkezetek

Értékadás változónak

Egy változónak új értéket hozzárendeléssel adhatunk. Az értékadás szintaktikája:

változó_értékadás ::=
változó := kifejezés ;

A változónak és a kifejezésnek elemi típusúnak kell lennie.

Pl: a := 1; alfa := 'a' ;

Feltételes utasítások

A feltételes utasításoknak két változata van: az **if** szerkezet és a **case** szerkezet. Az **if** szerkezet szintaktikája a következõ:

```
if_szerkezet ::=
        if feltétel then
                utasítás_sorozat
        {elsif feltétel then
                utasítás_sorozat}
        [else
                utasítás_sorozat]
        end if ;
```

A feltétel részekben szereplõ kifejezéseknek Boolean típusúnak kell lenniük. Ha egy feltétel igaz, akkor a feltétel után álló valamennyi utasítás végrehajtódik, ellenkezõ esetben a vezérlés a következõ feltételre ugrik.

Pl:
```
variable literal Character;
...
if literal = '0' then
        value := 0;
elsif (literal = '1' or literal = '2) then
        value := 1;
elsif (literal > = '3' and literal <= '9') then
        value := 1;
else
        value := 2;
end if;
```

A case szerkezet szintaktikája a következõ:

```
case_szerkezet ::=
        case  kifejezés  is
                case_alternatíva
                {case_alternatíva}
        end  case ;

case_alternatíva ::=
        when  választás =>
                utasítás_sorozat ;

választás ::= eset { | eset}
eset ::= egyszerû_kifejezés | diszkrét_tartomány | others
```

A **case** után álló kifejezés csak diszkrét elemi típus (egész szám, felsorolás típus, fizikai típus) vagy egydimenziós karaktertömb lehet. Az alternatívák közül mindig az hajtódik végre, amelyikben a *választás* értéke megegyezik a **case** utáni kifejezés értékével. Fontos, hogy minden választásnak különbözõnek kell lennie, tehát egyetlen *eset* sem szerepelhet kétszer. Nem kell viszont a **case** mögötti kifejezés valamennyi értékére alternatívát állítani, ugyanis az alternatívákban fel nem sorolt eseteket összefoghatjuk az **others** kulcsszóval, és valamennyihez ugyanazt az utasítássorozatot rendelhetjük hozzá.

Pl:
```
case literal is
        when  '0'               =>      value := 0 ;
        when  '1' | '2'  =>     value := 1 ;
        when  '3' to '9'        =>      value := 1 ;
        when  others =>         value := 2 ;
end case ;
```

Ciklusok

A VHDL-ben háromféle ciklust szervezhetünk. A **loop** ciklus feltétel nélküli hurkot eredményez, a **while** ciklus törzse addig ismétlõdik, amíg a feltétel részben szereplõ kifejezés igaz, a **for** ciklus pedig elõre megadott számú iterációt hajt végre. A **loop** ciklus szintaktikája:

```
loop_ciklus ::=
        [ címke  : ]
        loop
                utasítás_sorozat
        end  loop  [ címke ] ;
```

Pl:
```
add_loop:
loop
        a := a + 1 ;
end  loop  add_loop ;
```

A **while** ciklus szintaktikája:

```
while_ciklus ::=
        [ címke  : ]
        while  feltétel  loop
                utasítás_sorozat
        end  loop  [ címke ] ;
```

A feltétel részben szereplõ kifejezés az utasítássorozat végrehajtása elõtt mindig kiértéklõdik. Ha a feltétel hamissá válik, az iteráció véget ér, és kilépünk a ciklusból.

Pl:
```
sub_loop:
while (a /= 0) or (a /= 1) loop
        a := a - 2;
end  loop  sub_loop ;
```

A **for** ciklus szintaktikája:

```
for_ciklus ::=
        [ címke  : ]
        for  ciklusváltozó  in  diszkrét_tartomány
                utasítás_sorozat
        end  loop  [ címke ] ;
```

A ciklusváltozó a **for** ciklus törzsében mindig konstansként viselkedik, tehát az értékét nem változtathatjuk meg.

Pl:
```
exp_loop:
for  exp  in  1 to 4  loop
        base := base * base ;
end  loop  exp_loop ;
```

A VHDL-ben létezik két olyan speciális utasítás, amely lehetõvé teszi a ciklusokból való kilépést. Az egyik a **next** utasítás, amelynek hatásásra a vezérlés a ciklus elejére ugrik, a másik az **exit** utasítás, amely az iterációt leállítja, és a vezérlést a ciklust követõ elsõ utasításra irányítja. A két speciális utasítás szintaktikája a következõ:

```
next_utasítás ::=
        next  [ címke ]  [ when  feltétel ] ;

exit_utasítás ::=
        exit  [ címke ]  [ when  feltétel ] ;
```

Ha nem adunk meg címkét, akkor az aktuális ciklusból lépünk ki, egyébként a címkével jelzett ciklusból. A ciklusból történõ kilépés feltételhez is köthetõ; ha a feltétel rész igaz, akkor az utasítás végrehajtódik. Az **exit** uatsításnak elsõsorban a **loop** ciklusok esetén van jelentõsége, ugyanis használatával kiküszöbölhetjük a végtelen ciklusokat.

Pl:
```
load :
for  i  in  1 to max_length  loop
```

```
        a (i) := buffer (i) ;
        next when buffer (i) = ' ' ;
        exit load when buffer (i) = NUL ;
    end loop load ;
```

Egyéb utasítások

A vezérlési szerkezeteknél két utasítást kell még megemlíteni; az egyik a **null** utasítás, a másik az **assert** utasítás. A **null** utasítás olyan utasítás, amelynek semmilyen hatása nincs. Általában a **case** szerkezetnél használjuk, mivel elõfordulhatnak olyan esetek, amelyekhez nem akarunk utasításokat rendelni.

Pl:
```
    case literal is
        when '0'          =>    zero := true ;
        when '1' to '9' =>    value := true ;
        when others  =>    null ;
    end case ;
```

Az **assert** utasítás segítségével a program futása során üzeneteket küldhetünk a felhasználónak. Ha az **assert** utasítás után álló feltétel hamis, akkor a feltétel után írt szöveges rész megjelenik a program futása során. Ha nincs szöveges rész, akkor a default-ként megadott "Assertion violation" üzenet íródik ki. Az **assert** utasítás szintaktikája a következõ:

```
assert_utasítás ::=
    assert feltétel
        [ report kifejezés ]
        [ severity kifejezés ] ;
```

A **report** mögötti kifejezés bármilyen karaktersorozat lehet. A **severity** után álló kifejezés típusa az elõredefiniált *severity_level* típus.

Pl:
```
    assert delay = 0 ns
        report Nincs késleltetés.
        severity Note ;
```

Eljárások, függvények

Más magasszintû programozási nyelvekhez hasonlóan a VHDL-ben is lehetõség van eljárások és függvények definiálására. Az eljárásokat és a függvényeket az alábbi szintaktika szerint kell deklarálni:

```
alprogram_deklaráció ::=
    procedure azonosító [ ( formális_paraméter_lista ) ]
    | function azonosító [ ( formális_paraméter_lista ) ] return típus_jelzés

formális_paraméter_lista ::= interface_elem { ; interface_elem }

interface_elem ::=    interface_konstans_deklaráció
                    | interface_változó_deklaráció
                    | interface_jel_deklaráció*
```

interface_kostans_deklaráció ::=
[**constant**] azonosító_lista : [**in**] típus_megjelölés

interface_változó_deklaráció ::=
[**variable**] azonosító_lista : [**inout** | **out**] típus_megjelölés

Az eljárások és függvények deklarációs része csak a hivatkozáshoz szükséges azonosítót, a formális paramétereket, és függvények esetében még a visszatérési értéket tartalmazza. Az alprogramok mûködését definiáló törzs szintaktikája késõbb kerül bemutatásra.

Az interface elemek deklarációjánál a **constant** és az **in** kulcsszó közül legalább az egyiket meg kell adni, akárcsak a **variable** és az **inout** vagy **out** kulcsszavak esetén. Az **in** módba álló paraméterek az alprogram végrehajtásakor csak olvashatók, tehát konstansként viselkednek. Az **inout** módban álló paraméterek nem csak olvashatók, hanem értéket is lehet hozzájuk rendelni, míg az **out** módban álló paraméterek kizárólag értékadásra használhatók fel. Az **inout** és **out** módban álló paraméterek tehát változóként viselkednek az alprogram végrehajtása során.
Függvények esetén a formális paraméter lista csak **in** módban álló konstansokat tartalmazhat, ezért sem az **in**, sem a **constant** kulcsszót nem szükséges kiírni.

Pl:
```
procedure  reset
procedure  increment_reg (variable  reg : inout  word_32 ;
                                    constant  incr :  in  Integer := 1)
procedure decrement_reg ( result : out word_32 ;
                                    reg : inout  word_32 ;
                                    decr : in  Integer := 1)
function byte_to_int (byte : word_8) return Integer
```

Egy alprogram meghívásakor meg kell adnunk az alprogram azonosítóját, valamint a formális paraméterekbe behelyettesítendõ aktuális paramétereket. Az aktuális paramétereket az alprogram azonosítója után zárójelben kell felsorolni, méghozzá ugyanabban a sorrendben, ahogy az alprogram deklarációs részében a formális paramétereket soroltuk fel. Az aktuális paramétereket tetszõleges sorrendben is felsorolhatjuk, azonban ilyenkor a formális és az aktuális paraméterek között explicit hozzárendelésre van szükség.

Ha egy alprogramot kevesebb aktuális paraméterrel hívunk meg, mint ahány formális paramétert deklaráltunk, akkor az aktuális paraméterek rendre hozzárendelõdnek a formális paraméter lista elsõ elemeihez, a maradék formális paraméterek pedig vagy az általunk definiált kezdeti értéküket vagy annak hiányában a default értéküket veszik fel. Ha egy alprogram hívásakor az aktuális paraméterek száma több, mint a formális paramétereké, akkor a fordító hibát jelez.

Pl:
```
reset ;                                  -- paraméter nélküli eljáráshívás
incerment_reg (index_reg, offset) ;    -- adjuk hozzá az index_reg-hez
                                         -- az offset értékét
increment_reg (prog_counter) ;           -- adjunk 1-et a prog_counter-hez



int_value := byte_to_int ( In_reg ) ;   -- függvényhívás

-- explicit paraméterátadás:
```

```
decrement_reg (decr => offset, result => A_reg, reg => B_reg) ;
```

Az alprogramok mûködését a törzsükben definiált utasítássorozat írja le. Egy alprogramot teljes egészében – tehát törzsével együtt – a következõ szintaktika szerint írhatunk le:

```
alprogram ::=
        alprogram_deklaráció  is
                deklarációs_lista
        begin
                utasítás_sorozat
        end  [ azonosító ] ;

deklarációs_lista ::= {alprogram_deklaráció | típus_deklaráció |
                        altípus_deklaráció | konstans_deklaráció |
                        változó_deklaráció | alias_deklaráció}
```

A függvények törzse mindig tartalmaz egy speciális **return** utasítást, amelynek hatására a vezérlés visszatér a hívás helyére. Ha a **return** utasítás után valamilyen kifejezés áll, akkor a függvény visszatérési értéke az adott kifejezés értéke lesz. Az eljárások is tartalmazhatnak **return** utasítást, de ott a **return** után nem állhat semmilyen kifejezés.

Pl:
```
function  byte_to_int  (byte : word_8)  return  Integer  is
        variable  result : Integer := 0;
begin
        for  index  in  0 to 7  loop
                result := result*2 + bit'pos(byte(index));
        end  loop;
        return  result;
end  byte_to_int;

procedure  add_proc (variable  a : inout  Integer;
                        constant  incr :  in  Integer := 1)  is
begin
        loop
                a := a + inc;
                if  a > 100  then
                        return;
                end  if;
        end  loop;
end  add_proc;
```

A VHDL nyelv megengedi, hogy különbözõ alprogramoknak ugyanaz legyen az azonosítója, feltéve, hogy formális paramétereik száma vagy típusa nem egyezik meg (overloading). Amikor meghívunk egy ilyen alprogramot, akkor az aktuális paraméterek száma és típusa fogja meghatározni, hogy a megegyezõ nevû alprogramok közül melyik hajtódjon végre.

Pl:
```
function  check_limit  (value : Integer)  return  Boolean;
function  check_limit  (value : word_32)  return  Boolean;

test := check_limit (4095);              -- az elsõ függvényt hívja
test := check_limit (X"0000_0FFF");  -- a második függvényt hívja
```

Package-ek

A VHDL egyik jellemzõ tulajdonsága, hogy az entitásban deklarált objektumok csak az adott entitáshoz tartozó architektúrák számára látható, tehát más entitások számára nem elérhetõek. Ezért a VHDL-ben létrehozhatunk olyan önmagukban nem futtatható programmodulokat, amelyekben bármely entitás számára elérhetõ objektumokat (adattípusokat, konstansokat, alprogramokat) deklarálhatunk. Ezeket a programmodulokat package-eknek nevezzük. A package-ek – az alprogramokhoz hasonlóan – deklarációs részbõl és törzsbõl állnak. Egy package deklarációs részét a következõ szintaktika szerint adhatjuk meg:

```
package_deklaráció ::=
        package  azonosító  is
                package_deklarációs_lista
        end  [ azonosító ] ;

package_deklarációs_lista ::= {alprogram_deklaráció | típus_deklaráció  |
                                altípus_deklaráció | kostans_deklaráció |
                                alais_deklaráció | use_clause}

use_clause ::= use  package_név { ,  package_név } ;
package_név ::= név . kiterjesztés
```

A deklarációs részben deklarált objektumokat a package törzsén kívül bármely más entitás is felhasználhatja. Fontos megjegyezni, hogy a package törzsében deklarált objektumokat viszont csak a package törzse látja, tehát azokat más entitások vagy package-ek nem érik el. A package törzsét a következõ szintaktika szerint adhatjuk meg:

```
package_törzs ::=
        package  body  azonosító  is
                package_törzs_deklarációs_lista
        end  [ azonosító ] ;

package_törzs_deklarációs_lista ::= {alprogram | típus_deklaráció |
                                     altípus_deklaráció | konstans_deklaráció |
                                     alias_dekaráció | use_clause}
```

Az alprogramok törzse mindig a package-ek törzsében szerepel, mivel a package-ek deklarációs részében az alprogramoknak is csak a deklarációja állhat.

Pl:
```
package  data_types is
        type address is bit_vector (23 downto 0);
        type data  is bit_vector (15 downto 0);
        constant  vector_table_loc : address;
        function  data_to_int (value : data) return  Integer;
        function  int_to_data (value : Integer) return data;
end  data_types;
```

```
package body data_types is
        constant vector_table_loc : address := X"FFFF00";

        function data_to_int (value : data) return Integer is
                a data_to_int függvény törzse
        end data_to_int;

        function int_to_data (value : Integer) return data is
                az int_to_data függvény törzse
        end int_to_data;
end data_types;
```

Egy package-ben deklarált objektumra úgy hivatkozhatunk, hogy a package neve után kiterjesztésként megadjuk az objektum nevét.

Pl: **variable** proc_address : data_types.address;
offset := data_types.data_to_int (offset_reg);

Ha gyakran akarunk hivatkozni package-ekben deklarált objektumokra, akkor kényelmesebb megoldást jelent az ún. *use clause* használata, amely lehetõvé teszi számunkra, hogy a hivatkozásoknál elhagyhassuk a package nevét, és csak az objektum nevét kelljen leírnunk. A *use clause* mindig a VHDL program legelején áll.

Pl: **use** data_types.**all**;
...
variable proc_address : address;
offset := data_to_int (offset_reg);

Konkurrens vezérlési szerkezetek

Értékadás jelnek

A jelek olyan globális változók, amelyek lehetõvé teszik információk továbbítását a konkurrens programrészek között. Jeleknek a következõ szintaktika szerint adhatunk értéket:

```
jel_értékadás ::=
        jel <= [ transport ] jelalak ;

jelalak ::= jelalak_elem { , jelalak_elem }
jelalak_elem ::= érték_kifejezés [ after idõ_kifejezés ] |
                 null [ after idõ_kifejezés ]
```

Az idõ_kifejezés az érték_kifejezés késleltetését jelenti az aktuális szimulációs idõponthoz képest. Ha nem adjuk meg a késleltetés értékét, akkor a default értéke 0 fs lesz, tehát az értékadás az adott szimulációs idõpillanatban egyből végrehajtódik.

Pl: s <= '0' **after** 10 ns;
s <= '0', '1' **after** 4 ms, '0' **after** 20 ms;

Jelek értékadásánál a **transport** kulcsszó leírásával vagy elhagyásával két különbözõ viselkedését adhatjuk meg az adott jelnek. A különbség megértése céljából jelöljük (x, y)-nal azt, hogy a jelhez *y* késeltetéssel *x* értéket rendelünk hozzá. Ha egy jelhez különbözõ késeltetésekkel több értéket is hozzárendelünk, akkor az egyes értékadásokat fogjuk közre egy kapcsos zárójellel.

Pl: s <= '0', '1' **after** 4 ms, '0' **after** 20 ms;
-- s : {('0', 0 ms) ('1', 4 ms) ('0', 20 ms)}

Ha a **transport** kulcsszót kiírjuk (*transport delay*), akkor az új hozzárendelés során fölülíródnak azok a korábbi értékadások, amelyek az új hozzárendelés elsõ értékadása utáni idõpontokra lettek ütemezve. Tehát ha például a 0 ns idõpontban történt egy

s <= '1' **after** 20 ns, '0' **after** 36 ns;
-- s : {('1', 20 ns) ('0', 36 ns)}

hozzárendelés, akkor a 18 ns idõpontban az

s <= **transport** 'Z' **after** 10 ns;

hozzárendeléssel az s:{('1', 20 ns) ('Z', 28 ns)} ütemezést írjuk elõ.

Ha a **transport** kulcsszót elhagyjuk (*inertial delay*), akkor a korábbi értékadások általában mind törlõdnek. Ha van olyan korábbi értékadás, amely az új hozzárendelés elsõ értékadását közvetlenül megelõzi és ugyanazt az értéket rendeli a jelhez, akkor ez az egy korábbi értékadás megmarad.
Ha például a 0 ns idõpontban végrehajtunk egy

s <= '1' **after** 10 ns, '0' **after** 15 ns, '1' **after** 20 ns, 'Z' **after** 30 ns;
-- s : {('1', 10 ns) ('0', 15 ns) ('1', 20 ns) ('Z', 30 ns)}

hozzárendelést, akkor az 5 ns idõpontban az

s <= '1' **after** 20 ns;

hozzárendeléssel az s:{('1', 20 ns) ('1', 25 ns)} ütemezést írjuk elõ.

Ha egy jelhez egyidejûleg több értéket is rendelünk a **transport** kulcsszó megadása nélkül, akkor az elsõ értékadást *inertial delay*-ként, az utána álló összes többi értékadást viszont *transport delay*-ként értelmezi a fordító.

Folyamatok

A folyamatok (*processes*) olyan önmagukban szekvenciális programrészek, amelyek egymással párhuzamosan futnak. Egy folyamat futását speciális utasításokkal fel is függeszthetjük, ilyenkor a folyamat valamilyen esemény bekövetkezéséig várakozik. Folyamatokat a következõ szintaktika szerint írhatunk le:

folyamat ::=
[címke :]
process [(szenzitív_lista)]
process_deklarációs_lista
begin
process_utasítás_sorozat
end process [címke] ;

szenzitív_lista ::= jel_név { , jel_név }
process_deklarációs_lista ::= {alprogram | típus_deklaráció |
altípus_deklaráció | konstans_deklaráció |

```
                                    változó_deklaráció | alias_deklaráció |
                                     use_clause}
process_utasítás_sorozat ::=  {wait_utasítás | assert_utasítás |
                               jel_értékadás | változó_értékadás |
                               eljárás_hívás | if_szerkezet |
                               case_szerkezet | loop_ciklus
                               for_ciklus | while_ciklus |
                               next_utasítás | exit_utasítás |
                               return_utasítás | null_utasítás}
```

A folyamatok a törzsükben definiált utasítássorozatok ciklikusan ismétlik. A speciális **wait** utasítás hatására az utasítássorozat végrehajtása leáll, és a folyamat egészen addig várakozik, amíg valamilyen általunk megadott esemény be nem következik. Ilyen esemény lehet egy jel értékének megváltozása (sensitivity clause), valamilyen logikai feltétel teljesülése (condition clause) vagy egy megadott idõtartam eltelte (timeout clause). A wait utasítás szintaktikája a következõ:

```
wait_utasítás ::=
        wait  [ sensitivity_clause ]  [ condition_clause ]  [ timeout_clause ] ;

sensitivity_clause ::= on  szenzitív_lista
condition_clause ::= until  feltétel
timeout_clause ::= for  idõ_kifejezés
```

Ha a **wait** utasítás után nem áll semilyen kifejezés, akkor a folyamat végtelen hosszú ideig várakozik. A szenzitív listát megadhatjuk a folyamat deklarációs részében a **process** kulcsszó után is, ami egyenértékû azzal, mintha a folyamat törzsében az utolsó utasítás egy '**wait on** szenzitív_lista' utasítás lenne.

Pl:
```
process (reset, clock)
        variable  state : Boolean := false;
begin
        if  reset  then
                state := false;
        elsif  clock = '1'  then
                state := not state;
        end  if;
        s <= state  after  10 ns;
        -- wait on reset, clock;  –  implicit utasítás a deklarációs részben
end  process;

xor_proc:  process
begin
        wait  until  a = '1'  and  b = '0';
        q <= '1';
        wait  until  a = '0'  and  b = '1';
        q <= '0';
end  process  xor_proc;



clock:  process
```

```
begin
        Clk <=  not Clk;
        wait  for  10 ns;
end  process  clock;
```

Egy folyamatban tetszõleges számú olyan utasítás lehet, amely ugyanahhoz a jelhez rendel értékeket. Ezek az utasítások az adott jelre nézve egy *driver*-t alkotnak. Általában egy jelhez csak egy folyamatban rendelünk értékeket, más folyamatokban csak figyeljük az adott jelet, tehát egy jelhez egy *driver* tartozik. Speciális esetekben persze lehetõség van arra is, hogy egy jelhez több *driver* tartozzon, ilyenkor azonban gondoskodni kell arról, hogy a különbözõ értékek hozzárendeléséből adódó konfliktust valahogyan feloldjuk. Ezt a célt szolgálják az ún. feloldó függvények (resolution functions), mint például a huzalozott VAGY (Wired-OR) vagy a huzalozott ÉS (Wired-AND).

Feloldó függvények

A több *driver*-rel rendelkezõ jeleket feloldott jeleknek nevezzük. A feloldott jelekhez mindig meg kell adni egy feloldó függvényt. A VHDL fejlesztõrendszerek általában rendelkeznek néhány elõredefiniált feloldó függvénnyel, de magunkis definiálhatunk ilyeneket.

Pl:
```
type  Bit4  is  ('X',  '0',  '1',  'Z');
type  Bit4_vector  is  array  (Integer  range  <>)  of  Bit4;
...
function  Wired_Or  (Input : Bit4_vector)  return  Bit4  is
        variable  Result : Bit4 := '0';
begin
        for  i  in  Input'Range  loop
                if  Input(i) = '1'  then
                        Result := '1';
                        exit;
                elsif  Input(i) = 'X'  then
                        Result := 'X';
                else                                    -- Input(i) = 'Z'  or  '0'
                        null;
                end  if;
        end  loop;
        return  Result;
end  Wired_Or;
```

Feloldott jelet a következõ szintaktika szerint deklarálhatunk:

```
feloldott_jel_deklaráció ::=
        signal  azonosító_lista  :
                feloldó_függvény_név   típus_megjelölés  [ :=  kifejezés ] ;
```

Pl:
```
signal  Interrupt : Wired_Or  Bit4;
```

Konkurrens értékadás jeleknek

Gyakran elõfordul, hogy egy folyamat kizárólag egy jelhez rendel értékeket különbözõ feltételek mellett. Ilyen esetekben lehetõség van arra, hogy egy külön folyamat defniálása nélkül, egyszerûsített formában rendeljünk értékeket az adott jelhez. Az értékadásnak ezt a formáját a VHDL-ben konkurrens értékadásnak nevezzük. A konkurrens értékadás szintaktikája a következõ:

```
konkurrens_jel_értékadás ::=
        [ címke : ]  feltételes_jel_értékadás |
        [ címke : ]  kiválasztó_jel_értékadás

feltételes_jel_értékadás ::=
        jel  <= [ transport ]  feltételes_jelalak ;

feltételes_jelalak ::=  { jelalak  when  feltétel  else }  jelalak

kiválasztó_jel_értékadás ::=
        with  kifejezés  select
                jel  <=  [ transport ]  kiválasztó_jelalak ;

kiválasztó_jelalak ::=  { jelalak  when  választás , }  jelalak  when  választás
választás ::=  eset  { | eset }
```

A feltételes értékadás egyenértékû egy folyamatban definiált **if** szerkezettel.

```
Pl:     s <=    'Z'  when  en = '0'  else
                '0'  after  10 ns  when  sel = '0'  else
                '1'  after  10 ns ;

        -- a konkurrens értékadással ekvivalens folyamat:
        process
                if  en = '0'  then
                        s <=  'Z';
                elsif  sel = '0'  then
                        s <=  '0'  after  10 ns;
                else
                        s <=  '1'  after  10 ns;
                end  if;
                wait  on  en, sel;
        end  process;
```

A kiválasztó értékadás egy folyamatban definiált **case** szerkezettel egyenértékû.

Pl:
```
with alu_function select
        alu_result <= op1+op2 when alu_add | alu_incr,
                      op1–op2 when alu_sub,
                      op1 and op2 when alu_and,
                      op1 or op2 when alu_or;

-- a konkurrens értékadással ekvivalens folyamat:
process
        case alu_function is
                when alu_add | alu_incr =>
                  alu_result <= op1+op2;
                when alu_sub =>
                  alu_result <= op1–op2;
                when alu_and =>
                  alu_result <= op1 and op2;
                when alu_or =>
                  alu_result <= op1 or op2;
        end case;
        wait on alu_function;
end process;
```

Ha a konkurrens értékadásban egyik jelalak, feltétel vagy kifejezés sem hivatkozik jelre, akkor az ekvivalens folyamatban a **wait** utasítás után nem áll szenzitív lista, azaz a folyamat az elsõ értékadás után végtelen hosszú ideig várakozik.

Struktúrális leírás

Az entitás

A digitális rendszereket általában hierarchikusan egymásra épülõ modulok alkotják. Ezeket a modulokat a VHDL-ben entitásoknak nevezzük. Minden entitás rendelkezik egy vagy több porttal, amelyek az entitás és a külvilág között teremtenek kapcsolatot. Egy entitás lehet a tervezés legmagasabb szintû modulja, de lehet egy olyan komponens is, amely része egy másik modulnak. Entitást a következõ szintaktika szerint deklarálhatunk:

entitás_deklaráció ::=
entity azonosító **is**
[**generic** (generic_lista)]
[**port** (port_lista)]
[**begin**
utasítás_sorozat]
end [azonosító] ;

generic_lista ::=
[**constant**] azonosító_lista : [**in**] típus_jelzés [:= kifejezés]

port_lista ::=
[**signal**] azonosító_lista : [mód] típus_jelzés [**bus**] [:= kifejezés]

mód ::= **in | out | inout**

A generic lista olyan konstansokat tartalmaz, amelyeket az entitás struktúrájának és viselkedésének vezérléséhez használunk. A **constant** és az **in** kulcsszó használata nem kötelezõ, mivel a generic lista eleve csak konstansokat tartalamzhat.

A port lista azokat a jeleket tartalmazza, amelyek az entitás belseje és környezete közötti kommunikációt teszik lehetõvé. A **signal** kulcsszó használata itt sem kötelezõ, mivel a port listán kizárólag csak jelek szerepelhetnek. A **bus** kulcsszót akkor használjuk, ha a jelhez egynél több *driver* tartozik.

Pl:
```
entity processor is
        generic (max_clock_freq := 30 MHz);
        port (clock : in Bit;
                address : out Integer;
                data : inout word_32 bus;
                control : out word_16;
                ready : in Bit );
end processor;
```
A fenti példában a *max_clock_freq* konstansnak az entitás viselkedési leírásánál lesz szerepe. A következõ példában a generic paraméterek az entitás struktúráját határozzák meg.

Pl:
```
entity rom is
        generic (width, depth : positive);
        port (enable : in Bit;
                address : in Bit_vector (depth–1 downto 0);
                data : out Bit_vector (width–1 downto 0) );
        end rom;
```

A generic listán szereplõ *width* és *depth* konstansok egyike sem rendelkezik egyelõre konkrét értékkel. Értéket akkor kapnak majd, amikor az entitást komponensként fogjuk felhasználni egy másik entitásban.

Az architektúra

Egy adott interface deklaráció mellett az entitás implementálását több különbözõ módon is elvégezhetjük. Megadhatunk egy egyszerû viselkedési leírást, vagy akár egy olyan struktúrális leírást, amelyben különbözõ komponensekbõl hierarchikusan építjük fel az entitást. Az entitás egy adott implementációját architektúrának nevezzük. Az architektúra szintaktikája a következõ:

architektúra ::=
```
        architecture azonosító of entitás_név is
                architektúra_delarációs_lista
        begin
                architektúra_utasítás_sorozat
        end [ azonosító ] ;
```

architektúra_deklarációs_lista ::=
```
        {alprogram | típus_deklaráció | altípus_deklaráció |
         konstans_deklaráció | jel_deklaráció | alias_deklaráció |
         komponens_deklaráció | configuráció_specifikáció |
         use_clause }
```

architektúra_utasítás_sorozat ::=
```
        {konkurrens_jel_értékadás | folyamat | blokk | komponens_leképezés}
```

Konkurrens értékadást és folyamatokat elsõsorban a viselkedés-orientált implementációk tartalmaznak, míg blokkokat és komponenseket inkább a struktúra-orientáltak. Bonyolultabb rendszerek esetén – fõleg magasabb szintû modulokban – a viselkedési és a struktúrális leírást már nem lehet szétválasztani, ilyenkor egy achitektúrán belül a konkurrens értékadások, a folyamatok, a blokkok és a komponensleképzések egymás mellett is megtalálhatók.

A blokkok

A blokk olyan önálló modul, amely más modulok belsejében áll, de csak az egy modul használhatja, amelyikben definiáltuk. A blokk – akárcsak más modul – saját interfésszel rendelkezik, amelyen keresztül más blokkokal vagy az õt tartalmazó modul portjaival teremt kapcsolatot. Blokkot a következõ szintaktika szerint definiálhatunk:

```
blokk ::=
        címke :
        block
                [ generic  ( generic_lista ) ;
                  generic  map  ( generic_leképzési_lista ) ; ]
                [ port  ( port_lista ) ;
                  port  map  ( port_leképzési_lista ) ; ]
        begin
                blokk_utasítás_sorozat
        end  block  [ címke ] ;
```

A generic lista a blokk egészére jellemzõ konstansokat deklarálja, a port lista pedig azokat a jeleket, amelyek a blokk interfészét alkotják. A generic leképzési lista a generic konstansokhoz rendel közvetlenül vagy közvetve értékeket. Közvetett értékadás esetén a generic leképzési listán a generic konstansokhoz olyan érték nélküli konstansokat rendelünk, amelyek csak késõbb, a blokk törzsében kapnak értéket. A port leképzési listán a blokk portjaihoz rendelünk hozzá olyan jeleket, amelyeket a blokk törzsében deklarálunk. A blokk törzsében definiált utasítássorozat ugyanazokat az elemeket tartalmazza, mint az architektúra törzse.

Pl:
```
architecture  bock_structure  of  processor  is
        signal  int_control : data_path_control;
begin
        control_unit : block
                port (  clk : in  Bit;
                        bus_control : out  proc_control;
                        bus_ready : in  Bit;
                        control : out data_path_control);
                port  map (clk => clock, bus_control => control,
                                bus_ready => ready, control => int_control);
                a control_unit blokk deklarációi
        begin
                a control_unit blokk utasításai
        end  block  control_unit;

        data_path : block
                port (  address : out  Integer;
                        data : inout  word_32;
                        control : in  data_path_control);
                port  map  (address => address, data => data,
                                control => int_control);
                a data_path blokk deklarációi
        begin
                a data_path blokk utasításai
        end  block  data_path;
end  block_structure;
```

Fontos megjegyezni, hogy egy blokk belsejében újabb blokkokat definiálhatunk, így lehetõség nyílik olyan hierarchikus struktúra kialakítására, amelyben a viselkedési leírást tartalmazó modulok csak a legalsó szinten jelennek meg. A blokkok egy másik elõnyös tulajdonsága, hogy lehetõvé teszik a moduláris programozást, azaz az egyes blokkokat – az interfészek deklarálása után – egymástól függetlenül lehet implementálni.

A komponensek

A komponensek olyan entitások, amelyeket bármely más entitás felhasználhat az architektúra leírásához. A komponenseket tervezõi könyvtárakban szoktuk tárolni. A komponenseket úgy használhatjuk föl, hogy az architektúra deklarációs részében elõbb deklaráljuk õket az interfészükkel, majd az architektúra törzsében a komponens portjaihoz hozzárendeljük a komponenst tartalmazó entitás megfelelõ jeleit. Komponenst a következõ szintaktika szerint deklarálhatunk:

komponens_deklaráció ::=
component azonosító
[generic_lista]
[port_lista]
end component ;

Pl: **component** nand3
generic (Tpd : Time);
port (a, b, c : **in** logic_level;
y : **out** logic_level);
end component;

A komponensek leképezése a következõ szintaktika szerint történik:

komponens_leképezés ::=
címke : komponens_azonosító
[**generic map** (generic_leképzési_lista)]
[**port map** (port_leképzési_lista)] ;

A generic leképzési lista és a port leképzési lista jelentése itt ugyanaz, mint blokkok esetén.

Pl: enable_gate : nand3
generic map (Tpd => 1 ns);
port map (a => en1, b => en2, c => int_req, y => interrupt);

A komponensek jelentõsége abban áll, hogy lehetõvé teszik a könyvtáralapú tervezést, így egyszerûbb rendszerek esetén elegendõ a struktúrális leírással foglalkoznunk – a viselkedési leírást a komponensek tartalmazzák.

```
Pl:   architecture component_structure of processor is
        signal int_control : data_path_control;

        component control_unit
            port ( clk : in Bit;
                   bus_control : out proc_control;
                   bus_ready : in Bit;
                   control : out data_path_control);
        end component;

        component data_path
            port ( address : out Integer;
                   data : inout word_32;
                   control : in data_path_control);
        end component;

    begin
        component_1 : control_unit
            port map (clk => clock, bus_control => control,
                      bus_ready => ready, control => int_control);
        component_2 : data_path
            port map (address => address, data => data,
                      control => int_control);
    end component_structure;
```

Konfigurációk

A VHDL-ben lehetõség van arra, hogy a komponensekhez magunk válasszuk ki, melyik entitás melyik architektúráját rendeljük hozzá a leképezés során. A komponens mint entitás több architektúrával is rendelkezhet, tehát több féleképpen is implementálható. A különbözõ módon implementált komponensek különbözõ viselkedésû entitásokat eredményeznek, amiket az adott entitás konfigurációinak nevezünk. A blokkok mint beágyazott entitások szintén konfigurálhatók. Konfigurációkat a következõ szintaktika szerint specifikálhatunk:

```
konfiguráció_deklaráció ::=
    configuration azonosító of entitás_név is
        {use_clause}
        konfiguráció_specifikáció
    end [ azonosító ] ;

konfiguráció_specifikáció ::=
    for architektúra_név | blokk_címke
        {use_clause}
        {konfiguráció_specifikáció | komponens_konfiguráció}
    end for ;
```

```
komponens_konfiguráció ::=
        for  leképzési_lista  :  komponens_azonosító
                [ use  kötés_jelzés ; ]
                [ konfiguráció_specifikáció ]
        end  for ;

leképzési_lista ::=  címke  { ,  címke } | others | all

kötés_jelzés ::=
        entitás_kötés
        [ generic  map  ( generic_leképzési_lista ) ]
        [ port  map  ( port_leképzési_lista ) ]

entitás_kötés ::=
        entity  entitás_név  [ ( architektúra_név ) ] |
        configuration  konfiguráció_név |
        open
```

A szintaktikai leírásból kiderül, hogy egy adott konfigurációban további konfigurációkat adhatunk meg alacsonyabb szinten, tehát egy konfiguráció hierachikus felépítésû is lehet. Minden **for** keretbe ágyazott absztrakciós szint hivatkozhat tervezõi könyvtárakra és package-ekre, de a konfiguráció deklarációs részében az egész konfigurációra érvényes hivatkozásokat is megadhatjuk.

Komponens konfigurálása esetén a leképzési lista a komponensek kiválasztására szolgál. A kiválasztást elvégezhetjük a leképzéskor megadott címkék felsorolásával, valamint az **others** és az **all** kulcsszavak használatával. Az **others** kulcsszó lehetõvé teszi, hogy egy adott komponens esetén csak azokra a leképzésekre kelljen címkével hivatkozni, amelyeket egyedileg konfigurálunk. A többi leképzést az **others** kulcsszó segítségével közösen konfigurálhatjuk. Az **all** kulcsszó használatával egy komponens összes leképzéséhez ugyanazt a konfigurációt rendelhetjük hozzá.

Kötés jelzésekor a generic és a port leképezést csak akkor kell megadnunk, ha azt architektúrában nem végeztük el. Az entitáskötésnél vagy egy entitást rendelünk a komponenshez, vagy megadjuk, hogy melyik konfiguráció tartalamazza a hozzárendelést. Ha egy komponenshez nem akarunk semmit hozzárendelni, akkor az **open** kulcsszót használjuk.

Pl:

```
configuration  block_config  of  processor  is

use  work.processor_types.all;

for  block_structure    -- architecture of entity processor
        for  control_unit                -- block of arcitecture block_stucture
                for c_alu : alu_control  -- component of block control_unit
                        use  entity  proc_cells.alu_cell (behaviour)
                                generic  map  (width => 32)
                                port  map  (function_code => function,
                                                operand_1 => op_1,
                                                operand_2 => op_2,
                                                result => result);
                end  for;
                további komponensek konfigurálása
        end  for;
```

```
        for data_path          -- block of  architecture block_structure
            for d_alu : alu_data   -- componentof block data_path
                use configuration proc_cells.alu_struct
                    generic map (width => 32)
                    port map (function_code => function,
                              operand_1 => op_1,
                              operand_2 => op_2,
                              result => result);
            end for;
            további komponensek konfigurárlása
        end for;

    end for;
```

A control_unit blokkban az alu_control komponenshez a proc_cells tervezõi könyvtárban található alu_cell entitás behaviour architektúráját rendeltük hozzá. A data_unit blokkban az alu_data komponenshez az alu_struct nevû konfiguráció tartalmazza az entitáskötést. Az alu_struct konfiguráció szintén a proc_cells tervezõi könyvtárban található.

Példánkban feltételeztük, hogy a komponensek leképzésekor a generic és a port leképezéseket nem adtuk meg, ezért szerepelnek itt a kofiguráció specifikációban.

Pl:

```
    configuration component_config of processor is

    for component_structure      -- architecture of entity processor
        for component_1 : control_unit      -- component
            use entity work.controller (behaviour);
        end for;

        for component_2 : data_path         -- component
            use entity work.data_cells (behaviour)
            for behaviour          -- architetcure of entity data_cells
                for a_1, a_2 : adder    -- selected components
                    use entity msi_lib.half_adder (behaviour);
                end for;
                for others : adder              -- unselected components
                    use entity msi_lib.full_adder (behaviour);
                end for;
                for all : bus_driver            -- all components
                    use entity msi_lib.bidir_driver (structure);
                end for;
        end for;
    end for;
```

Az iménti példában feltételeztük, hogy a komponensek leképzésekor a generic és a port leképezéseket is elvégeztük. A *component_2* cimkéjû *data_path* komponens konfigurálása során további alacsonyabb szintû komponensek konfigurálására is sor került, amelyek segítségével a leképzési lista különbözõ lehetséges változatainak használatát és a hierarchikus szervezést is nyomon követhetjük.

Reguláris struktúrák

A VHDL-ben a blokkokból, komponens leképezésekbõl vagy folyamatokból álló tömböket reguláris struktúráknak hívjuk. Reguláris struktúrát a **generate** konkurrens utasítással hozhatunk létre egy architektúra belsejében. Ennek szintaktikája a következõ:

```
generate_utasítás ::=
        címke :
        többszörözõ_utasítás  generate
                {konkurrens_utasítás}
        end  generate  [ címke ] ;

többszörözõ_utasítás ::=
        for  ciklusváltozó  in  diszkrét_tartomány |
        if  feltétel
```

Többszörözõ utasításként a **for** ciklust akkor használjuk, amikor azonosan ismétlõdõ elemekbõl képezünk tömböt. Az **if** utasítás a reguláris struktúrán belüli eltérõ esetek szétválasztására biztosít lehetõséget. A két többszörözõ utasítás használatát a következõ példa is szemlélteti:

```
Pl:     adder : for  i  in  0  to  width-1  generate

                ls_bit : if  i = 0  generate
                        ls_cell : half_adder port  map (a(0), b(0), sum(0), c_in(0));
                end  generate  ls_bit;

                middle_bit : if  i > 0  and  i < width-1  generate
                        middle_cell :
                        full_adder port  map (a(i), b(i), c_in(i), sum(i), c_in(i+1));
                end  generate  middle_bit;

                ms_bit : if  i = width-1  generate
                        ms_cell :
                        full_adder port  map (a(i), b(i), c_in(i), sum(i), carry);
                end  generate  ms_bit;

        end  generate  adder;
```

Tervezõi könyvtárak

Az compiler által lefordított VHDL programokat tervezõi könyvtárakban helyezzük el. A könyvtári elemeknek két fõ típusa van. Az entitásdeklarációkat, a package-deklarációkat és a konfiguráció specifikációkat elsõdleges könyvtári elemeknek nevezzük, míg az architektúrák és a package-ek törzsei másodlagos könyvtári elemek. A két típus között az a különbség, hogy a másodlagos könyvtári elemek lefordítása elõtt minden esetben el kell végezni a nekik megfelelõ elsõdleges könyvtári elem lefordítását.
Egy komplett VHDL leírás számos könyvtári elemet tartalmazhat. Ezeket a könyvtári elemeket együttesen tervezõi file-nak nevezzük. A tervezõi file felépítését a következõ szintaktika adja meg:

```
tervezõi_file ::= tervezõi_egység  { tervezõi_egység }
tervezõi_egység ::= context_clause  könyvtári_elem
```

context_clause ::= { library_clause | use_clause }
library_clause ::= **library** logikai_név_lista
logikai_név_lista ::= logikai_név { , logikai_név }
könyvtári_elem ::= elsõdleges_elem | másodlagos_elem
elsõdleges_elem ::= entitás_deklaráció | konfiguráció_deklaráció |
package_deklaráció
másodlagos_elem ::= architektúra | package_törzs

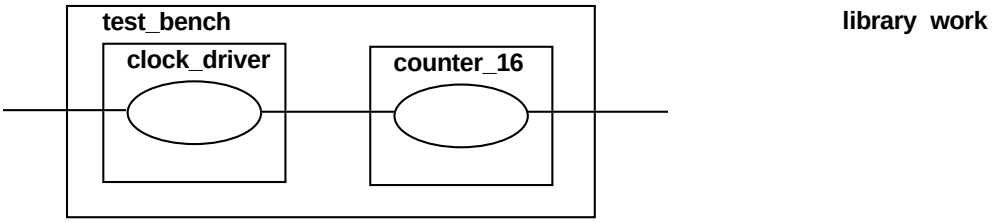
A tervezõi könyvtárakra logikai nevekkel hivatkozhatunk. Egy tervezõi könyvtár logikai neve és az operációs rendszer által ismert fizikai neve egymástól független, de általában megegyezik. A logikai és fizikai nevek egymáshoz rendelését, valamint a tervezõi könyvtárakkal kapcsolatos adminisztrációs feladatokat a VHDL fejlesztõrendszer végzi el.
Két olyan speciális tervezõi könyvtár van, amelyek tartalma minden könyvtári elem számára elérhetõ, függetlenül attól, hogy az adott könyvtári elem melyik tervezõi könyvtárban van. Az egyik a *work* könyvtár, amelybe az aktuális tervezõi file kerül a lefordítás után, a másik az *std*, amely a *standard* és a *textio* package-et tartalmazza. E két tervezõi könyvtárra sohasem szükséges külön hivatkoznunk.

Példaprogram

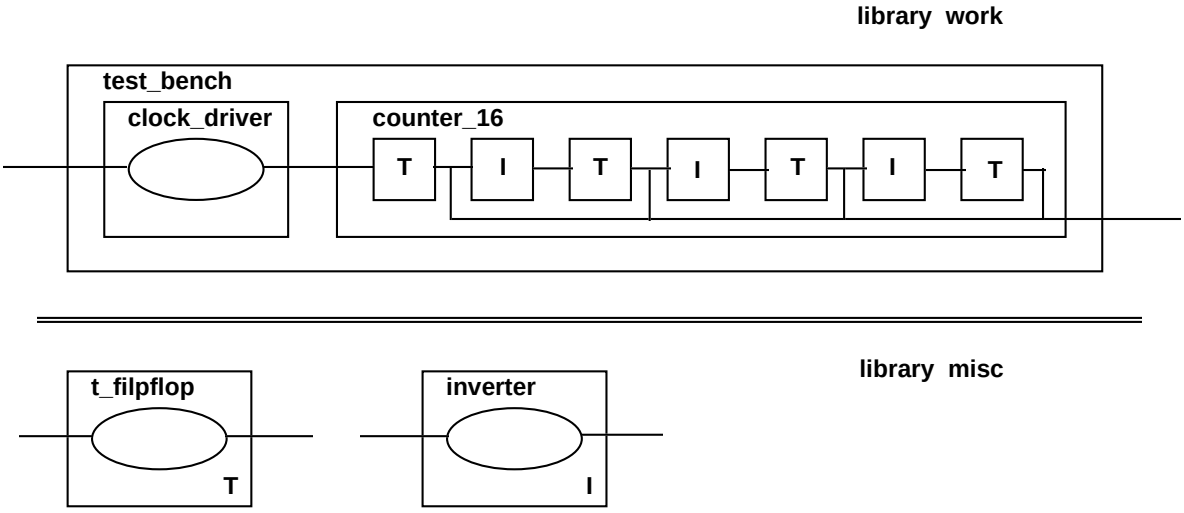
A következõkben bemutatásra kerülõ VHDL program egy egyszerû tervezési feladat eredményét mutatja be. A példa kidolgozása során arra törekedtünk, hogy a VHDL nyelvben használható programozási eszközöket minél szélesebb körben szemléltessük.

A tervezési feladat részletes leírása:
Készítsük el a VHDL leírását egy olyan digitális áramkörnek, amely órajel-engedélyezõ bemenettel és 4 bites kimenettel rendelkezik. Az órajel-engedélyezõ bemenet logikai '1' szintje egy 10 MHz-es órajel generátor mûködését engedélyezi. Az órajel felfutó éle egy 4 bites bináris számlálót léptet, a számláló kimenete egyben a rendszer kimenete is. A feladat megoldása során a *misc* tervezõi könyvtárban található *t_flipflop* és *inverter* entitásokat tekintsük elõredefiniáltnak.

A VHDL leírás elkészítésekor a 4 bites számlálót kétféle módon is implementáltuk; a *structure* nevû architektúrában egy struktúra-orintált leírást, a *behaviour* nevû architektúrában pedig egy viselkedés-orientált leírást adtunk meg. A VHDL program szerkezetét a kétféle leírásnak megfelelõen az 1. ábrán és a 2. ábrán látható modellek szemléltetik.



1. ábra



2. ábra

```
package counter_types is

        type word4 is Bit_vector (3 downto 0);

        function int_to_word4 (int_value : Integer) return word4;
        procedure increment_mod16 (value : inout Integer, delta : in Integer);

end counter_types;

package body counter_types is

        function int_to_word4 (int_value : Integer) return word4 is
                variable bit_value : word4;
                variable a : Real;
        begin
                for i in 3 downto 0 loop
                        a := int_value / 2**i ;
                        if a >= 1.0 then
                                bit_value(i) := '1';
                                int_value := int_value - 2**i ;
                        else
                                bit_value(i) := '0';
                        end if;
                end loop;
                return bit_value;
        end int_to_word4;

        procedure increment_mod16 (value : inout Integer, delta : in Integer) is
        begin
                value := (value + delta) mod 16;
                assert (value > 0)
                        report Counter overflow
                        severity Note;
        end increment_mod16;

end counter_types;



use work.counter_types.all;

entity counter_16 is
        generic (prop_delay : Time := 10 ns);
        port (clock : in Bit, q_vector : out word4);
end counter_16;

architecture behaviour of counter_16 is

begin
        count_up : process (clock)
                variable count_value : natural := 0;
        begin
                if clock = '1' then
                        increment_mod16 (count_value, 1);
                        q_vector <= int_to_word4 (count_value);
                end if;
        end process count_up;

end behaviour;
```

```
architecture structure of counter_16 is

        component t_flipflop
                port (clk : in Bit; q : out Bit);
        end component;
        component inverter
                port (a : in Bit; y : out Bit);
        end component;

        signal ff_in, ff_out : word4;

begin
        cascade : for i in 0 to 3 generate

                first_cell : if i = 0 generate
                        first_flipflop :
                                t_flipflop port map (clk => clock, q => ff_out(i));
                end generate first_cell;

                chain : if i > 0 generate
                        cell_inverter :
                                inverter port map (a => ff_out(i-1), y => ff_in(i));
                        cell_flipflop :
                                t_flipflop port map (clk => ff_in(i), q => ff_out(i));
                end generate chain;

        end generate cascade;

        q_vector <= ff_out;

end structure;

use work.counter_types.all;

entity test_bench is
        port (enable : in Bit; output : out word4);
end test_bench;

architecture structure of test_bench is

        signal test_clock : Bit;

        component counter_16
                port (clock : in Bit; q_vector : out word4);
        end component;

begin
        counter : counter_16
                port map (clock => test_clock, q_vector => output);

        clock_driver : block
                port (clock_enable : in Bit; clock_out : out Bit);
                port map (clock_enable => enable, clock_out => test_clock);

        begin
                process
                begin
                        if clock_enable = '1' then
                                clock_out <= '0', '1' after 50 ns;
                                wait for 100 ns;
                        else
                                clock_out <= '0';
                                wait until clock_enable = '1';
                        end if;
                end process;
        end clock_driver;

end structure;
```

```
configuration test_bench_behaviour of test_bench is

        for structure
                for counter : counter_16
                        use entity work.counter_16 (behaviour);
                end for;
        end for;

end test_bench_behaviour;



library misc;

configuration test_bench_structure of test_bench is

        for structure       -- of test_bench
                for counter : counter_16
                        use entity work.counter_16 (structure);
                        for structure       -- of counter_16
                                for all : t_flipflop
                                        use entity misc.t_flipflop (behaviour);
                                end for;
                                for all : inverter
                                        use entity misc.inverter (behaviour);
                                end for;
                        end for;
                end for;
        end for;

end test_bench_structure;


entity t_flipflop is
        generic (t : Bit := '1');
        port (clk : in Bit; q : out Bit);
end t_flipflop;

architecture behaviour of t_flipflop is
begin
        if (clk'event) and (clk = '1') then
                q <= not q after 5 ns;
        end if;
end behaviour;



entity inverter is
        port (a : in Bit; y : out Bit);
end inverter;

architecture behaviour of inverter is
begin
        y <= not a after 5 ns;
end behaviour;
```

Függelék

Attribútumok

Típus és altípus attribútumok

T'Base	a T elemi típus típusa (A Base attribútum csak más attribútum elõtagjaként használható.)
T'Left	a T skalár típus elsõ eleme
T'Right	a T skalár típus utolsó eleme
T'High	a T skalár típus felsõ korlátja
T'Low	a T skalár típus alsó korlátja
T'Pos(X)	X poziciója a T diszkrét vagy fizikai típusban
T'Val(X)	a T diszkrét vagy fizikai típus X-edik eleme
T'Succ(X)	az X-et követõ elem poziciója a T diszkrét vagy fizikai típusban
T'Pred(X)	az X-et megelõzõ elem poziciója a T diszkrét vagy fizikai típusban
T'Leftof(X)	a T diszkrét vagy fizikai típus (X–1)-edik eleme
T'Rightof(X)	a T diszkrét vagy fizikai típus (X+1)-edik eleme

Tömb attribútumok

A'Left(N)	az A tömb N-edik eleme típusának elsõ eleme
A'Right(N)	az A tömb N-edik eleme típusának utolsó eleme
A'High(N)	az A tömb N-edik eleme típusának felsõ korlátja
A'Low(N)	az A tömb N-edik eleme típusának alsó korlátja
A'Range(N)	az A tömb N-edik eleme típusának értékkészlete (Ha az értékkészlet növekvõ sorrendû, akkor a visszatérési érték az *elsõ_elem* **to** *utolsó_elem* tartomány, ha csökkenõ sorrendû, akkor az *elsõ_elem* **downto** *utolsó_elem*.)

A'Reverse_Range(N)	az A tömb N-edik eleme típusának fordított értékkészlete (Ha az értékkészlet növekvõ sorrendû, akkor a visszatérési érték az *utolsó_elem* **downto** *elsõ_elem* tartomány, ha csökkenõ sorrendû, akkor az *utolsó_elem* **to** *elsõ_elem*.)
A'Length(N)	az A tömb N-edik elemének elemszáma

Jel attribútumok

S'Delayed(T)	az S jel T idõvel késleltetve
S'Stable(T)	az S jel T ideje nem kapott új értéket (A visszatérési érték Boolean típusú)
S'Quiet(T)	az S jel T ideje nem kapott értéket (A visszatérési érték Boolean típusú)
S'Transaction	az S jel értékváltozásaira billenõ jel (A visszatérési érték egy Bit típusú jel.)
S'Event	az S jel értéke megváltozott
S'Active	az S jelhez új jelalak rendelõdött
S'Last_Event	az S jel legutolsó értékváltozása óta eltelt idõ
S'Last_Active	az S jel legutolsó új jelalak hozzárendelése óta eltelt idõ
S'Last_Value	az S jel elõzõ értéke

Blokk attribútumok

B'Behaviour	a B blokk vagy architektúra csak viselkedési leírást tartalmaz (A visszatérési érték Boolean típusú, és akkor igaz, ha a blokk vagy architektúra nem tartalmaz komponens leképzést.)
B'Structure	a B blokk vagy architektúra csak struktúrális leírást tartalmaz (A vissztérési érték Boolean típusú, és akkor igaz, ha a blokk vagy architektúra nem tartalmaz jel értékadást, tehát minden folyamat passzív.)