

UNIVERSITEIT ANTWERPEN  
Academiejaar 2014-2015

Faculteit Toegepaste Ingenieurswetenschappen

## **2- Basis Elektronica (Practicum Digitale)**

### **VHDL Inleiding**

**Ann Beniest**

# 1. VHDL.

## 1.1. Inleiding

Een digitaal ontwerp heeft tegenwoordig een zodanige grote complexiteit dat het op een hoog niveau moet beschreven worden en vervolgens automatisch moet vertaald worden naar de hardware. We hebben dus een middel nodig om hardware te beschrijven op hoog niveau, dit kan met VHDL.

VHDL staat voor VHSIC Hardware Description Language  
VHSIC = Very High Speed Integrated Circuit

VHDL is een gestructureerde taal die gebruikt wordt om digitale systemen te beschrijven, zowel structureel, in de vorm van een schema, als functioneel, via een gedragsbeschrijving.

Sinds 1980 deed de overheid van de V.S. grote investeringen voor het ontwikkelen van een standaard VHDL-taal. Een werkgroep, bestaande uit werknemers van Intermetrix, IBM en Texas Instruments, ontwikkelde de taal VHDL.

Deze taal werd voor het eerst vastgelegd door de IEEE (The Institute of electrical and electronics engineers) in 1987: VHDL-standaard IEEE-1076.

In 1993 verscheen een tweede versie.

## 1.2. VHDL-ontwerp

### **VHDL-beschrijving:**

Het doel van VHDL is een goede beschrijving te geven van de gewenste hardware. Hierbij kan men gebruik maken van verschillende ontwerpstechnieken zoals top-down aanpak, gebruik van basisbouwblokken die al eerder ontwikkeld werden, ...

### **Functionele simulatie:**

Als de hardware dan duidelijk beschreven is, kan de VHDL ook aangewend worden voor het uittesten van het ontwerp. Dit kan door het schrijven van testbenches waar we de werking van het ontwerp mee kunnen simuleren.

### **Synthese:**

Na het uittesten kan de hardware automatisch gegenereerd worden, dit noemen we synthese. De synthese kan gebeuren naar een IC of een FPGA.

### **Timing verificatie:**

Wanneer er aan de timing niet voldaan is, moet het ontwerp herschreven of aangepast worden. We kunnen ook een andere, snellere component kiezen.

## 1.3. De verschillende abstractieniveaus

Een digitaal ontwerp kunnen we in VHDL beschrijven op vier verschillende niveaus:

**Het hoogste niveau:** gedragsbeschrijving.

Bij simulatie kan het functionele tijdsgedrag gesimuleerd worden: “after 10 ns”.

**Het RTL (register transfer level) -niveau:**

Dit niveau beschrijft het gedrag op basis van bouwblokken, zoals een multiplexer, decoder, vermenigvuldiger, ...

Hier worden processen gebruikt. Er bestaan twee soorten processen:

- een combinatorisch proces
- een register (geklokt) proces

Alle synchrone processen worden opgebouwd met flipflops en worden beschreven met een FSM.

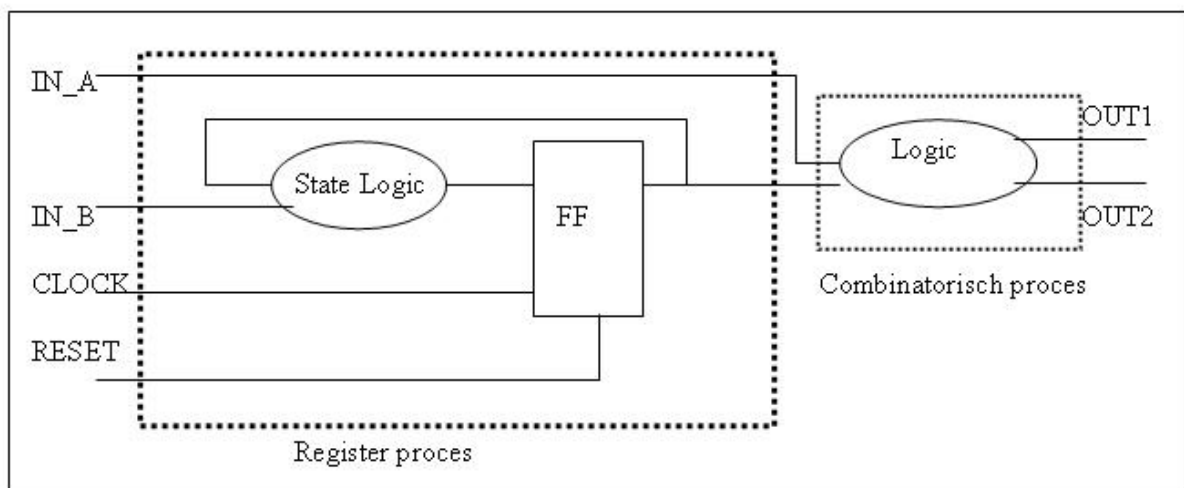


Fig. 1: RT Level in VHDL

De simulatie geeft geen informatie over tijdsvertragingen. Het is dus onmogelijk te voorspellen of alle signalen al dan niet een stabiele toestand bereikt hebben binnen één klokperiode.

**Het logisch niveau :** beschrijft het ontwerp aan de hand van poorten: and, or, nand, ...

Bij simulatie kunnen delays aan de gebruikte poorten toegevoegd worden.

**Het layout niveau:** beschrijft het ontwerp op transistor niveau (zie analoge technieken.)

Hier zijn de lengtes van de verbindingen gekend. (path delays) Bijgevolg zijn de Propagation delays gekend.

Meestal zal men de hardware beschrijven op RTL niveau en het synthese tool een vertaling laten maken naar het laagste niveau.

Het is wel eens nuttig dit lager niveau te bekijken om een inzicht te krijgen in de werking.

Vertrekken van gedrag zou resulteren in een grotere, duurdere component (FPGA) of in grotere oppervlakte (IC). Dit komt omdat het synthese tool veel te veel bits zal nemen om de getallen voor te stellen.

## 1.4. Voordelen van het gebruik van VHDL.

1. De verschillende abstractieniveaus:

Beschrijvingen zijn mogelijk op systeem-, register- en poortniveau. Alle niveaus mogen worden vermengd.

2. Hiërarchie wordt ondersteund.

Als ontwerpmethoden heeft men zowel top-down als bottom-up mogelijkheden.

3. Flexibiliteit:

De VHDL-code kan gebruikt worden op verschillende CAD-programma's.

Verschillende fabrikanten ondersteunen VHDL.

VHDL kan zowel gebruikt worden voor FPGAs, CPLDs als ASIC.

4. VHDL is technologie onafhankelijk:

De code kan gebruikt worden op verschillende componenten en later gemakkelijk herbruikt worden.

Op het laatste moment kan men nog beslissen om van implementatie te veranderen.

5. VHDL is taal gebaseerd:

Ontwerpen gaat sneller dan bij een schematische invoer.

Is ook beter te lezen door derden, laat toe om commentaar bij te voegen.

6. Belangrijk is het dat ook de testomgeving in VHDL is beschreven.

7. VHDL is een standaard.

## 2. Modelvorming in VHDL

### 2.1. Algemeen

Fig. 2. Stelt een digitaal systeem voor. Dit systeem is discreet, in tegenstelling tot analoge, continue systemen. Discrete waarden aan de ingangen worden getransformeerd naar discrete waarden voor de uitgangen: op de ingangswaarden worden een aantal bewerkingen uitgevoerd waarvan de resultaten worden doorgegeven naar de uitgangen.

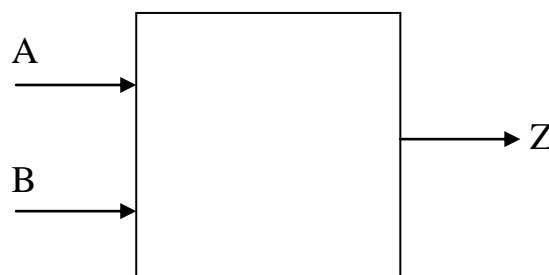


Fig. 2: Digitaal systeem

In VHDL wordt een dergelijk digitaal systeem een **design entity** genoemd.

Als voorbeeld voor dit digitaal systeem nemen we een comparator. De twee ingangen (van 4 bits) worden met elkaar vergeleken. Als ze identiek zijn wordt de uitgang hoog, in het ander geval laag.

De VHDL-beschrijving ziet er als volgt uit:

```
entity comp is
  port( A, B: in bit_vector(3 downto 0);
        Z: out bit
      );
end comp;

architecture gedrag of comp is
begin

  process
  begin
    wait on A,B;

    if A=B then Z<='1';
      else Z<='0';
    end if;

  end process;

end gedrag;
```

Code 1: Voorbeeld van een VHDL-beschrijving.

Deze design entity bestaat uit twee delen:

Het eerste deel is **de entity declaratie**. Hierin vinden we de interface van dit bouwblok naar de buitenwereld. In dit voorbeeld kreeg de entity de naam “comp”.

Het tweede deel is **de architecture body**. Hierin wordt het gedrag beschreven van de entity. Welke transformaties de ingangen ondergaan en daarna aan de uitgangen worden aangeboden. We vermelden steeds met welke specifieke entity deze body moet verbonden worden. Een enkele entity kan meerdere architecturen hebben.

Deze body kan op drie manieren beschreven worden:

1. **Functionele beschrijving:** dit gebeurt in de vorm van een sequentiële lijst van programmabevelen of commando's.
2. **Structuurbeschrijving:** de architectuur van een circuit wordt beschreven als een hiërarchische samenstelling van onderling verbonden deelsystemen of componenten.
3. **Data\_flowbeschrijving:** hier beschrijft VHDL een reeks parallele (concurrent) waardetoekenningen aan signalen.

Deze drie methoden kunnen door elkaar gebruikt worden.

## 2.2. De entity declaratie.

Als voorbeeld nemen we een volledige opteller:

Het digitaal systeem noemen we V\_OPTELLER en heeft drie ingangen ( A, B en CI) en twee uitgangen( S en CO).

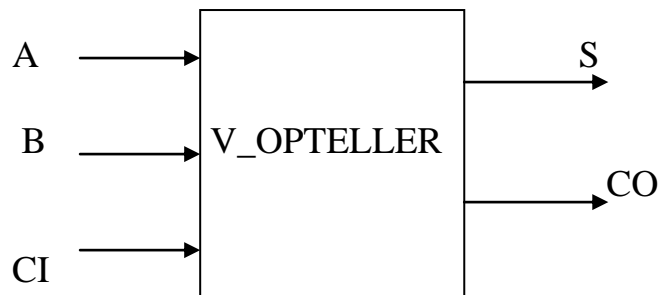


Fig.3: Een volledige opteller.

A	B	CI	CO	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Fig.4: functietabel

Hierbij hoort de volgende entity declaratie:

```
entity V_OPTELLER is
port(  A: in bit;
       B: in bit;
       CI: in bit;
       S: out bit;
       CO: out bit
);
end V_OPTELLER;
```

Code 2: Voorbeeld van een Entity declaratie.

Een entity geeft de interface van het bouwblok weer met de buitenwereld. Hoe dit blok opgebouwd is, is voor de entity niet belangrijk.

Binnen de entity staat steeds een **port statement**, waarin de verschillende signalen die het bouwblok verbinden met de buitenwereld vermeld staan.

Elk signaal heeft:

- Een eigen naam.
- Een richting. Deze richting kan
  - “in” zijn: het is een signaal dat binnenkomt.
  - “out” zijn: het is een signaal dat naar buiten gaat. Intern mag dit signaal NIET gebruikt worden als input van een ander blok.
  - “buffer” zijn: Het is een signaal dat naar buiten gaat maar intern nog kan hergebruikt worden in een ander blok.
  - “inout” zijn: Dit is een signaal dat zowel kan binnenkomen als naar buiten gaan. Het is best deze enkel te gebruiken voor echte bi-directionele signalen.

Het is belangrijk het juiste type van de richting te kiezen. Wanneer een signaal inout is moet de hardware een tristate buffer voorzien die hoogimpedant is als de pin een ingang is en die aangedreven wordt als de pin een uitgang is. Dit vereist dus meer hardware.

- Een data type.

## 2.3. Architecture body.

Zoals hierboven vermeld, kan de architectuur op drie verschillende wijzen worden beschreven.

### 2.3.1 Gedragsbeschrijving of functionele beschrijving

In code 3 wordt de architectuur van de hierboven beschreven volledige opteller gegeven in de vorm van een gedragsbeschrijving.

```

architecture gedrag of V_OPTELLER is
begin

    process

        variable N: integer;
        constant SUM: bit_vector(0 to 3) := "0101";
        constant CARRY: bit_vector(0 to 3) := "0011";

        begin

            wait on A, B, CI;
            N:=0;
            if A='1' then N:=N+1; end if;
            if B='1' then N:=N+1; end if;
            if CI='1' then N:=N+1; end if;
            S <= SUM(N) after 6 ns;
            CO <= CARRY(N) after 9 ns;

        end process;

    end gedrag;

```

Code 3: Gedragsbeschrijving van de volledige opteller.

Deze code geeft het volgende tijdsdiagram:

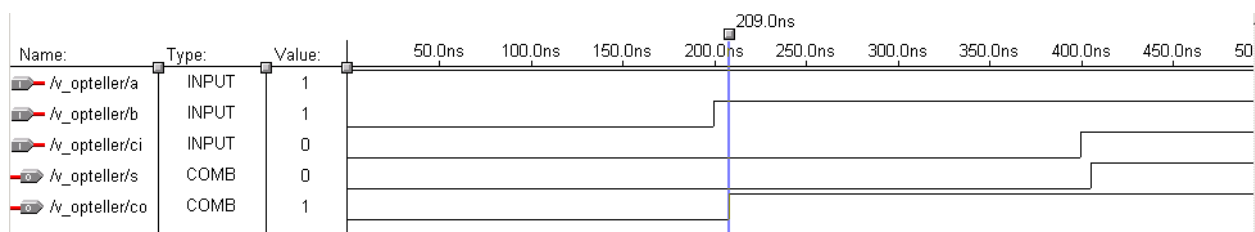


Fig. 5: tijdsdiagram van de volledige opteller

De basis waarop een gedragsbeschrijving steunt is het “process”. Elke transformatie die in een digitaal systeem plaatsvindt van de ingangen naar uitgangen, wordt beschreven in een dergelijk process.

Een process lijkt op een programma. Het is samengesteld uit sequentiële commando’s, die in de gegeven volgorde worden uitgevoerd en waarbij subprogramma’s kunnen worden opgeroepen. In VHDL worden alle processen parallel ( concurrent ) uitgevoerd.

Een VHDL-eenheid is een geheel van onafhankelijke processen die los van elkaar worden uitgevoerd.



De signalen zorgen voor de communicatie. Ze vormen datapaden tussen de processen. Aan elk datapad wordt een bepaald datatype geassocieerd (bit, bit\_vector, integer, real, ...).

Processen worden constant uitgevoerd tot ze worden gestopt. Een beëindigd proces kan opnieuw worden geactiveerd. Zo kunnen processen gevoelig worden gemaakt voor waardeveranderingen (events) op bepaalde datapaden.

Als er een waardeverandering optreedt, dan wordt het proces opnieuw geactiveerd.

Dit kan worden gerealiseerd door een WAIT-commando (vb. WAIT ON A, B, CI; ) of door een event-lijst bij het proces te voegen (vb. Process (A,B,CI) ).

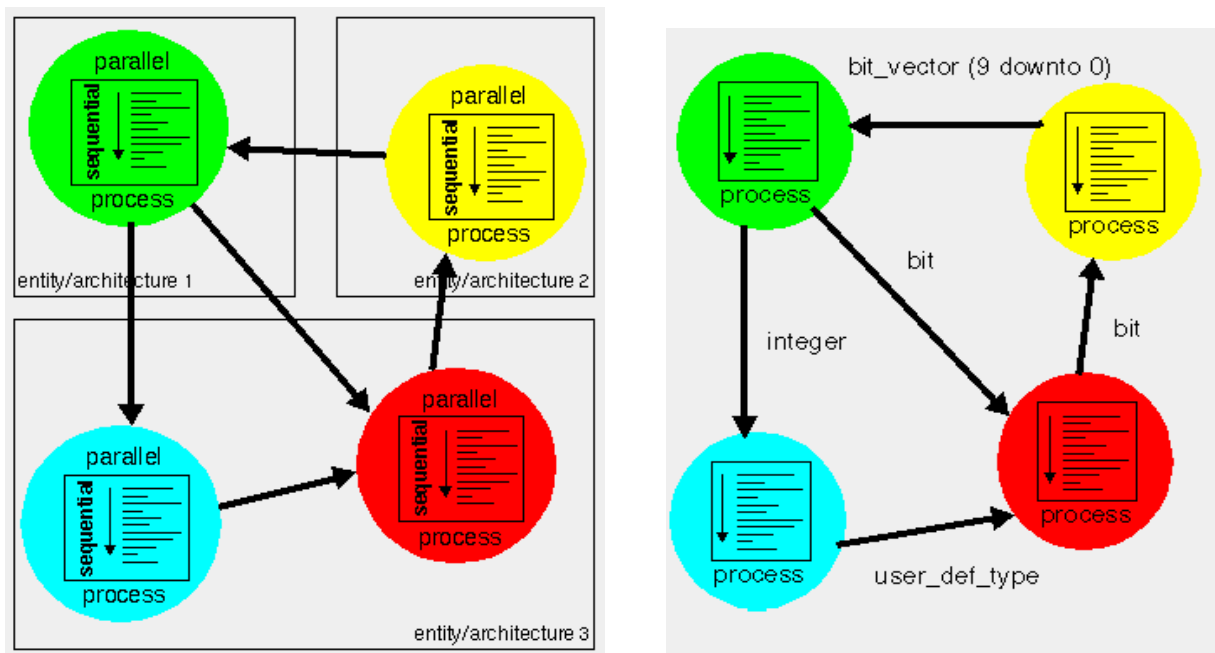


Fig. 6: Voorstelling parallelle processen

### 2.3.2. Structuurbeschrijving.

VHDL biedt de mogelijkheden om hiërarchie in te bouwen. Zo kunnen we een bouwblok volledige opteller maken op basis van twee halve opteller bouwblokken en een or-poort met twee ingangen.

De architectuur van de volledige opteller, volgens de structuur van fig.7, wordt gegeven in code 4.

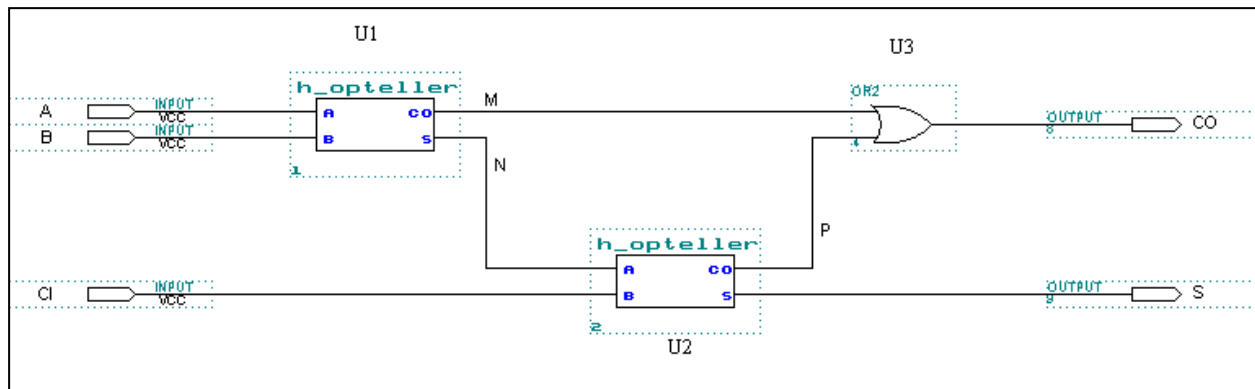


Fig. 7: Structuur van de volledige opteller.

Bij deze structuurbeschrijving hoort dezelfde entity als bij de gedragsbeschrijving zie code 2. Hierin werden de buitenste datapaden (A,B,CI,S,CO) gedeclareerd.

De architecture body bestaat uit een opsomming van onderling verbonden componenten. Voor dat ze worden gebruikt als subsystemen worden deze componenten eerst gedeclareerd. Voor elke gebruikte component moet er een design entity bestaan, bijvoorbeeld in een bibliotheek. De port-lijst van de componenten in het declaratiegedeelte (bvb: A, B, Y voor de component or2) moet overeenkomen met de port-lijst in de entity declaratie van de design entity van de component, dit wat betreft aantal ports, type en volgorde.

Interne signalen (M,N,P) vormen de datapaden tussen de componenten.

Een port is een uitwendig signaal van een bepaalde component. Deze ports worden verbonden met de datapaden. De juiste verbinding wordt bepaald door een **port map**.

**architecture structuur of v\_opteller is**

```
component h_opteller
  port(A, B: in bit;
        C, S: out bit);
end component;
```

```
component or2
  port( A,B: in bit;
        Y: out bit);
end component;
```

```
signal M,N,P: bit;
begin
  U1: h_opteller port map( A, B, M, N );
  U2: h_opteller port map( N, CI, P, S );
  U3: or2 port map (M, P, CO);
end structuur;
```

Code 4: Structuurbeschrijving van de volledige opteller

Bij “**positie-associatie**” die gebruikt werd in code 4, is de rangschikking van de signalen van zeer groot belang!

```
U1: h_opteller port map( A, B, M, N );
```

We kunnen echter ook gebruik maken van “**naam-associatie**”, hier is de rangschikking van geen belang.

```
U1: h_opteller port map( A => A,  
                        S=> N,  
                        B=> B,  
                        C=> M );
```

### 2.3.3. Data-flowbeschrijving.

In code 5 wordt de architectuur van de volledige opteller gegeven volgens het dataflowprincipe. Ook bij deze data-flowbeschrijving hoort de in code 2 gegeven entity declaratie.

De werking wordt beschreven door een lijst van parallelle (concurrent) waardetoekenningen aan signalen.

```
architecture data_flow of v_opteller is  
    signal N: bit;  
    begin  
        N <= A xor B after 3 ns;  
        S <= N xor CI after 3ns;  
        CO <= ( A and B) or (N and CI) after 6ns;  
    End data_flow;
```

Code 5: Data-flowbeschrijving van de volledige opteller.

Alle interne objecten( signalen, constanten, ...) zoals N, moeten eerst worden gedeclareerd. Het commando  $N \leq A \text{ xor } B \text{ after } 3\text{ns}$ ; is een concurrent waardetoekenning aan het signaal N. De volgorde waarin deze commando's worden gegeven heeft geen belang. Zo kan eerst het commando  $S \leq N \text{ xor } CI \text{ after } 3\text{ns}$ ; worden gegeven en dan pas  $N \leq A \text{ xor } B \text{ after } 3\text{ns}$ ;

Een concurrent waardetoekenning aan een signaal kan worden beschouwd als een verkorte notatie voor een proces. Zo is

```
N <= A xor B after 3ns; (1)
```

equivalent met het volgende proces:

```

process
  begin
    N<= A xor B after 3ns;
    wait on A, B;
  end process;

```

(2)

Of

```

process( A, B )
  begin
    N <= A xor B after 3ns;
  end process;

```

(3)

Een concurrent commando wordt alleen maar geactiveerd als aan één van de signalen uit het rechterlid, bijvoorbeeld A of B in commando (1), een nieuwe waarde wordt toegewezen. Men noemt zulk commando **event driven**.

In een proces moet de ontwerper het event-driven karakter van het model programmeren met een WAIT-commando, zoals in het proces (2) of met een sensitiviteitslist zoals in het proces (3).

## 3. De VHDL-taal en syntax.

In dit hoofdstuk worden de syntaxregels van VHDL aan de hand van voorbeelden uitgelegd. Voor alle bijzonderheden verwijzen we naar de VHDL Standard Language Reference Manual.

### 3.1. Lexicale conventies

#### 3.1.1. Typografische tekens

Een VHDL –tekst mag alleen maar bestaan uit de volgende tekens:

- hoofdletters A...Z
- kleine letters: a...z
- cijfers: 0...9
- spatie
- speciale tekens; “’#()\*+,-./:;<=>\_|
- controletekens: TAB CR LF FF

#### 3.1.2. Namen

VHDL is “case insensitive” d.w.z. Er wordt geen onderscheid gemaakt tussen kleine letters en hoofdletters.

Namen mogen letters, cijfers en underscore bevatten. Ze moeten met een letter beginnen. Men mag geen gereserveerde woorden gebruiken zoals:

abs , access , after , alias , all , and , architecture , array , assert , attribute  
 begin, block , body , buffer , bus ,  
 case , component , configuration , constant  
 disconnect , downto  
 else , elsif , end , entity , exit  
 file , for , function  
 generate , generic , guarded  
 if , in , inout , is  
 label , library , linkage , loop  
 map , mod  
 nand , new , next , nor , not , null  
 of , on , open , or , others , out  
 package , port , procedure , process  
 range , record , register , rem , report , return  
 select , severity , signal , subtype  
 then , to , transport , type  
 units , until , use  
 variable  
 wait , when , while , with  
 xor

Deze woorden mogen enkel in hun specifieke betekenis gebruikt worden.

### 3.1.3. Constanten

Constanten worden genoteerd zoals in bekende hogere programmeertalen.

#### 3.1.3.1. Decimale getallen

Gehele getallen:	12	0	13476
Reële getallen:	12.0	0.0	0.456
Exponentiële schrijfwijze:	1.34E-12	17E6	

#### 3.1.3.2 Getallen met een ander grondtal

Het grondtal (radix) is een geheel getal tussen 2 en 16.

Gehele getallen:	2#11111111#	(= $11111111_2 = 255_{10}$ )
	16#FF#	(= $FF_{16}$ )
Reële getallen:	16#F.FF#E2	(= $F.FF_{16} \times 16^2$ )
	2#1.1101#E+11	(= $1.1101_2 \times 2^{11}$ )

#### 3.1.3.3. Karakters

Een ‘constant’ teken staat tussen enkele aanhalingstekens.

‘A’                      ‘Z’                      ‘#’

#### 3.1.3.4. Constante ‘strings’

Constante strings staan tussen dubbele aanhalingstekens.

“de setup- en hold tijd waren te kort”

#### 3.1.3.5. Bitstring-constanten

Een bitsring-constante is een rij bestaande uit nullen en enen (type BIT). De rij wordt tussen dubbele aanhalingstekens geplaatst en voorafgegaan door een codeletter die de gebruikte radix aangeeft: B voor binair, O voor octaal en X voor hexadecimaal.

X”FFF” = B”1111111111”

O”707” = B”111000111”

### 3.1.3.6. Fysische constanten

Fysische constanten bestaan uit een gehele of reële constante, gevolgd door een eenheid.

2.3sec          10Kohm

### 3.1.4. Commentaar

Commentaar begint met twee koppeltekens en eindigt op het einde van de regel.

```
-- nu volgt het hoofdproces
```

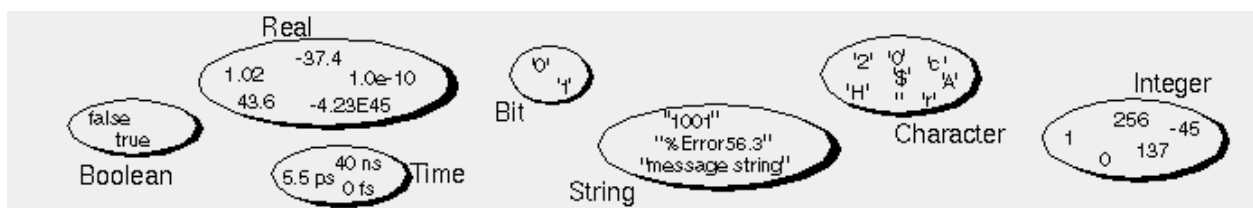
## 3.2. Datatypes

Elke grootheid die een bepaalde waarde kan aannemen (vb. een variable of een signaal ) wordt een object genoemd. In VHDL is elk object van een bepaald type. Het type geeft aan welk soort waarden het object kan aannemen en welke bewerkingen ermee mogen uitgevoerd worden. Het type moet vastgelegd worden wanneer het signaal gedeclareerd wordt dit kan in de entity als port of in de architectuur als een intern signaal.

VHDL heeft een aantal voorgedefinieerde types die in een package STANDARD zijn gedeclareerd.

```
package STANDARD is
  type BOOLEAN is (FALSE,TRUE);
  type BIT is ('0','1');
  type CHARACTER is (--ascii set);
  type INTEGER is range -- implementation_defined ;
  type REAL is range --implementation_defined ;

  --BIT_VECTOR, STRING, TIME
end STANDARD;
```



De gebruiker kan ook zelf nieuwe types invoeren. De declaratie gebeurt op de volgende manier:

**type TYPENAAM is TYPEDEFENITIE;**

Er bestaan vier soorten types: scalaire types, samengestelde types, access-types en file-types.

De verzameling van toegelaten waarden voor een object van een bepaald type, kan aan een voorwaarde( beperking )worden onderworpen. Een **subtype** is een type met een dergelijke beperking. Alle bewerkingen die toegelaten zijn op objecten van een bepaald type zijn ook toepasselijk op de subtypes. Een waardetoekenningcommando kan echter alleen maar waarden van het juiste subtype toekennen.

### 3.2.1. Het scalaire type

Tot de scalaire types horen:

- opsommingstypes (enumeration types)
- geheel-getal types (integer types)
- fysische types
- bewegende-kommatypes (floating point types)

Alle scalaire types zijn geordend: voor scalaire waarden zijn alle relationele operatoren (zoals < > = ) voorgedefinieerd. Alle waarden van een discreet of een fysisch type hebben een volgnummer.

#### 3.2.1.1. Het opsommingstype (Enumeration type)

Het enumeration type wordt gedefinieerd door de mogelijke waarden op te sommen. De lijst van waarden worden tussen ronde haakjes geplaatst. De waarden mogen namen of tekenconstanten zijn:

```

type VIJFWAARDIG is ('0','1','U','D','X');
    --dit is een opsomming van tekenconstanten
type SPANNINGSTOESTAND is ( LOW, HIGH, UNKNOWN );
    --dit is een opsomming van namen
type STATE is ( RESET, START, COUNT, STOP );
    --dit is een opsomming van toestanden bij een FSM
type BIT is ('0','1')
    --type bit is voorgedefinieerd
  
```

De volgorde in de opsomming bepaalt de **rangorde**: de eerst vermelde waarde krijgt volgnummer 0.



De voorgedefinieerde enumeration types zijn : CHARACTER, BIT, BOOLEAN en SEVERITY\_LEVEL.

### 3.2.1.2. Het geheel-getaltype (integer type)

We geven enkele voorbeelden:

```
type TWEE_COMPLEMENTGETAL is range -32768 to 32767;
type BYTE_BEREIK is range 0 to 255;
subtype HOOGSTE_BIT_LAAG is BYTE_BEREIK range 0 to 127;
type WOORD_INDEX is range 31 downto 0;
```

Een integer-typedefinitie beperkt de mogelijke waarden tot gehele getallen binnen een bepaald bereik (range). Een bijkomende beperking bepaalt een **subtype**. Het bereik van het voorgedefinieerd type INTEGER hangt af van de software-uitvoering. Het volgnummer van een waarde is hier gelijk aan de getalwaarde zelf.

Een **rangebeperking** kan opgegeven worden in stijgende of in dalende volgorde (respectievelijk met de sleutelwoorden TO of DOWNTO). Een range bepaalt dus een deelverzameling. Deze deelverzameling kan leeg zijn, bijvoorbeeld indien bij een stijgende volgorde de ondergrens groter is dan de bovengrens.

### 3.2.1.3. Het fysische type

De waarden van een fysische type stellen hoeveelheden voor van een bepaalde grootheid, met vermelding van de eenheid. Elke fysische waarde is een veelvoud van de basiseenheid, zoals in het volgende voorbeeld:

```
Type WEERSTAND is range 1 to 10E9
    units OHM;                --dit is de basiseenheid
    KOHM = 100 OHM;          --afgeleide eenheid
    end units;

signal R1: WEERSTAND; .....
    .....
    .....

R1<= R1 + 4 KOHM;
```

De toegelaten waarden worden nu opgegeven door een rangebeperking. De typedeclaratie moet nu ook een eenheidsdefinitie bevatten, eventueel gevolgd door afgeleide eenheden (schaalfactoren).

Het enige voorgedefinieerde fysische type is **TIME**. De basis eenheid is femtoseconde. Andere eenheden zijn : ps, ns, us, ms, sec, min en hr.

Dit datatype bestaat uit een numerieke waarde en een eenheid. Het wordt gebruikt om het uitvoeren van commando's te vertragen met een bepaalde tijdseenheid bijvoorbeeld in testbenches (zie par. 4.3. ) of bij de architecture dataflowbeschrijving ( zie par. 2.3.3.). Signalen van het type 'time' kunnen vermenigvuldigd of gedeeld worden door integer of real waarden. De resultaten van deze bewerkingen zijn terug van het type 'time'.

Voorbeeld:

```

.....
constant period := 50 ns;

begin
  process
    begin
      wait for 50 ns;
      .....
      wait for period;
      .....
      wait for 5 * period;
      .....
    end process;
  .....

  clk <= not clk after period/2;
  .....

```

#### 3.2.1.4. Het bewegende-kommatype (floating point type)

Ook een bewegende-kommatype wordt gedeclareerd met een rangebeperving, zoals in het volgende voorbeeld:

```

type FRACTIE is range 0.1E-10 to 0.1 E-1;

```

Het enige voorgedefinieerde floating-point-type is **REAL**. Het bereik van dit type is ook afhankelijk van de software-uitvoering.

#### 3.2.2. Het samengestelde type

Samengestelde types worden gebruikt om een verzameling van waarden te definiëren. VHDL kent twee samengestelde types: de **ARRAY** en de **RECORD**.

Een array is samengesteld uit elementen van hetzelfde type.

De waarden in een record mogen van een verschillend type zijn.

### 3.2.2.1. Het array-type

Arrays kunnen één of meerdere dimensies hebben. Elke dimensie heeft een type dat discreet moet zijn. De dimensiedeclaratie kan begrensd of onbegrensd zijn, zoals blijkt uit de voorbeelden:

```
--begrenste dimensiedeclaratie
type WOORD is array( 15 downto 1 ) of BIT;
type MATRIX is array(1 to 80, 1 to 24) of BOOLEAN;

--onbegrenste dimensiedeclaratie
type SCHERM is array (INTEGER range <>, INTEGER range <>) of PIXEL;
```

Bij de onbegrenste dimensiedeclaratie wordt wel het type van de dimensie opgegeven, maar niet het bereik. Dit wordt aangegeven door '**range <>**'.

Er bestaan twee voorgedefinieerde arraytypes: BIT\_VECTOR en STRING. Deze zijn onbegrensd. De grootte, het aantal elementen wordt tijdens de signal/port declaratie gedefinieerd. De BIT\_VECTOR is een array bestaande uit elementen van het type BIT en de STRING is een array bestaande uit elementen van het type CHAR.

Voorbeeld:

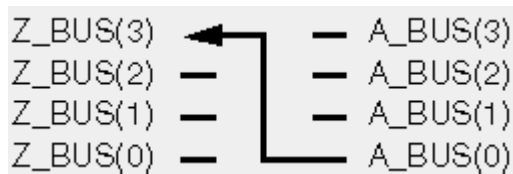
```
variable C: string := "dit is een tekst";
signal A_BUS, Z_BUS: bit_vector( 3 downto 0 );
```

```
Z_BUS <= A_BUS;
```



Er kan ook één element geselecteerd worden uit de bit-vector zoals blijkt uit volgend voorbeeld:

```
Z_BUS(3) <= A_BUS(0);
```



Hier volgen twee voorbeelden van architecturen die gebruik maken van ‘bit’ en ‘integer’ en na synthese hetzelfde resultaat geven:

<pre>architecture EXAMPLE_1 of DATATYPES is     signal SEL : <b>bit</b>;     signal A, B, Z : <b>integer</b> range 0 to 3;  begin     A &lt;= 2;     B &lt;= 3;      process(SEL,A,B)     begin         if SEL = '1' then             Z &lt;= A;         else             Z &lt;= B;         end if;     end process; end EXAMPLE_1;</pre>	<b>OR:</b>	<pre>architecture EXAMPLE_2 of DATATYPES is     signal SEL : <b>bit</b>;     signal A, B, Z : <b>bit_vector</b> (1 downto 0);  begin     A &lt;= "10";     B &lt;= "11";      process(SEL,A,B)     begin         if SEL = '1' then             Z &lt;= A;         else             Z &lt;= B;         end if;     end process; end EXAMPLE_2;</pre>
--	------------	---

#### Opmerking:

Uit het volgende voorbeeld blijkt dat de positie en niet de nummering van de elementen van belang is. De richting van de arrays moet steeds hetzelfde gedefinieerd zijn!

<pre>architecture EXAMPLE of ARRAYS is     signal Z_BUS : bit_vector (3 downto 0);     signal C_BUS : bit_vector (0 to 3); begin     Z_BUS &lt;= C_BUS; end EXAMPLE;</pre>	
--	--

#### 3.2.2.2. Het record-type

Een record-type is heterogeen samengesteld, zoals blijkt uit het volgende voorbeeld:

```
type DATUM is
    record
        DAG: INTEGER range 1 to 31;
        MAAND: INTEGER range 1 to 12;
        JAAR: INTEGER range 0 to 4000;
    end record;
```

Elke waarde van het type DATUM bestaat uit drie elementen of velden. Van elk record-veld wordt de naam en het type gegeven.

Toegang tot een element van het record:

```
signal X: DATUM;
.....
X.MAAND<=6;
```

### 3.2.3. Concatenatie

Bij een signaaltoekenning is het van belang dat de datatypes aan de beide zijden van de operator identiek zijn. Daarom is het soms noodzakelijk om een array aan te passen. De concatenatie operator ‘&’ groepeerde de elementen ( die enkel van hetzelfde datatype moeten zijn ) aan zijn kant. De positie van de elementen in de arrays zijn van belang!

De concatenatie operator mag alleen langs de rechter kant van een toekenning operator ‘<=’ gebruikt worden.

Voorbeeld:

<pre>architecture EXAMPLE_1 of CONCATENATION is   signal BYTE : bit_vector (7 downto 0);   signal A_BUS, B_BUS : bit_vector (3 downto 0); begin   BYTE &lt;= A_BUS &amp; B_BUS; end EXAMPLE_1;</pre>	
<pre>architecture EXAMPLE_2 of CONCATENATION is   signal Z_BUS : bit_vector (3 downto 0);   signal A_BIT, B_BIT, C_BIT, D_BIT : bit; begin   Z_BUS &lt;= A_BIT &amp; B_BIT &amp; C_BIT &amp; D_BIT; end EXAMPLE_2;</pre>	

### 3.2.4. Aggregaten

Een andere manier om waarden toe te kennen aan samengestelde types is gebruik maken van aggregaten. Een aggregaat bestaat uit een lijst (tussen ronde haken) van waarden voor de elementen van een array of van een record.

Voorbeeld:

architecture example of aggregates is

```

signal TOT_BYTE : bit_vector( 7 downto 0 );
signal Z_BUS: bit_vector( 3 downto 0 );
signal A_BIT, B_BIT, C_BIT, D_BIT : bit;

begin
  Z_BUS <= ( A_BIT, B_BIT, C_BIT, D_BIT );           (1)

  (A_BIT, B_BIT, C_BIT, D_BIT) <= bit_vector("1011");

  (A_BIT, B_BIT, C_BIT, D_BIT) <= TOT_BYTE( 3 downto 0 );

  TOT_BYTE <= ( 7 => '1', 5 downto 1 => '1', 6 => B_BIT, others => '0' );  (2)
end example;

```

In voorbeeld (1) is gebruik gemaakt van “**plaatsassociatie**” : het eerste element van de array krijgt de eerste waarde van het aggregaat, enz. Zo heeft het element Z\_BUS(0) de waarde van D\_BIT.

Er is ook een “**naamassociatie**” mogelijk, zoals blijkt uit voorbeeld (2).

Aggregaten mogen langs de beide kanten van een toekenningbevel staan. Het keyword ‘**others**’ selecteert alle overgebleven elementen.

### 3.2.5. Partities van arrays

Het tegenovergestelde van een concatenatie en aggregatie is het selecteren van delen van de arrays.

Voorbeeld:

```

architecture example of partitie is
signal TOT_BYTE: bit_vector( 7 downto 0 );
signal A_BUS, Z_BUS: bit_vector( 3 downto 0 );
signal A_BIT: bit;
begin
  TOT_BYTE( 5 downto 2 ) <= A_BUS;
  Z_BUS( 1 downto 0 ) <= '0' & A_BIT;
  A_BIT <= A_BUS(0);
end example;

```

### 3.2.6. Het access-type

Het dynamische alloceren, en het vrijgeven, van geheugen ruimte tijdens de uitvoering van een beschrijving is mogelijk in VHDL. Daarvoor heeft VHDL het access type (bij hogere programmeertalen **pointertype** genoemd ). In VHDL kan dit bijvoorbeeld worden toegepast om lijsten te modelleren. Ook kan het handig worden gebruikt om een groot geheugen te modelleren, vooral als slechts een klein deel van het geheugen wordt gebruikt.

Voorbeeld:

```
type mem;
type pointer is access mem;
type mem is
  record
    address: natural;
    content: integer;
    nxt: pointer;
  end record;
```

Voor meer details verwijzen we naar de IEEE-standaard.

### 3.2.7. Het bestandstype ( file type)

Bestandstypes worden vooral gebruikt om informatie te lezen of te schrijven in bestanden in de omgeving van het computerbestuurssysteem. Een file bestaat uit een sequentiële lijst van waarden van eenzelfde type:

```
Type BESTAND is file of ELEMENT;
F: BESTAND;
```

In dit voorbeeld is F een bestand dat bestaat uit een lijst van waarden van het type ELEMENT. Op dergelijke files zijn drie bewerkingen toegelaten:

- lezen via de procedure READ
- schrijven via de procedure WRITE
- de functie ENDFILE die de waarde TRUE geeft als het einde van het bestand bereikt is, anders FALSE.

Een voorgedefinieerd pakket TEXTIO maakt lezen en schrijven van ASCII-bestanden mogelijk. (zie par. 4.4.)

## 3.3. Extended Data Types



### 3.3.1. IEEE Standard Logic Type

Het type 'bit' heeft alleen de waarden '0' en '1' en is bij het begin van elke simulatie steeds geïnitieerd op '0'. Dit maakt het onmogelijk te testen of bij reset de signalen de juiste waarden krijgen toegekend.

Voor simulatie en synthese hebben we dus bijkomende waarden nodig:

- uninitialized
- high impedance
- undefined
- 'don't care'
- different driver strengths

Dit resulteerde in de IEEE Standard Logic Type:

<b>Type STD_ULOGIC IS(</b>		
'U',	--uninitialized	bv. Een FF waar nog nooit een waarde in geschreven is
'X',	--strong 0 or 1 (=unknown)	
'0',	--strong 0	 bv. direct aan een voeding
'1',	--strong 1	
'Z',	--high impedance	
'W',	--weak 0 or 1 (=unknown)	
'L',	--weak 0	 bv. via weerstand aan de voeding
'H',	--weak 1	
'-');	--don't care	

Dit data type werd gedefinieerd in de package '**IEEE.std\_logic\_1164**' .

Vooraan in de VHDL beschrijving plaatst men:   **library ieee;**  
   **use ieee.std\_logic\_1164.all;**

Er bestaat ook een gelijkaardig data type '**std\_logic**' met dezelfde waarden.

Ook array types zijn beschikbaar: '**std\_logic\_vector**' en '**std\_ulogic\_vector**'

Het voordeel van '**std\_ulogic**' en '**std\_ulogic\_vector**' is dat er errors worden gegenereerd indien er meerdere concurrent signaaltoekenningen optreden.

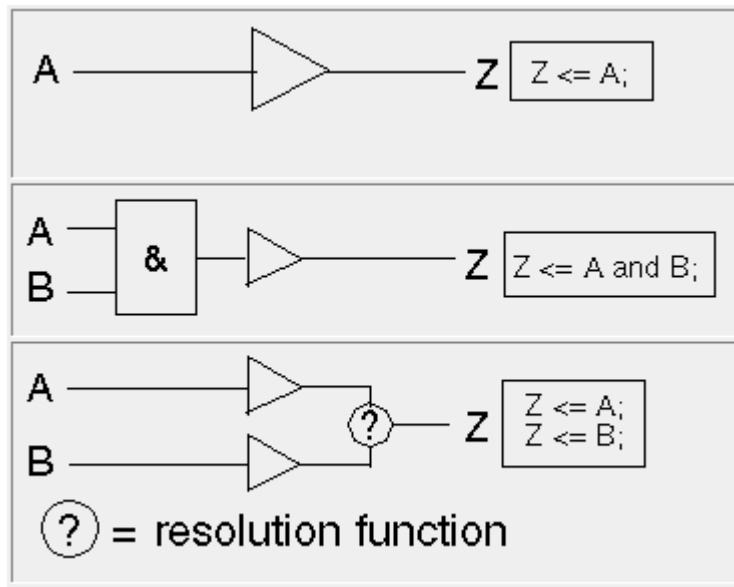
Het voordeel van '**std\_logic**' en '**std\_logic\_vector**' is dat het tri-state bussen toelaat. Ze worden meestal aanbevolen bij een RTL ontwerp.



### 3.3.2. Resolved en unresolved types

Ieder proces waarin een toekenning aan een signaal wordt gedaan creëert één driver voor dat signaal. Ook al zijn er meerdere toekenningen binnen een proces aan hetzelfde signaal, er wordt toch maar één driver gecreëerd.

Hebben we echter meerdere processen met toekenningen aan hetzelfde signaal, dan wordt er in elk proces een driver gecreëerd. Hierdoor wordt de waarde van het signaal onbepaald. VHDL zal dit ook niet accepteren.



Toch zijn er situaties waarin het wenselijk is dat één signaal meerdere sources heeft, bijvoorbeeld voor het beschrijven van een wired-or of het beschrijven van een bidirectionele bus. Door middel van een 'resolved signal' is dit probleem op te lossen.

Een signaal met meerdere sources moet een **resolutiefunctie** hebben. Een resolutiefunctie verzamelt de waarden van deze sources van een signaal en bepaalt op basis hiervan een resulterende waarde voor het signaal.

Het resultaat wordt als volgt bepaald:

Het huidige resultaat selecteert de rij van de resolutie-tabel en de waarde van de volgende driver selecteert de kolom, dit geeft de uiteindelijke signaal waarde.

```

FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic IS
  CONSTANT resolution_table : std_logic_table := (
    --
    --   U   X   0   1   Z   W   L   H   -
    --
    ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- U
    ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- X
    ( 'U', 'X', '0', 'x', '0', '0', '0', '0', 'X' ), -- 0
    ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- 1
    ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- Z
    ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- W
    ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- L
    ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- H
    ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- - );
  VARIABLE result : std_ulogic := 'Z'; -- weakest state default
BEGIN
  IF (s'LENGTH = 1) THEN
    RETURN s(s'LOW);
  ELSE
    FOR i IN s'RANGE LOOP
      result := resolution_table(result, s(i));
    END LOOP;
  END IF;
  RETURN result;
END resolved;

```

### 3.3.3. Subtypes en resolutiefuncties

Een subtype is een type met een beperking. Er bestaan twee soorten beperkingen, **range-** en **indexbeperkingen**:

```

--rangebeperking
subtype LETTERS is CHARACTER range 'a' to 'z';

--indexbeperking
subtype REGISTER is BIT_VECTOR (7 downto 0);

```

Een subtypedeclaratie kan ook de naam van een functie bevatten. Het resultaat van de functie moet tot het subtype behoren. De functie wordt dan resolutiefunctie genoemd. Ze moet aan twee voorwaarden voldoen:

1. De resolutiefunctie heeft één parameter, die als type een onbegrensde array heeft.
2. Het type van het resultaat dat de resolutiefunctie geeft, is hetzelfde als het type van de elementen van de array.

Zoals vroeger al vermeld werd, laat een subtypedeclaratie met een resolutiefunctie toe om resolved signalen te declareren. Een resolved signaal kan meerdere bronnen (drivers) hebben: de resolutiefunctie bepaalt de resulterende waarde van het signaal. Als voorbeeld geven we de volgende declaraties:

```

type BASISLOGICA is ('0','1','Z');

type INTERACTIETABEL is array( INTEGER range <>) of BASISLOGICA;

--declaratie resolutiefunctie:
function MEER_BRONNEN(P:INTERACTIETABEL) return BASISLOGICA;

--declaratie subtype
subtype LOGICA is MEER_BRONNEN BASISLOGICA;

--declaratie van een resolved signaal
signal S:LOGICA;

```

## 3.4. Objecten in VHDL

In VHDL bestaan er drie klassen van objecten: signalen, variabelen en constanten.

Signalen en variabelen kunnen steeds nieuwe waarden aannemen. Een constante krijgt slechts éénmaal, bij declaratie, een waarde.

Een variable neemt een nieuw toegewezen waarde direct aan; bij een signaal kan dit met een zekere tijdsvertraging gebeuren. Signalen stellen verbindingen voor, variabelen echter hebben geen hardware-equivalent.

In VHDL moet elk object eenduidig getypeerd worden bij de declaratie. We geven enkele voorbeelden:

```

constant ROM_SIZE: INTEGER := 16#FFFF#;
variable ADRES: INTEGER range 0 to ROM_SIZE;
signal CLK, CLEAR: BIT ;
signal ENABLE : BIT:= '0';

```

De notatie `:= <expressie>` laat toe een “**beginwaarde**” op te geven.

## 3.5. Packages

Een package is een verzameling van voorgedefinieerde datatypes, subprogramma's, constanten enz...Dit is vooral nuttig als we met een aantal mensen aan eenzelfde project werken. Packages bestaan uit een package-declaratie en een package body.

### 3.5.1. Package declaratie

De algemene vorm van een package-declaratie is:

```
package PACKAGE_PROJECT is
    --constants
    --data types
    --components
    --subprogramma's declaratie
end PACKAGE_PROJECT;
```

### 3.5.2. Package body

De algemene vorm van een package body is:

```
package body PACKAGE_PROJECT is
    --
    --basisdeclaraties
    --bodies van de subprogramma's
    --
end PACKAGE_PROJECT;
```

De package body bevat de subprogram bodies van de subprogram-declaraties die in de package-declaratie vermeld staan. De package-declaratie bevat de publieke, toegankelijke declaraties. De declaraties uit de package body daarentegen zijn niet beschikbaar voor andere VHDL-onderdelen. De items uit de package-declaratie komen ter beschikking van andere VHDL-onderdelen door gebruik te maken van een **USE**-bevel.

### 3.5.3. Voorbeeld

Deze werkwijze kunnen we als volgt illustreren: Er wordt een package LOGIC gedefinieerd.

```
package LOGIC is
    type TRISTATE is ('0','1','Z');
    constant ONBEKEND: TRISTATE := '0';
    function COMPLEMENT (A: TRISTATE) return TRISTATE;
end logic;

package body LOGIC is
    function COMPLEMENT...
        ...
    end COMPLEMENT;
end LOGIC;
```

Van dit package wordt gebruik gemaakt om een invertor te beschrijven:

```

use LOGIC.TRISTATE, LOGIC.COMPLEMENT;

entity inverter is
    port( X: in TRISTATE; Y: out TRISTATE);
end inverter;

architecture beschrijving of inverter is
begin
    process
    begin
        Y <= COMPLEMENT(X) after 10 ns;
        wait on X;
    end process;
end beschrijving;

```

Het USE- bevel zorgt ervoor dat in de design entity inverter het type TRISTATE en de functie COMPLEMENT uit het package kan gebruikt worden.

Om gebruik te kunnen maken van alle gedeclareerde objecten en subprogramma's kan men ook schrijven:

```

use LOGIC.all;

```

## 3.6. Libraries

Bij de analyse (compilatie) van een VHDL-eenheid ,worden eerst een syntactische en een semantische controle uitgevoerd. Indien deze compilatie foutloos verloopt, dan wordt de gecompileerde VHDL-eenheid aan een bibliotheek (library) toegevoegd.

In VHDL zijn er twee klassen van library-eenheden: primaire en secundaire.

Primaire eenheden zijn de entity-declaratie, de package-declaratie en de configuratiedeclaratie. Secundaire eenheden zijn de package body en de architecture body.

In een library mag maar één primaire eenheid met een bepaalde naam aanwezig zijn; er mogen echter meerdere secundaire eenheden met dezelfde naam zijn. De volgorde waarin de eenheden gecompileerd worden is belangrijk. Dit heeft te maken met de beschikbaarheid van de declaraties. Zo moet een primaire eenheid steeds vóór de corresponderende secundaire eenheid geanalyseerd worden.

Een library-naam is een logische naam voor een directory in het bestuursstelsel van de computer, waarin de VHDL-eenheden van de library worden bewaard. Deze logische namen kunnen gebruikt worden in een VHDL-broncode.

Een voorbeeld zal dit duidelijk maken:

Stel: de library BASIC\_LIB bevat volgende package LOGIC:

```
package LOGIC is ...
end LOGIC;

....
package body LOGIC is ...
end LOGIC;
```

Dan kunnen we in een andere library ( bijvoorbeeld WORK ) de packages uit library BASIC\_LIB gebruiken zoals in de volgende code wordt getoond:

```
library BASIC_LIB;

use BASIC_LIB.LOGIC;
use LOGIC.all;

....
entity INVERTOR is ...
end INVERTOR;

....
architecture BESCHRIJVING_1 of INVERTOR is ...
end BESCHRIJVING_1;

....
```

Om een library beschikbaar te maken voor een andere VHDL-eenheid, maakt men gebruik van de library-oproep:

```
library LIBRARY_NAAM;
```

Om eenheden uit deze library ter beschikking te krijgen, wordt een USE-zin gebruikt:

```
use LIBRARY_NAAM.EENHEIDSNAAM;
```

Elke VHDL-omgeving kent steeds een library met de naam WORK, waarvoor geen specifieke oproep moet gebeuren: elke VHDL-eenheid heeft rechtstreeks toegang tot de WORK-library. De WORK library is de library waarin de VHDL-eenheden na compilatie worden bewaard. Meestal heeft elke gebruiker een eigen WORK-library. Dit is meestal de huidige working directory.

Naast de library WORK bestaat er in VHDL nog de standaard library STD. De library STD bevat twee packages: de package STANDARD en de package TEXTIO.

## 3.7. Attributen

Verschillende items in VHDL kunnen attributen hebben. Dit geldt met name voor:

- types, subtypes
- procedures, functies
- signalen, constanten, variabelen
- entities, architectures, configurations, packages
- componenten
- statement-labels

Een attribuut is een **kenmerkende eigenschap** van een item. Sommige attributen kunnen een bepaalde waarde aannemen in specifieke omstandigheden. Deze waarde kan aangeduid worden via de attribuutnaam van het item.

De algemene vorm hiervoor is:

<b>ITEMNAAM'ATTRIBUUTNAAM</b>
-------------------------------

In VHDL kan een ontwerper zelf attributen definiëren.

Er bestaan echter al een aantal voorgedefinieerde attributen zoals LEFT, RIGHT, HIGH, LOW, EVENT, ...

Hier volgen enkele voorbeelden van attributen voor arrays:

Gegeven zijn volgende declaraties:

```
type BITPLAATS is range 10 downto 0;
type teller is range -10 to 10;
....
```

Dan gelden de volgende gelijkheden:

```
BITPLAATS'LEFT=15
BITPLAATS'LOW=0
TELLER'RIGHT=10
TELLER'HIGH=10
```

Een voorbeeld van attribuut voor signalen:

of	<pre><b>wait on clk'event;</b>  <b>if ( clk'event and clk='1' ) then ....</b></pre>
----	---

### 3.8. Voorgedefinieerde Operatoren

VHDL stelt de ontwerper een aantal operatoren ter beschikking, die zonder declaratie bruikbaar zijn. De volgende tabel geeft een overzicht:

mod: deling(geheel getal)

rem: restbepaling

\*\* : machtsverheffing

abs: absolute waarde

sll: shift left (logical) functie

srl: shift right (logical) functie

sla: shift left (arithmetic) functie

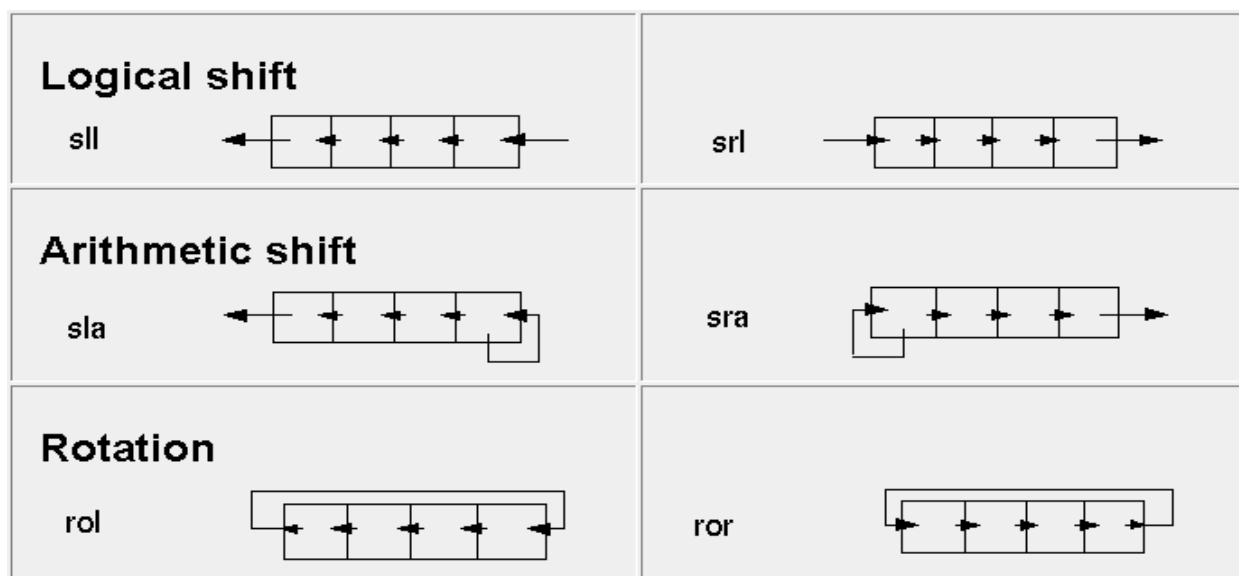
sra: shift right (arithmetic) functie

rol: rotate left (logical) functie

ror: rotate right (logical) functie

<b>logical</b>	<b>not</b>					
	<b>and</b>	<b>or</b>	<b>nand</b>	<b>nor</b>	<b>xor</b>	<b>xnor</b>
<b>relational</b>	<b>=</b>	<b>/=</b>	<b>&lt;</b>	<b>&lt;=</b>	<b>&gt;=</b>	<b>&gt;</b>
<b>shift</b>	<b>sll</b>	<b>srl</b>	<b>sla</b>	<b>sra</b>	<b>rol</b>	<b>ror</b>
<b>arithmetic</b>	<b>+</b>	<b>-</b>				
	<b>*</b>	<b>/</b>	<b>mod</b>	<b>rem</b>		
	<b>**</b>	<b>abs</b>				





Voor de rekenkundige operatoren met twee operands geldt dat beide operands van hetzelfde type moeten zijn. Een uitzondering op deze regel is het delen en vermenigvuldigen van een fysische grootheid door of met een geheel of reëel getal.

Een uitdrukking waarin een relationele grootheid wordt gebruikt geeft als resultaat een waarde van het BOOLEAN type. Ook hier moeten de operands van hetzelfde type zijn.

De logische operatoren zijn gedefinieerd voor de types BIT, BIT\_VECTOR en BOOLEAN en eveneens voor eendimensionale arrays van elementen van deze types. Het resultaat is dan van hetzelfde type als de operands.

Uitdrukkingen (expressions) worden samengesteld uit namen (operands), constanten en operatoren, zoals in hogere programmeertalen. Het gewone gebruik van ronde haakjes is toegelaten. De volgorde van de bewerkingen volgt de gewone regels.

Voorbeeld:

```
Variable A,B,C : INTEGER;
C:= (A/B)*B+(A rem B)      --dan is C=A
```

## 3.9. Sequentiële Statements

Alle statements in processen of subprogramma's worden sequentieel (de een na de ander ) uitgevoerd.

### 3.9.1. Toekenningsbevel voor variabelen

Variabelen kunnen enkel in één process gedefinieerd en gebruikt worden. Variabelen kunnen dus nooit informatie doorgeven aan andere processen.

De algemene vorm is:

**Variabelenaam** := <expressie>;

Een nieuwe waarde voor een variabele wordt direct toegekend zonder vertraging!

### 3.9.2. Toekenningsbevel voor signalen

Signalen worden gebruikt om informatie tussen processen door te geven. Een signaal heeft een actuele waarde en een lijst van toekomstige waarden, die bewaard worden in een zgn. driver ( zie par. 4.1.2.)

Een signaal kan meerdere bronnen hebben. In dat geval wordt het signaal ‘resolved’ genoemd. Een uiteindelijke waarde wordt dan berekend door een resolutiefunctie( zie par. 3.3.2. )

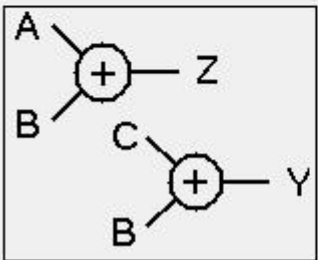
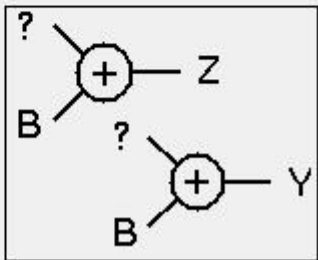
De algemene vorm is:

**signaalnaam** <= <expressie>;

### 3.9.3. Variabelen vs. signalen

- Variabelen krijgen direct hun waarden toegekend tijdens het verloop van het process. Signalen echter krijgen hun waarden pas op het einde van het process.
- Alleen de laatste signaaltoekenning wordt doorgegeven:  
     Voorbeeld 2: M <= A; wordt overschreven door M <= C;  
                   De 2<sup>de</sup> input van de adder wordt verbonden met C.

Voorbeeld:

<pre> signal A, B, C, Y, Z : integer; begin   process (A, B, C)  <b>variable M, N : integer;</b>     begin       M := A;       N := B;       Z &lt;= M + N;       M := C;       Y &lt;= M + N;     end process; </pre>	<pre> signal A, B, C, Y, Z : integer; <b>signal M, N : integer;</b> begin   process (A, B, C, M, N)      begin       M &lt;= A;       N &lt;= B;       Z &lt;= M + N;       M &lt;= C;       Y &lt;= M + N;     end process; </pre>
	

### 3.9.4. IF statement

De bedoeling van dit bevel is duidelijk. Meerdere vormen zijn mogelijk:

```

if CONDITION then
  -- sequential statements
end if;

if CONDITION then
  -- sequential statements
else
  -- sequential statements
end if;

if CONDITION then
  -- sequential statements
elsif CONDITION then
  -- sequential statements
  ...
else
  -- sequential statements
end if;

```

```

entity IF_STATEMENT is
port( A, B, C, X : in bit_vector( 3 downto 0 );
      Z          : out bit_vector( 3 downto 0 )
);
end IF_STATEMENT;

architecture EXAMPLE of IF_STATEMENT is
begin

process( A, B, C, X )
begin
    if ( X < "1000") then Z<=B;
    elsif ( X="1111") then Z<=C;
    else Z<= A;
    end if;
end process;

end EXAMPLE;

```

### 3.9.5. CASE Statement

Het CASE statement laat toe om één bevel uit een lijst van bevelen te selecteren met behulp van een selectie-expressie.

```

case EXPRESSIE is

    when waarde_1 =>                -- sequentiële statements

    when waarde_2 | waarde_3 =>    -- sequentiële statements

    when waarde_4 to waarde_N =>  -- sequentiële statements

    when others =>                -- sequentiële statements

end case;

```

Zoals blijkt uit het voorbeeld moeten alle mogelijke waarden van de expressie voorzien zijn. Voor al de waarden die niet gedefinieerd zijn en overblijven, gebruiken we het keyword ‘**others**’.

Het type van de expressie moet overeen komen met het type van de waarden.

Waarden kunnen gegroepeerd worden met het symbool: ‘|’ (= or) of dmv. een range.

Voorbeeld:

```
entity CASE_STATEMENT is
  port (A, B, C, X: in integer range 0 to 15;
        Z: out integer range 0 to 15;
  end CASE_STATEMENT;

  architecture EXAMPLE of CASE_STATEMENT is
  begin
    process (A, B, C, X)
    begin
      case X is
        when 0 =>
          Z <= A;
        when 7 | 9 =>
          Z <= B;
        when 1 to 5 =>
          Z <= C;
        when others =>
          Z <= 0;
        end case;
      end process;
    end EXAMPLE;
```

### 3.9.6. For-lus

De algemene vorm is:

```
LUS_LABEL:
  for LUSVARIABLE in BEREIK loop
    ....
    ....      --statements
    ....
  end loop LUS_LABEL;
```

Dit bevel laat toe te controleren hoeveel maal de lus doorlopen wordt.

De lusvariabele hoeft niet gedeclareerd te worden. Zijn waarde is read only dwz. Het aantal keren dat de lus doorlopen wordt staat vast op het moment dat de for-lus uitgevoerd wordt en kan tijdens het uitvoeren van de lus niet gewijzigd worden.

De lus label is optioneel.

Voorbeeld:

```

entity faculteit is
port( n: in integer;
      fac: out integer);
end faculteit;

architecture for_lus of faculteit is
begin
    process( n )
    variable prd: integer;
    begin
        prd:= 1;
        for i in 1 to n loop
            prd:= prd * i;
        end loop;
        fac <= prd;
    end process;
end for_lus;

```

### 3.9.7. While lus.

Dit bevel wordt gebruikt wanneer het beëindigen van een lus geboden is onder een bepaalde voorwaarde. De lus wordt verlaten indien het resultaat van een Booleaanse expressie fout is.

De algemene vorm is:

```

LUS_LABEL:
    while <LUSCONDITIE> loop
        ....
        ....    --statements
        ....
    end loop LUS_LABEL;

```

Voorbeeld:

```

entity faculteit is
port( n: in integer;
      fac: out integer);
end faculteit;

architecture while_lus of faculteit is
begin
    process( n )
        variable prd, f: integer;
        begin
            prd:=n;
            f:=1;
            while prd > 0 loop
                f:=f*prd;
                prd:=prd-1;
            end loop;
            fac <= f;
        end process;
    end while_lus;

```

### 3.9.8. Lus met exit conditie

Als algemene vorm stellen we voorop:

```

LUS_LABEL: loop
    ....
    ....    --statements
    exit LUS_LABEL;
  of
    exit LUS_LABEL when <conditie>;
    ....
end loop LUS_LABEL;

```

Een voorbeeld kan dit verduidelijken:

```

entity faculteit is
port( n: in integer;
      fac: out integer);
end faculteit;

architecture exit_lus of faculteit is
begin
  process( n )
  variable prd, f: integer;
  begin
    prd:= n;
    f:= 1;
    loop
      exit when prd <=0;
      f:= f*prd;
      prd:= prd -1;
    end loop;
    fac <= f;
  end process;
end exit_lus;

```

### 3.9.9. WAIT statement

Bij het uitvoeren van een WAIT-bevel in een proces, wordt de uitvoering ervan onderbroken en de voorwaarden voor reactivering ingesteld.

Er bestaan vier soorten voorwaarden:

- het proces wordt onderbroken gedurende de aangegeven tijd:
- er moeten events optreden aan bepaalde signalen, na een event op een van de signalen van de lijst gaat het proces verder:
- het proces wordt gereactiveerd wanneer aan een bepaalde logische voorwaarde voldaan is:
- de uitvoering van het proces wordt opgeschort gedurende de rest van de simulatietijd:

**wait for** specific\_time;

**wait on** signal\_list;

**wait until** condition;

**wait;**

De voorwaarden mogen ook gecombineerd worden, zoals in:

**wait on** signal\_list **until** condition;



Een eenvoudig voorbeeld situeert het WAIT-bevel in een proces:

```

OF_FUNCTIE:
  process
  begin
    Y <= A or B;
    wait on A, B;
  end process;

```

Indien een proces steeds voor dezelfde signalen gevoelig is, dan kan men een zgn. **sensitivity-lijst** invoeren. Het gebruik van het WAIT-bevel is nu verboden!

```

OF_FUNCTIE:
  process( A, B )           --dit is de sensitivity-lijst
  begin
    Y <= A or B;
  end process;

```

### 3.9.10. ASSERT statement.

Dit bevel geeft aan de ontwerper de mogelijkheid om bepaalde testen uit te voeren, of bepaalde voorwaarden te toetsen in de modelbeschrijving van een VHDL-eenheid. De voorwaarde is Booleaans van vorm. Zij wordt tijdens de simulatie geëvalueerd. Indien niet aan de voorwaarde voldaan is, wordt er een bericht naar het scherm gestuurd.

```

assert <VOORWAARDE>
report "BERICHT"
severity <SEVERITY_LEVEL>;

```

Een eenvoudig voorbeeld situeert het assert-statement in een proces:

```

assert a>b
report "b is groter of gelijk aan a"
severity warning;

```

Het type 'severity level' wordt gedefinieerd in het package STANDARD en kan de volgende waarden aannemen: NOTE, WARNING, ERROR en FAILURE.

Afhankelijk van deze waarde vervolgt de simulatie zich, of wordt het simuleren afgebroken.

### 3.9.11. Procedure oproep.

Met dit bevel worden procedures geactiveerd gedurende de uitvoering van een proces. Een procedure met de commando's die zij bevat, kan door verschillende processen worden opgeroepen.

De algemene vorm is:

```
procedurenaam( ASSOCIATIELIJST );
```

### 3.9.12. RETURN statement.

Het return statement wordt gebruikt om de uitvoering van een subprogramma te beëindigen. De controle wordt teruggegeven aan het oproepende proces. In het geval van een functie kan een waarde als resultaat worden meegegeven.

Een voorbeeld:

```
function ANDFUNCTIE( X, Y: in BIT) return BIT is
begin
    if X='1' and Y='1' then
        return '1';
    else
        return '0';
    end if;
end ANDFUNCTIE;
```

## 3.10. Concurrent Statements

Concurrent statements worden op hetzelfde tijdstip uitgevoerd, onafhankelijk van de volgorde waarin ze voorkomen.

Concurrent bevelen in VHDL zijn verkorte notaties voor een bepaald proces. Ze laten toe om overzichtelijk en compacte VHDL-codes te schrijven. Elk concurrent bevel mag door een label voorafgegaan worden.

### 3.10.1. Eenvoudig concurrent toekenningsbevel

Algemene vorm:

```
(labelnaam:) signaalnaam <= (optie) <waveform>;
```

Als optie kan TRANSPORT of GUARDED gebruikt worden. (zie par. 4.1.2.)

Voorbeeld:

```
architecture rtl of fulladder is
    signal A,B: bit;
begin
    A <= X xor Y;
    B <= A and Cin;
    Som <= A xor Cin;
    Cout <= B or ( X and Y);
end rtl;
```

In de architectuurbeschrijving komen verschillende concurrent statements voor. Zij beschrijven de functie van de fulladder.

Een concurrent statement is “**event-driven**” : het wordt alleen maar uitgevoerd na een waardeverandering van een signaal uit de waveform. De volgorde van de concurrent statements in een architectuurbeschrijving heeft geen belang.

### 3.10.2. Voorwaardelijk toekenningsbevel.

Algemene vorm:

```
(labelnaam:) signaalnaam <= (optie)
    VALUE_1 when CONDITION_1 else...
    VALUE_2 when CONDITION_2 else...
    ....
    VALUE_N when CONDITION_N else ...
VALUE;
```

De Booleaanse condities geven aan welke waveform aan het signaal wordt toegekend.  
De vermelding (optie) staat eventueel voor het sleutelwoord GUARDED of TRANSPORT.  
Een voorwaardelijk toekenningsbevel staat equivalent met een “if..., elsif...,else constructie”.

In het voorbeeld zien we twee equivalente beschrijvingen van een multiplexer:

```

entity CONDITIONAL_ASSIGNMENT is
  port (A, B, C, X: in bit_vector (3 downto 0);
        Z_CONC : out bit_vector (3 downto 0);
        Z_SEQ  : out bit_vector (3 downto 0));
end CONDITIONAL_ASSIGNMENT;

architecture EXAMPLE of CONDITIONAL_ASSIGNMENT is
begin
  -- Concurrent version of conditional signal assignment
  Z_CONC <= B when X = "1111" else
           C when X > "1000" else
           A;

  -- Equivalent sequential statements
  process (A, B, C, X)
  begin
    if (X = "1111") then
      Z_SEQ <= B
    elsif (X > "1000") then
      Z_SEQ <= C;
    else
      Z_SEQ <= A;
    end if;
  end process;
end EXAMPLE;

```

### 3.10.3. Selectief toekenningsbevel.

Het gedrag van een selectief toekenningsbevel is equivalent aan een case statement.

Algemene vorm:

```

(labelnaam:) with EXPRESSION select
  signaalnaam <= (optie)
    VALUE_1 when CHOICE_1,
    VALUE_2 when CHOICE_2 | CHOICE_3,
    VALUE_3 when CHOICE_4 to CHOICE_5,
    ....
    VALUE_N when others;

```

Het volgende voorbeeld zal dit verduidelijken:

```

entity SELECTED_ASSIGNMENT is
  port (A, B, C, X: in integer range 0 to 15;
        Z_CONC : out integer range 0 to 15;
        Z_SEQ  : out integer range 0 to 15);
end SELECTED_ASSIGNMENT;

architecture EXAMPLE of SELECTED_ASSIGNMENT is
begin
  -- Concurrent version of selected signal assignment
  with X select
    Z_CONC <= A when 0,
              B when 7 | 9,
              C when 1 to 5,
              0 when others;

  -- Equivalent sequential statements
  process (A, B, C, X)
  begin
    case X is
      when 0      => Z_SEQ <= A;
      when 7 | 9  => Z_SEQ <= B;
      when 1 to 5 => Z_SEQ <= C;
      when others => Z_SEQ <= 0;
    end process;
  end EXAMPLE;

```

**Opmerking:**

- Deze voorbeelden van concurrent statements beschreven allen de functionaliteit van een multiplexer. Het is onmogelijk om alleen met concurrent statements geheugenelementen zoals flipflops te beschrijven!
- Deze concurrent statements worden gebruikt als “shortcuts” van sequentiële statements.

### 3.10.4. Het concurrent ASSERTION-bevel.

De syntaxis hiervoor is dezelfde als die van het sequentiële assertion-bevel binnen in een proces. We geven hieronder een voorbeeld van een concurrent assertion-bevel, gevolgd door een proces dat zijn sequentieel equivalent bevat.

```
--concurrent version of assertion
```

**SRFF\_CHECK:**

```

  assert not (S='1' and R='1')
    report “S en R beide gelijk aan 1”
    severity ERROR;

```

```

--sequentieel equivalent

SRFF_CHECK:
process( S, R )
begin
    assert not (S='1' and R='1')
        report "S en R beide gelijk aan 1"
        severity ERROR;
end process;

```

Het equivalent proces is een passief proces: het veroorzaakt geen waardeveranderingen van signalen. Het bevel mag daarom in een entity-declaratie worden geplaatst.

### 3.10.5. De concurrent PROCEDURE-oproep.

In VHDL kan een procedure ook op een concurrent manier worden opgeroepen. Een dergelijke concurrent procedure-oproep is equivalent met een proces zonder sensitivity-lijst en met een uitvoerbaar gedeelte dat bestaat uit een sequentiële procedure-oproep. We illustreren dit met een voorbeeld:

```

--concurrent procedure-oproep

PROC_1( IN_SIGNAAL, UIT_SIGNAAL, IN_VARIABELE);

--sequentieel equivalent

process
begin
    PROC_1( IN_SIGNAAL, UIT_SIGNAAL, IN_VARIABELE);
    wait on IN_SIGNAAL;
end process;

```

Een concurrent procedure-oproep wordt uitgevoerd telkens als een van deingangssignalen van waarde verandert.

## 3.11. RTL-Style

### 3.11.1. Het PROCESS-commando

In VHDL moet een ontwerper het gedrag van een discreet systeem op twee niveaus beschrijven: een sequentieel en een concurrent niveau. Het coderen van het inwendige gedrag in een proces, vindt plaats op het sequentiële niveau. Op het concurrent niveau worden de relaties en de communicatie tussen de processen vastgelegd. Het process-bevel is een concurrent bevel dat bepaalt wat er moet gebeuren als een proces geactiveerd wordt.

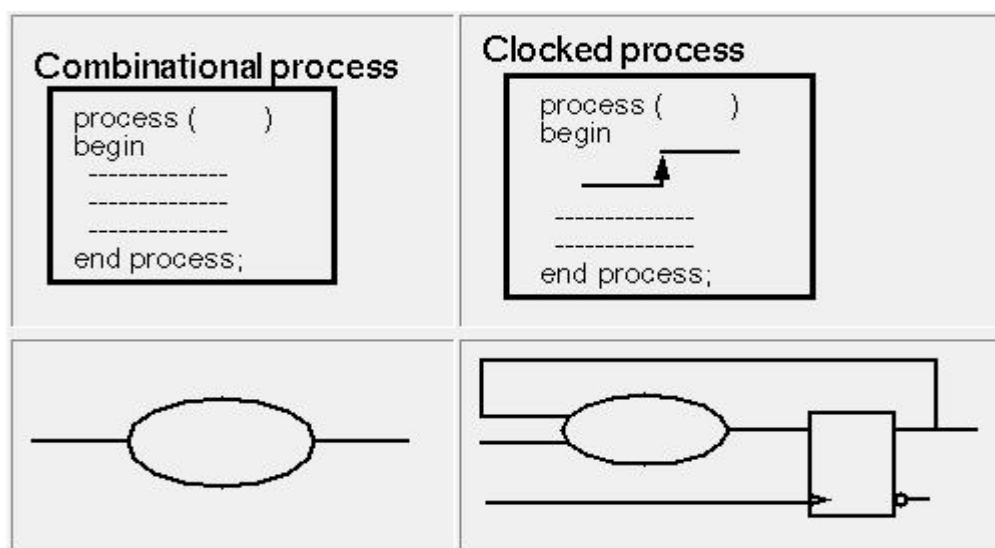
De algemene vorm van een proces is:

<b>PROCESLABEL:</b>	
<b>Process</b>	
.....	-- declaraties
<b>begin</b>	
.....	-- sequentiële bevelen
<b>end process;</b>	

Het proceslabel dient om een naam te geven aan het proces. Lokale variabelen e.d. worden gedefinieerd in het declaratiegedeelte. De bevelen uit het proces vormen een sequentieel uit te voeren programma, dat het gedrag van het proces bepaalt.

Het activeren gebeurt door waardeveranderingen of events op signalen waarvoor het proces gevoelig is. De door het proces berekende resultaten worden toegekend aan de uitgangssignalen, die dan andere processen kunnen activeren. Telkens een proces nieuwe informatie ontvangt, wordt het opnieuw geactiveerd. De uitvoering van een proces kan echter tijdelijk opgeschort worden door middel van een wait-bevel (zie verder). Een proces dat geen waarden aan signalen toekent, noemt men een passief proces, omdat het geen andere processen kan activeren. Een dergelijk passief proces mag in de entity-declaratie worden geplaatst.

We gaan twee soorten processen bekijken namelijk een combinatorisch en een synchroon proces.



### 3.11.2. Het combinatorisch proces

#### 3.11.2.1. Sensitivity-lijst.

De sensitivity lijst van een combinatorisch proces bevat alle signalen die tijdens dat proces gelezen worden. Tijdens de simulatie wordt het proces enkel gestart indien er een event gebeurt op een van de signalen uit de sensitivity lijst. Alle sequentiële statements worden uitgevoerd en na het laatste sequentiële statement wordt er terug gewacht op een nieuw event.

Voorbeeld van een multiplexer:

```
process( A, B, SEL )
begin
    if ( SEL ='1' ) then OUT <= A;
                        else OUT <= B;
    end if;
end process;
```

#### 3.11.2.2. WAIT Statement.

Hetzelfde resultaat wordt bekomen indien we i.p.v. de sensitivity lijst een wait statement gebruiken. Het wait statement moet als laatste statement geplaatst worden in het proces en bevat dezelfde signalen als in de sensitivity lijst.

Alle sequentiële statements worden uitgevoerd tot we het wait statement bereiken. Nu wordt er gewacht tot er aan het wait statement voldaan is.

Indien er geen sensitivity lijst aanwezig is moet een proces steeds via een wait statement onderbroken worden zoniet komt de simulator terecht in een oneindige lus.

```
process
begin
    if ( SEL ='1' ) then OUT <= A;
                        else OUT <= B;
    end if;
    wait on A, B, SEL;
end process;
```



### 3.11.3. Het synchrone proces

Het basiselement van de synchrone logica is de “positieve flank getriggerde D-flipflop”. De data wordt opgenomen in het geheugenelement op de stijgende flank van de klok. De data blijft in het geheugenelement aanwezig tot de volgende stijgende flank van de klok.

De VHDL-beschrijving van een positieve flank getriggerde D-flipflop ziet er als volgt uit:

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

entity dff_logic is
port( d, clk: in std_logic;
      q: out std_logic);
end dff_logic;

architecture example of dff_logic is
begin
    process( clk )
    begin
        if ( clk'event and clk='1') then q <= d;
        end if;
    end process;
end example;
```

Hieruit blijkt hoe een stijgende klokflank kan gedetecteerd worden.

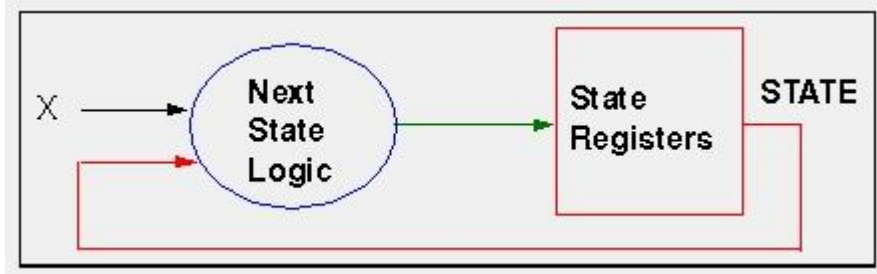
We raden aan om elk ontwerp te voorzien met een reset mogelijkheid. De reset in volgend voorbeeld is asynchroon en actief laag. Deze reset mag niet in de sensitivity lijst ontbreken!

De sensitivity lijst bevat dus enkel het kloksignaal en alle andere asynchrone signalen (reset).

```
process( clk, reset)
begin
    if reset ='0' then ...
        --asynchronous register reset
    elsif ( clk'event and clk='1') then ....
        --combinatorics
    end if;
end process;
```

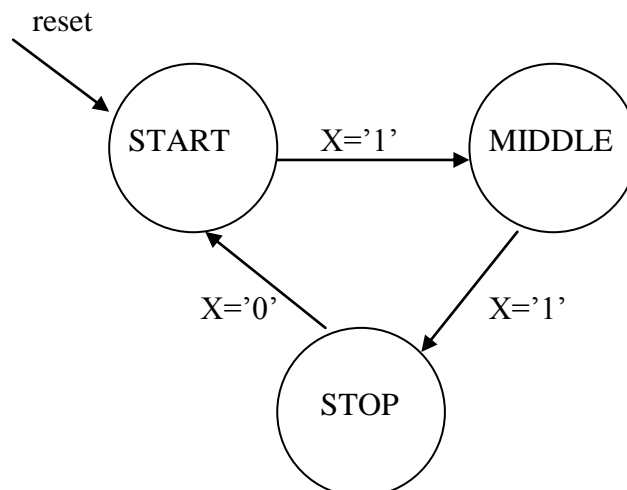
### 3.11.4. Finite State Machines and VHDL

#### 3.11.4.1. Algemene structuur.



#### Voorbeeld FSM:

Toestandsdiagram:



In het toestandsdiagram geven de cirkels de verschillende mogelijke toestanden weer. Indien de voorwaarde die bij de pijl staat WAAR is op het moment van een stijgende klokflank, dan wordt er overgegaan naar de volgende toestand. Dit is een synchroon gebeuren. Indien de asynchrone reset actief wordt dan gaat er direct, zonder te wachten op een stijgende klokflank, overgegaan worden naar de toestand: start.

VHDL code:

```

library ieee;
use ieee.std_logic_1164.all;

entity fsm_1 is
port(
X: in std_logic;
CLK: in std_logic;
CLR: in std_logic
);
end fsm_1;

architecture gedrag of fsm_1 is
type state_type is ( start, middle, stop);
signal state: state_type;

begin
FSM: process( CLK, CLR )
begin
if CLR = '0' then state <= start;
elsif ( CLK'event and CLK='1') then

    case state is
        when start => if X='1' then state <=middle;
                        end if;
        when middle => if X= '1' then state <=stop;
                        end if;
        when stop => if X= '0' then state <=start;
                        end if;
        when others => state <= start;
    end case;
end if;
end process FSM;

end gedrag;

```

### 3.11.4.2. State encoding.

Een FSM is een abstracte beschrijving van een digital systeem. Voor synthese is het noodzakelijk dat we de toestanden een binaire voorstelling kunnen geven. Deze omzetting noemen we “**state encoding**”.

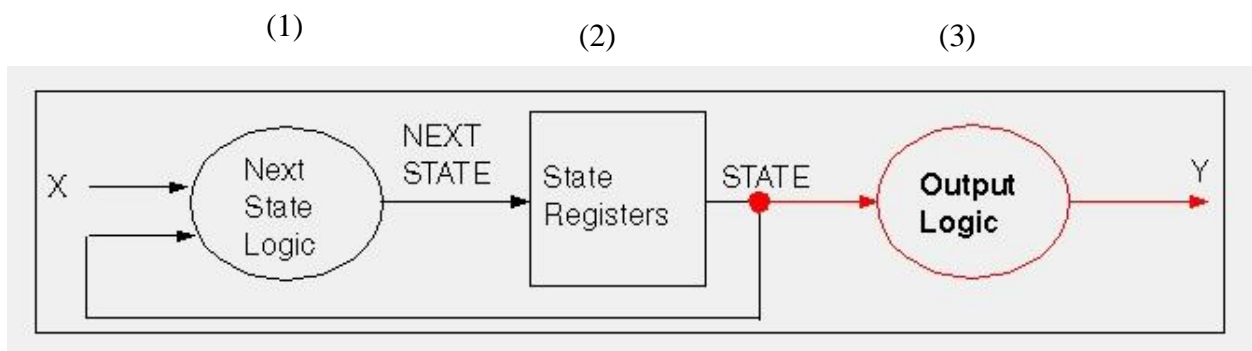
<pre>type STATE_TYPE is ( START, MIDDLE, STOP ); signal STATE : STATE_TYPE;</pre>	<b>. State encoding responsible for safety of FSM</b>
<pre>START  -&gt; "00" MIDDLE -&gt; "01" STOP   -&gt; "10"</pre>	<b>. Default encoding: binary</b>
<pre>START  -&gt; "001" MIDDLE -&gt; "010" STOP   -&gt; "100"</pre>	<b>. Speed optimized default encoding: one hot</b>

### 3.11.4.3. Machine van Moore.

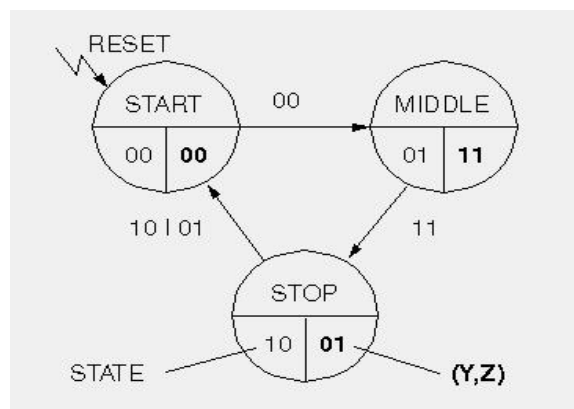
De uitgang is een functie van de huidige toestand:  $Y = f(STATE)$ .

We splitsen het FSM op in :

1. een combinatorisch deel waarin de ingangen mee de volgende toestand bepalen.
2. een sequentieel deel
3. output deel: bij Moore staat geen enkele input in dit combinatorisch deel



Voorbeeld:



Voorbeeld met 3 processen:

```

library ieee;
use ieee.std_logic_1164.all;

entity fsm_moore is
port(
  A,B : in std_logic;
  clk,clr: in std_logic;
  Y, Z: out std_logic
);
end fsm_moore_1;

architecture gedrag1 of fsm_moore is

  type state_type is (start, middle, stop );
  signal state, next_state: state_type;

begin
  comb: process( A,B, state )
  begin
    case state is
      when start => if (A or B)= '0' then next_state<= middle;
                      end if;
      when middle => if (A and B)='1' then next_state <= stop;
                      end if;
      when stop => if (A xor B)='1' then next_state <= start;
                      end if;
      when others => next_state <= start;
    end case;
  end process comb;

  seq: process( clk, clr )
  begin
    if clr='0' then state <= start;
    elsif (clk'event and clk='1') then state <= next_state;
    end if;
  end process seq;

  outp:process( state )
  begin
    case state is
      when start => Y <= '0'; Z<='0';
      when middle => Y <= '1';Z<='1';
      when stop => Y <= '0';Z<='1';
      when others => Y <='0'; Z<='0';
    end case;
  end process outp;
end gedrag1;

```

Voorbeeld met 2 processen:

```

library ieee;
use ieee.std_logic_1164.all;

entity fsm_moore is
port(
A,B : in std_logic;
clk,clr: in std_logic;
Y, Z: out std_logic
);
end fsm_moore;

architecture gedrag2 of fsm_moore is

type state_type is (start, middle, stop );
signal state: state_type;

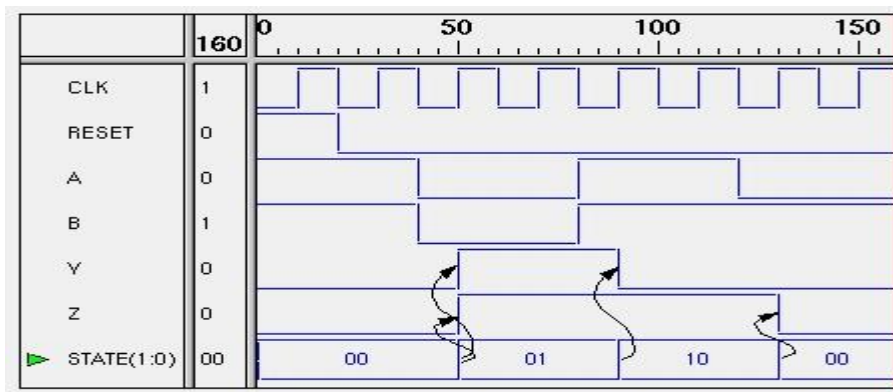
begin
    seq: process( clk, clr )
    begin
        if clr='0' then state <= start;
        elsif (clk'event and clk='1') then
            case state is
                when start => if X= '1' then state<= middle;
                                end if;
                when middle => if X='1' then state <= stop;
                                end if;
                when stop => if X='0' then state <= start;
                                end if;
                when others => state <= start;
            end case;
        end if;
    end process seq;

    outp:process( state )
    begin
        case state is
            when start => Y <= '0'; Z<='0';
            when middle => Y <= '1';Z<='1';
            when stop => Y <= '0';Z<='1';
            when others => Y <='0'; Z<='0';
        end case;
    end process outp;

end gedrag2;

```

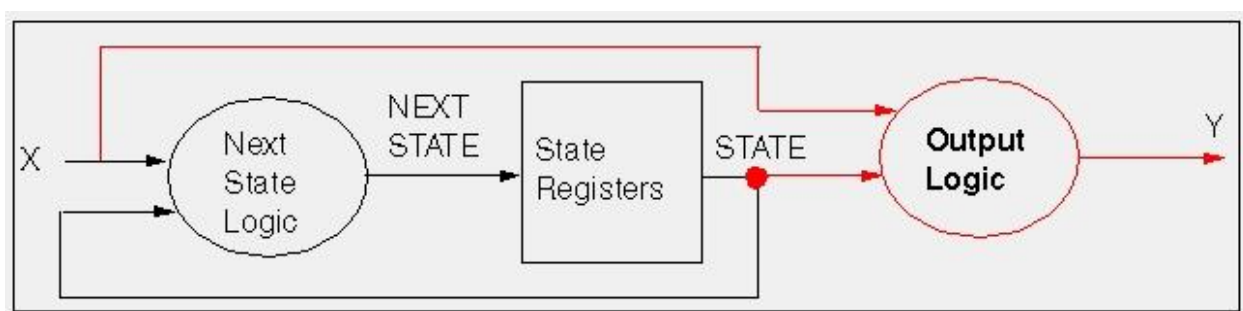
Simulatieresultaat:



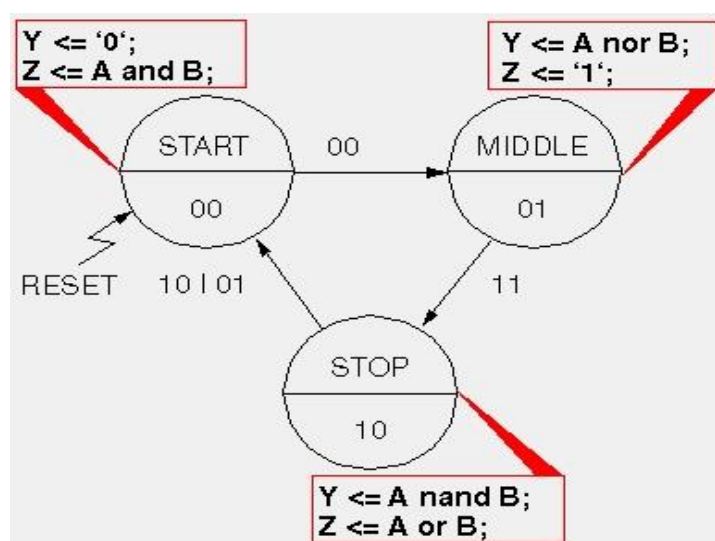
De uitgang Y en Z veranderen simultaan met de state! En niet met de ingang!

### 3.11.4.4. Mealy.

De uitgang is een functie van de ingang(en) en de huidige toestand:  $Y = f(X, STATE)$ .



Voorbeeld:



Voorbeeld met drie processen:

architecture gedrag of fsm\_mealy is

```
type state_type is (start, middle, stop );
signal state, next_state: state_type;
```

```
begin
```

```
  comb: process( A, B, state )
```

```
  begin
```

```
    --zoals bij het voorbeeld van Moore!
```

```
    --
```

```
  end process comb;
```

```
  seq: process( clk, clr )
```

```
  begin
```

```
    --zoals bij het voorbeeld van Moore!
```

```
    --
```

```
  end process seq;
```

```
  outp:process( state,A,B )
```

```
  begin
```

```
    case state is
```

```
      when start => Y <= '0'; Z<=A and B;
```

```
      when middle => Y <= '1';Z<=A nor B;
```

```
      when stop => Y <= '0';Z<=A nand B;
```

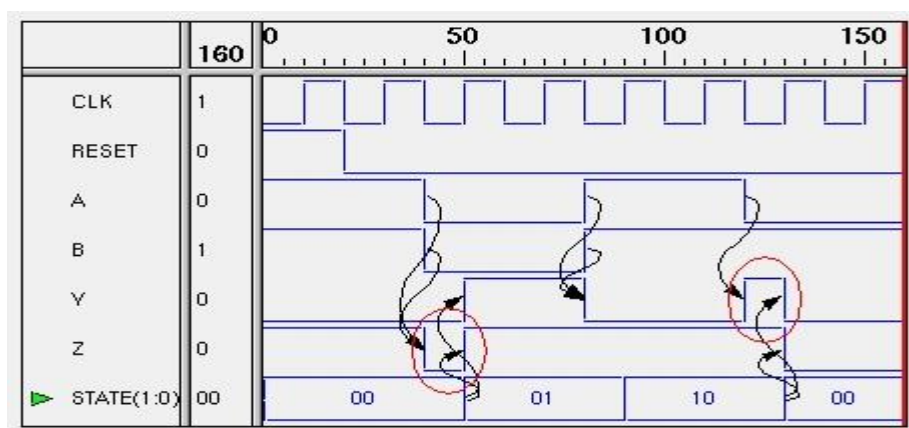
```
      when others => Y <='0'; Z<='0';
```

```
    end case;
```

```
  end process outp;
```

```
end gedrag;
```

Simulatieresultaat:



De uitgang Y en Z veranderen met de ingang!



## 3.12. Geavanceerde technieken

### 3.12.1. Het gebruik van GENERICS.

Naast de al besproken port-lijsten kan men in entity-declaraties ook GENERICS invoeren. Via generics kan een oudercomponent een bepaalde waarde doorgeven aan een kindcomponent. Interessante toepassingen hiervoor zijn: vertragingparameters, subtypebereik, aantal instances van een componenttype, de grootte van de dimensies van een array, enz.

In het volgende voorbeeld wordt een counter ontworpen die telt tot een waarde “max\_value”. Deze waarde wordt gedefinieerd via een generic in de entity declaratie en wordt behandeld als een constante in de architectuur.

Aan een generic kan men een default waarde toekennen (dit gebeurt in het generic statement), die gebruikt wordt indien de generic geen specifieke toekenning krijgt.

```
entity counter is
  generic ( max_value : integer:=15 );           --default value
  port( ....
        count: out integer range 0 to max_value
        );
end counter;
```

Een generic krijgt zijn specifieke waarde toegekend via een “**generic map**”, tijdens het component-instantiation-bevel. Dit wil zeggen dat de default waarde dan overschreven wordt.

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

architecture rtl of totaal_counters is

  component counter
    generic( max_value:= integer:=15);
    port(...);
  end component;

  begin
    counter1: counter port map (...);           --max_value with default waarde
    counter2: counter generic map ( max_value => 31)
      port map (...);
    .....
  end rtl;
```

### 3.12.2. Het GENERATE statement.

In veel ontwerpen vinden we regelmatige patronen van componenten van hetzelfde componenttype terug. Deze structuren kunnen we beschrijven met een GENERATE-statement.

De algemene vorm is:

**for** INDEX **in** <DISCREET BEREIK>

Voorbeeld:

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

architecture rtl of generate_counter is

    component counter
        generic( max_value: integer := 15);
        port(...);
    end component;

    begin
        GEN: for I in 2 to 5 generate
            counterX: counter generic map( max_value => 2**I-1)
                                port map(...);
        end generate;

        ....
    end rtl;
```

Het “ for generate” commando heeft steeds een label nodig.

De parameter I is enkel gedefinieerd binnen het generate statement en moet daarom niet worden gedeclareerd.

Om het generate statement nog krachtiger te maken kan een “**if-generate statement**” gebruikt worden. Het laat toe onregelmatigheden aan de rand van een overigens regelmatige structuur te beschrijven. In een if-generate statement mogen geen “else-” of “**elsif-constructies**” voorkomen.

In het volgende voorbeeld worden 8 counters aangemaakt.

Voor I gaande van 1 tot 5 wordt de maximale counter waarde berekend met de formule:  $2^{**}I-1$  (dit is: 1,3,7,15,31).

Voor hogere indexwaarden blijft de maximale counter waarde 31.

Voorbeeld:

```

entity generate_counter is
    port(...);
end entity;

architecture rtl of generate_counter is

    component counter
        generic ( max_value: integer:= 15);
        port(...);
    end component;

begin
    GEN: for I in 1 to 8 generate

        COND1: if I <=5 generate

            counterX: counter generic map (max_value => 2**I-1)
                                port map (...);

        end generate;

        COND2: if I >5 and vlag generate

            MAX: counter generic map (max_value => 31)
                                port map(...);

        end generate;

    end generate;

    ....
end rtl;

```

## 4. Simulatie.

### 4.1. Tijdsmodel in VHDL.

VHDL heeft een Event-driven simulator. In tegenstelling met gewone code speelt de tijd een belangrijke rol. Signalen veranderen steeds na een bepaalde vertraging.

#### 4.1.1. Waveforms en drivers.

Sequentiële en ook parallelle (concurrent) waardetoekenningen hebben de volgende vorm:

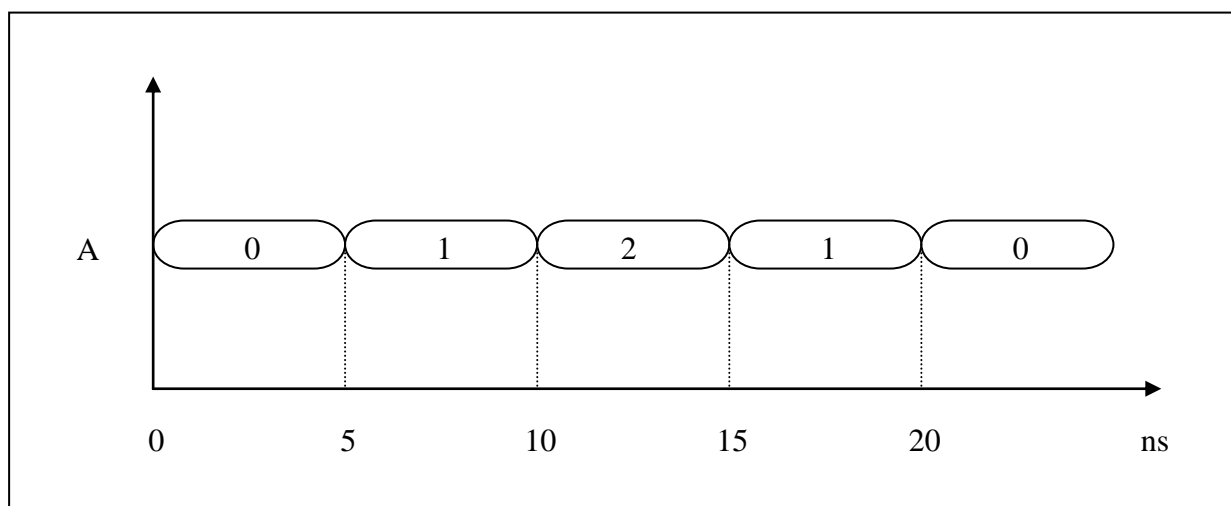
**[signaal] <= [waveform-element]**

Vb: `b <= not( a ) after 3ns;`

Een wave-form element bestaat uit een waarde en de vertraging die de waardetoekenning kenmerkt.

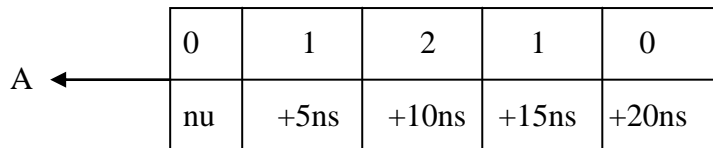
vb: `A <= transport 1 after 5 ns, 2 after 10 ns, 1 after 15 ns, 0 after 20 ns;`

We nemen aan dat A een intern signaal is van het integer-type. Er kunnen dus geen waardetoekenningen aan het signaal gebeuren vanuit andere processen of concurrent commando's. Indien A op het huidige simulatiemoment de waarde 0 heeft, dan kunnen we het tijdsverloop schetsen. Vertragingen van waardetoekenningen aan signalen worden steeds relatief t.o.v. het huidige simulatietijdstip uitgedrukt.



Tijdsverloop van het signaal A.

De waveform-elementen (events) bestaan dus uit koppels van een waarde en een tijdstip. Bij het uitvoeren van het waardetoekeningscommando worden de gegevens bewaard in een datastructuur die de **driver** (eventlist) van het signaal wordt genoemd. De waveform-elementen worden achteraan de driver bijgevoegd, zoals voorgesteld in volgende figuur. Het signaal krijgt dan de gepaste waarden naarmate de simulatietijd vordert.



0	1	2	1	0
nu	+5ns	+10ns	+15ns	+20ns

De driver van een signaal A

Als een signaal gekoppeld is aan een port, dan kan ook deze port een bron zijn van waarden voor dit signaal. Een voorbeeld hiervan is te vinden in Code 5 (par.2.3.3.).

De effectieve waarde van een signaal dat meerdere bronnen heeft, wordt bepaald door de resolutiefunctie.

De simulator maakt een driver voor elk element uit het linkerlid van een waardetoekeningscommando. Zo ook in het volgende voorbeeld:

`A_REG(0 to 3) <="0101" after 10 ns;`

Voor de vier elementen van het register zal een driver worden gecreëerd. Bij het uitvoeren van het commando wordt de eerste bit (0) toegevoegd aan de driver van het eerste signaal `A_REG(0)`, enz...

In een proces wordt er voor elk signaal dat in het linkerlid van een sequentieel waardetoekeningscommando staat slechts één driver aangemaakt, zelfs al wordt dit signaal meerdere malen vernoemd. De driver krijgt een beginwaarde die gelijk is aan de standaardwaarde (default) die hoort bij de declaratie van de port van het signaal.

## 4.1.2. Delay modellen.

Er zijn twee delay modellen in VHDL:

- Transport delay
- Inertial delay (standaardmodel)

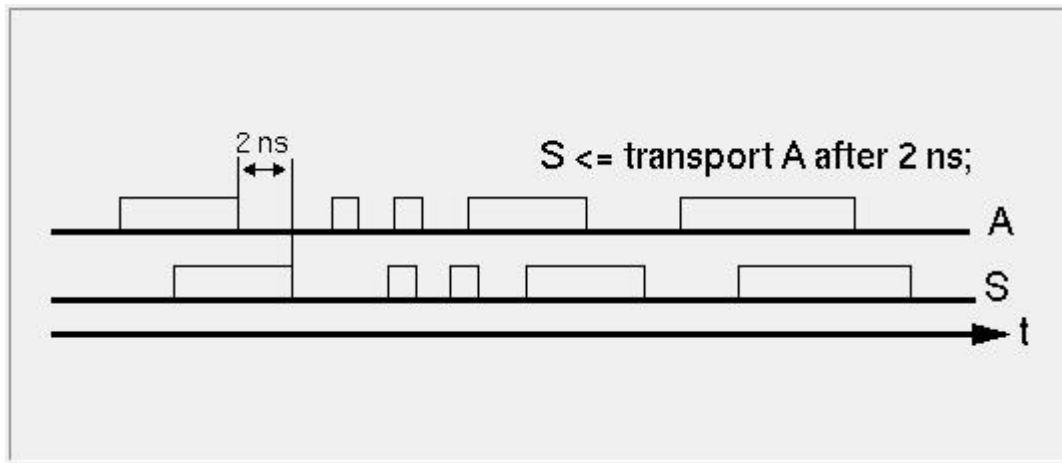
### 4.1.2.1. Transport delay

Eender welke verandering die aan een ingang plaatsvindt, zal men terugvinden aan de uitgang. Het sleutelwoord **transport** moet expliciet vermeld worden in het waveform-element.

Voorbeeld:

**S <= transport A after 2 ns;**

Dit geeft de volgende simulatieresultaten:



Situatie 1:

Indien bij transport delay de vertragingstijd in een toe te voegen waveform-element groter is dan alle vertragingen die reeds in de driver zijn opgenomen, dan wordt dit waveform-element achteraan de driver toegevoegd.

```

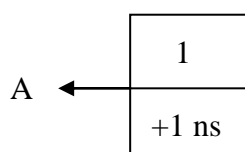
signal A: integer:=0;

P1: process
begin
    A <= transport 1 after 1 ns;           (1)
    A <= transport 2 after 2 ns;         (2)
    wait;
end process;

```

Bij het uitvoeren van dit proces worden er twee waveform-elementen aan de driver toegevoegd.

Driver na commando (1) :



Driver na commando (2) :

A ←	1	2
	+1 ns	+2 ns

### Situatie 2:

Indien de vertragingstijd in een toe te voegen waveform-element van een signaal kleiner is dan de vertraging van reeds opgenomen waveform-elementen in de driver van dat signaal, dan worden deze laatste elementen uit de driver verwijderd en het nieuwe element wordt achteraan toegevoegd.

```
signal A: integer:=0;
```

```
P2: process  
begin
```

```
    A <= transport 1 after 1 ns; 3 after 3 ns; 5 after 5 ns; (1)
```

```
    A <= transport 4 after 4 ns; (2)
```

```
    wait;
```

```
end process;
```

Driver na commando (1):

A ←	1	3	5
	+1 ns	+3 ns	+5 ns

Driver na commando (2):

A ←	1	3	4
	+1 ns	+3 ns	+4 ns

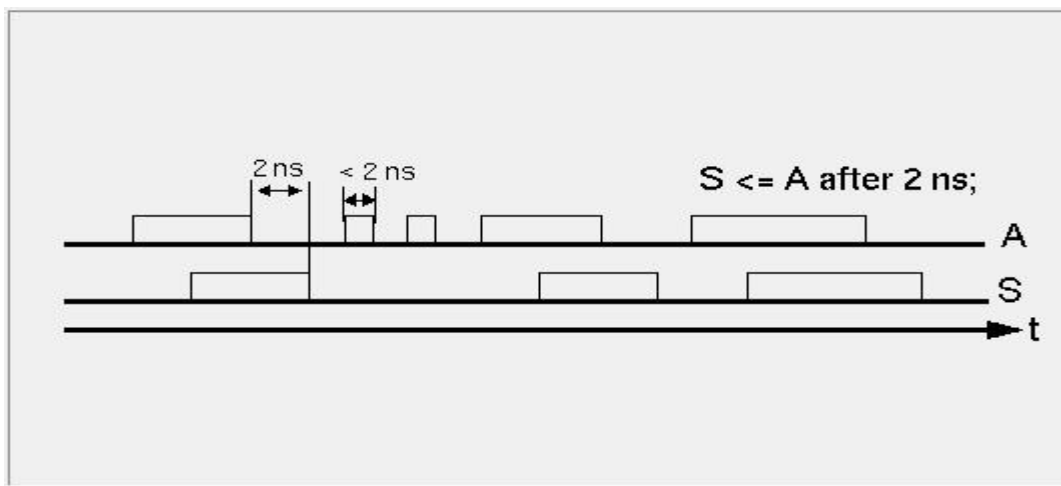
#### **4.1.2.2. Inertial delay**

Dit model is het standaardmodel van VHDL. Het geeft de waardeveranderingen door aan een signaal na de opgegeven vertragsperiode, op voorwaarde dat er gedurende de vertragsperiode geen nieuwe waardeveranderingen meer optreden.  
( spike-proof!)

Voorbeeld:

**S <= A after 2 ns;**

Dit geeft de volgende simulatieresultaten:



Situatie 1:

**signal A: integer:=0;**

**P3: process**

**begin**

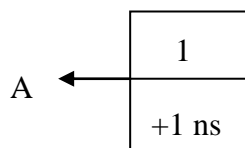
**A <= 1 after 1 ns;                   --(1)**

**A <= 2 after 2 ns;                   --(2)**

**wait;**

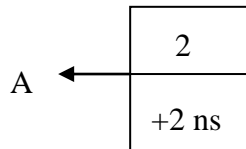
**end process;**

Driver na commando (1):



Driver na commando (2):





De waarde 2 uit het tweede commando is verschillend van de waarde 1 uit het eerste toekenningcommando. Daarom wordt het element (1, 1ns) uit de driver verwijderd en overschreven door (2, 2ns).

Algemeen geldt de volgende regel:

Indien bij inertial delay de waarde van een toe te voegen waveform-element verschillend is van de waarde van een waveform-element dat al vroeger in de driver werd opgenomen, dan wordt het vroegst opgenomen waveform-element uit de driver verwijderd, en het laatst aangebrachte ingeschreven. Indien de waarde niet verschillend is, dan blijven de toekenningen op de driver staan.

#### Situatie 2:

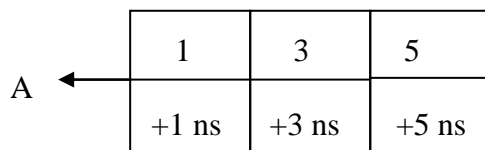
```

signal A: integer:=0;

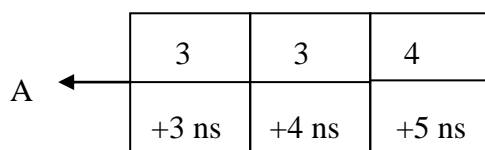
P4: process
begin
    A <= 1 after 1 ns; 3 after 3 ns; 5 after 5 ns;           --(1)
    A <= 3 after 4 ns; 4 after 5 ns;                       --(2)
    wait;
end process;

```

Driver na commando (1):



Driver na commando (2):



Het tweede commando verwijdert het waveform-element (1, 1ns) uit de driver omdat de waarde 1 verschillend is van de waarde 3. De toekenning (3, 3ns) blijft op de driver staan. De toekenning (5, 5ns) wordt verwijderd omdat de vertraging 5 ns groter is dan de vertraging van het eerste element (3, 4ns) van commando(2). Het element (4, 5ns) wordt achteraan toegevoegd.

Merk op dat alleen het eerste element uit een waveform met een inertial delay wordt verwerkt; voor de andere elementen geldt het transport-delay-model.

In het algemeen stellen we:

Indien bij inertial delay een toe te voegen waveform-element een vertraging heeft die kleiner is dan die van de reeds opgenomen waveform-elementen in de driver, dan worden deze laatste uit de driver verwijderd. Het nieuwe waveform-element wordt achteraan toegevoegd.

#### 4.1.2.3. Delta delay

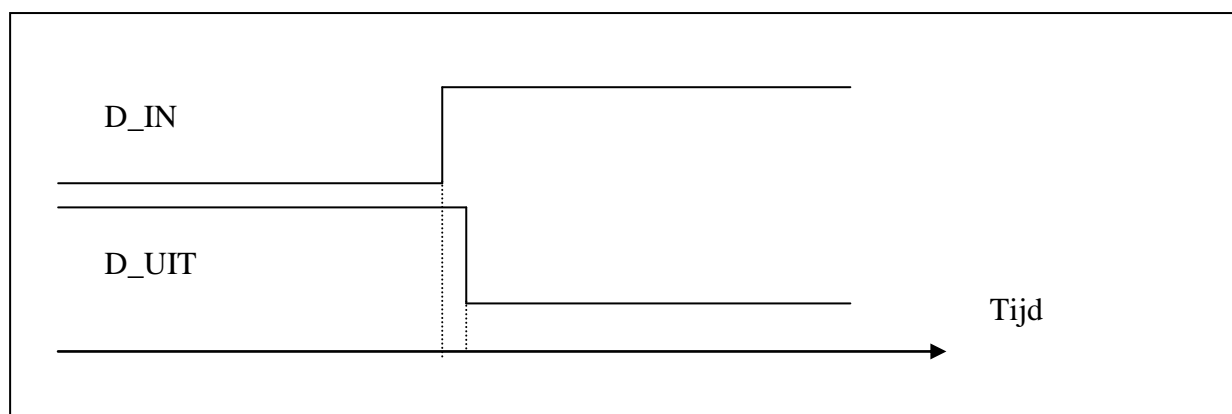
Het delta-delay-model wordt gebruikt indien géén tijdsvertraging wordt opgegeven in een waardetoekenning aan een signaal. Het is een speciaal vertragingmodel voor het simuleren van events zonder propagatietijden. De delta delay stelt een infinitesimaal kleine vertraging voor.

We geven als voorbeeld een eenvoudige waardetoekenning aan een signaal D\_UIT:

$D\_UIT \leq \text{not } D\_IN;$
----------------------------------

De driver na dit commando is:

D_UIT ←	1	0
	Nu	0 ns



Bij het uitvoeren van een simulatie worden waardetoekenningen met delta delay steeds als eerste verwerkt. Nieuwe waarden voor de signalen worden toegekend in een volgende simulatiecyclus.

## 4.2. Verloop van een VHDL-simulatie

Tijdens een VHDL-simulatie reageren processen op stimuli: zij worden geactiveerd door waardeveranderingen (events) van bepaalde signalen of datapaden op het actuele simulatietijdstip. Zonder stimuli verkeert een proces in een wachtttoestand.

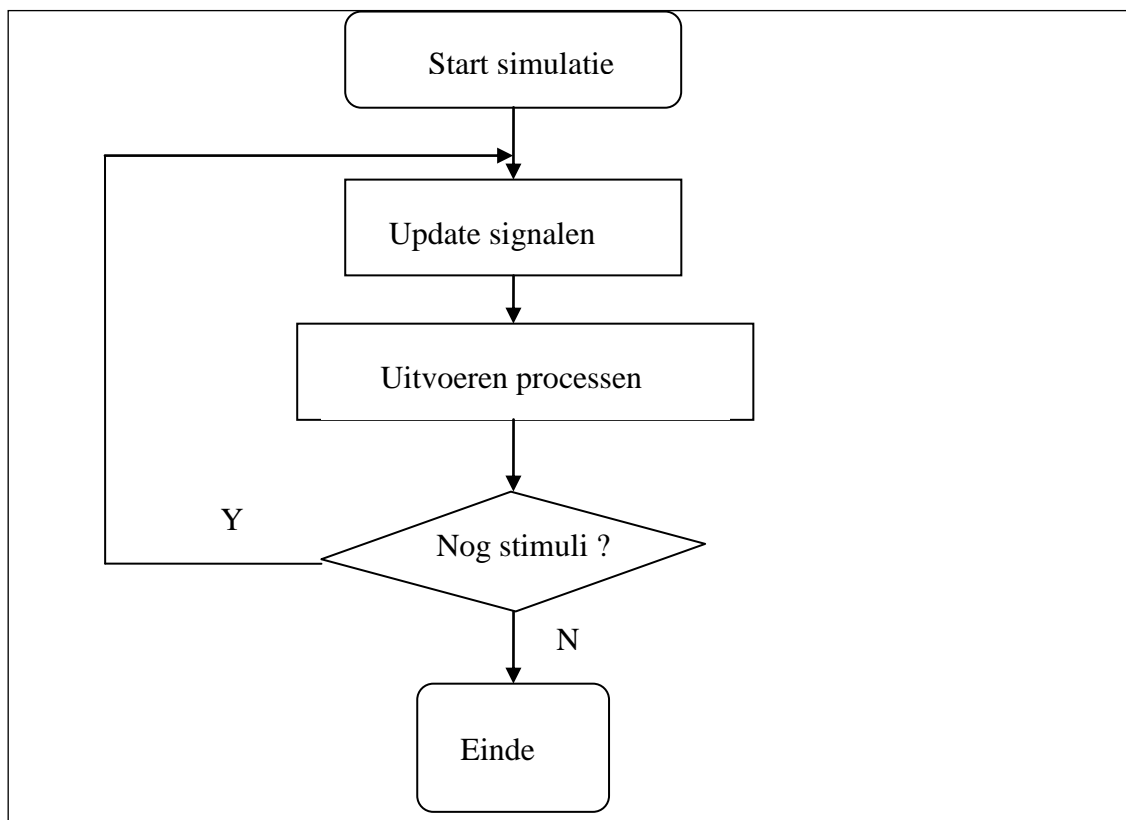
Door het activeren van een proces, via o.a. events, worden op het simulatietijdstip nieuwe waardeveranderingen van bepaalde signalen berekend.

Deze toekomstige waarden worden bewaard in de driver van het signaal.

De simulatie bestaat uit een reeks van simulatiecycli op opeenvolgende tijdstippen. De uitvoering van een simulatiecyclus op een bepaald tijdstip bestaat uit twee opeenvolgende fasen:

1. Eerst worden de nieuwe waarden die van kracht zijn op het actuele simulatietijdstip via de datapaden (signalen) overgebracht.
2. Daarna worden de processen die nieuwe informatie hebben ontvangen via hun zgn. sensitiviteitskanalen geactiveerd. De uitvoering van toekenningscommando's zorgen ervoor dat eventueel nieuwe events op de drivers geplaatst worden, al dan niet met een bepaalde vertraging. Als alle processen doorlopen zijn, dan begint een nieuwe simulatiecyclus.

Indien de nieuwe events met een delta-delay werden weggeschreven naar de driver, dan zal de simulatietijd met een waarde delta verhoogd worden. Indien er voor het actuele simulatie tijdstip geen nieuwe events gepland zijn dan wordt de simulatietijd verhoogd naar het volgende actieve tijdstip.



Voorbeeld:

```

library ieee;
use ieee.std_logic_1164.all;

entity demo_sim is
port( A, B: in std_ulogic;
      Y, Z: out std_ulogic);
end demo;

architecture voorbeeld of demo_sim is
signal X: std_ulogic;

begin
    process( A, B, X )
    begin
        Y <= A;
        X <= B;
        Z <= X;
    end process;
end voorbeeld;

```

Stel dat tijdens de eerste simulatiecyclus het proces geactiveerd wordt door een event op B. De signalen krijgen de volgende waarden:

Y krijgt de huidige waarde van signaal A.  
 X krijgt de nieuwe waarde van B.  
 Z krijgt de waarde van X namelijk de oude waarde van B.

X krijgt dus een nieuwe waarde. Daar X in de sensitivity lijst staat wordt het proces opnieuw geactiveerd: de tweede simulatiecyclus wordt gestart. De signalen krijgen de volgende waarden:

Y krijgt de huidige waarde van A ( geen verandering!).  
 X krijgt de huidige waarde van B ( geen verandering!).  
 Z krijgt de huidige waarde van X ( de nieuwe waarde van B).

Daar er geen verdere events gebeuren op A, B en X gaat het proces in een wachtoestand.

## 4.3. Testbenches

### Doel:

Een testbench wordt gebruikt om de functionaliteit van een ontwerp te testen.

Bij een goed ontwerp besteedt men tijd aan het uitdenken en uitwerken van testen die verzekeren dat het ontwerp functioneert zoals het moet.

In een testbench kan je systematisch alle te voorziene situaties implementeren, zodat je de zekerheid hebt dat de component onder alle omstandigheden zal werken.

Debuggen kan dus zonder de component in werkelijkheid te programmeren.

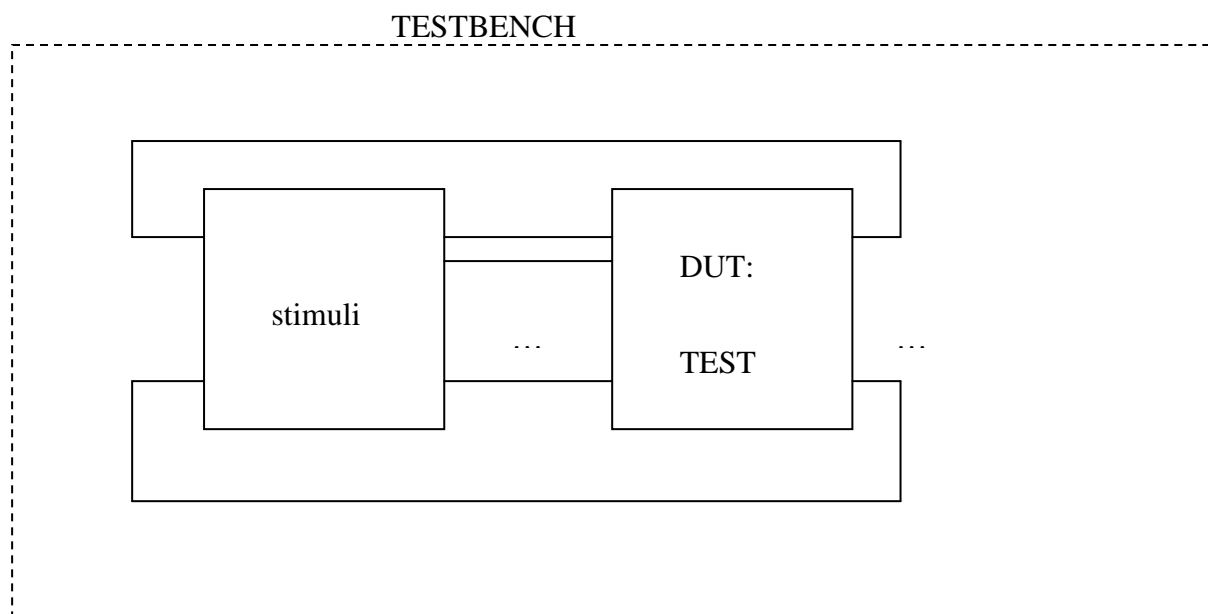
De testbench levert de stimuli aan het ontwerp of DUT (Device Under Test) , analyseert de manier waarop het ontwerp hierop reageert en stockeert indien gewenst de resultaten in een file. Het simulatietool Modelsim visualiseert de signalen in een waveform. Het is nu aan de ontwerper om deze signalen te beoordelen en te besluiten of de functionaliteit van het ontwerp klopt.

### Implementatie:

We creëren een nieuwe entity: TESTBENCH (van een hoger niveau)

Deze entity gebruikt het te testen blok als component (=DUT):

- ingangssignalen worden aangelegd
- uitgangssignalen worden opgemeten
- de verschillen tussen wat verwacht wordt en wat bekomen wordt, wordt gerapporteerd: naar het scherm toe of naar een tekst bestand.



### Voorbeeld:

```

entity testbench is                                --(1)
end testbench;

architecture stimuli of testbench is

  component TEST                                    --(2)
  port( in0, in1, in2, in3: in bit;
    uit: out bit);
  end component;

  signal in0, in1, in2, in3: bit;                  --(3)
  signal uit: bit;

  begin
  dut1: TEST port map (in0, in1, in2, in3, uit);    --(4)

  process
  begin
    in0 <='0';                                       --(5)
    in1 <='1' after 1 ns;
    wait for 10 ns;
    in2 <='0';
    in3 <='1' after 1 ns;
    wait for 10 ns;
    in0 <='1';
    in1 <='0' after 1 ns;
    wait for 10 ns;
  end process;

  end stimuli;

```

- (1) De entity van de testbench bevat geen port() instructie.
- (2) De architectuur vangt aan met het declareren van de te testen component.
- (3) Alle interne signalen die later zullen verbonden worden met de in en out signalen van het DUT worden gedeclareerd.  
Eventueel ook een file om resultaten in weg te schrijven.
- (4) Verbinding interne datapaden met de ports van de DUT.
- (5) Hier volgen processen die inputs aanleggen of outputs wegschrijven.  
Merk op dat een wait-statement noodzakelijk is om het proces te stoppen. Indien dit statement niet aanwezig was zou het proces steeds herstarten.

Clock stimulus :

De klok kan gegenereerd worden door een speciaal process:

```
klok: process  
begin  
    clk <= '1';  
    wait for 100 ns;  
    clk <= '0';  
    wait for 100 ns;  
end process klok;
```

of met een concurrent signaaltoekenning:

```
clk <= not clk after PERIOD/2;
```

#### Reset :

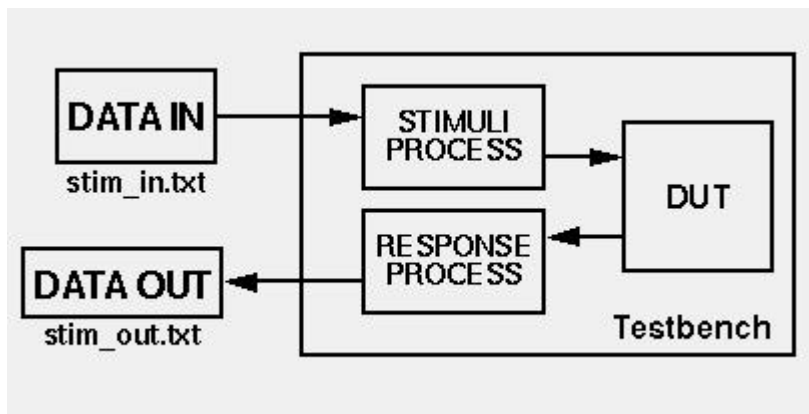
Het realiseren van een reset is eenvoudig: in het begin wordt reset geïnitieerd op '1', dan geactiveerd op '0' (indien actief lage reset) en ten slotte terug inactief gemaakt voor de rest van de simulatie: '1'.

Voorbeeld:

```
Reset <= '1';  
Reset <= '0' after 20 ns;  
Reset <= '1' after 40 ns;
```

## 4.4. FILE I/O

In VHDL is het mogelijk om gegevens uit een file te halen en/of gegevens naar een file te schrijven. Hiervoor is het noodzakelijk de TEXTIO package toe te voegen van de STD library. Dit maakt het mogelijk om stimuli te laten genereren door gespecialiseerde software als ook de simulatieresultaten te laten beoordelen.



De informatie wordt, lijn per lijn, uit de file gelezen ( **READLINE** procedure) en wordt in een variabele '**line**' gestockeerd.

Later is het mogelijk om deze data te lezen met het '**READ**' commando.

Meestal bevat een line informatie van verschillende datatypes. Deze moeten juist geïnterpreteerd worden ( een string als een string, een integer als een integer,... )

Het is noodzakelijk de spatie tussen de data met een apart read statement in te lezen.

Voorbeeld: de stimuli wordt via een file aangeboden.

```

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

entity read_file is                                --(1)
end read_file;

architecture test of read_file is
begin

    stimuli: process
        variable data: integer;
        variable naam: string( 1 to 4 );
        variable L-IN: line;
        file stim_in: text is in "stim_in.txt";
        begin
            ...
            while not endfile ( stim_in ) loop
                readline ( stim_in,L_IN );
                read ( L_IN, naam );
                read ( L_IN, data );
            end loop;
        wait;
    end process;

end test;

```

De file output gebeurt op equivalente wijze: een line wordt samengesteld met het '**WRITE**' commando en wordt uiteindelijk naar de file geschreven met het **WRITELINE** statement.



Voorbeeld: De resultaten van een process kunnen naar een log-file weggeschreven worden.

```

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

entity testbench is                                --(1)
end testbench;

architecture stimuli of testbench is

component TEST                                    --(2)
port( in0, in1, in2, in3: in bit;
      uit: out bit);
end component;

signal in0, in1, in2, in3: bit;                    --(3)
signal uit: bit;
file log: text is out "resultaat.std";

begin
dut1: TEST port map (in0, in1, in2, in3, uit);      --(4)

process
begin
.....
end process;

process ( in0, in1, in2, in3, output )
variable templine: line;
begin
    write( templine, now);                          --now: geeft huidige simulatietijd
    write(templine, ' ');
    write( templine, in0);
    write(templine, ' ');
    write( templine, in1);
    write(templine, ' ');
    write( templine, in2);
    write(templine, ' ');
    write( templine, in3);
    write(templine, ' ');
    write( templine, output);
    writeline( log, templine );
end process;
end stimuli;

```

## **BIBLIOGRAFIE**

1. IEEE Standard VHDL Language Reference Manual  
IEEE Std 1076 – 1987  
IEEE , New York, 1988
2. VHDL: Hardware Description and Design  
R. Lipsett, C.F. Schaeffer, C. Ussery  
Kluwer Academic Publishers, 1989  
Isbn 0-7923-9030-X
3. VHDL User's Manual, Volume 1: Tutorial  
Intermetrics Inc., 1985
4. Inleiding tot de VHDL-TAAL.  
Marijn Temmerman.
5. VHDL Tutorial  
Prof. Dr.-Ing. Wolfram H. Glauert

<b>1. VHDL</b>	<b>0</b>
1.1. Inleiding	1
1.2. VHDL-ontwerp	1
1.3. De verschillende abstractieniveaus	2
1.4. Voordelen van het gebruik van VHDL.	3
<b>2. Modelvorming in VHDL</b>	<b>3</b>
2.1. Algemeen	3
2.2. De entity declaratie.	5
2.3. Architecture body.	6
2.3.1 Gedragsbeschrijving of functionele beschrijving	6
2.3.2. Structuurbeschrijving.	8
2.3.3. Data-flowbeschrijving.	10
<b>3. De VHDL-taal en syntax.</b>	<b>12</b>
3.1. Lexicale conventies	12
3.1.1. Typografische tekens	12
3.1.2. Namen	12
3.1.3. Constanten	13
3.1.3.1. Decimale getallen	13
3.1.3.2 Getallen met een ander grondtal	13
3.1.3.3. Karakters	13
3.1.3.4. Constante 'strings'	13
3.1.3.5. Bitstring-constanten	13
3.1.3.6. Fysische constanten	14
3.1.4. Commentaar	14
3.2. Datatypes	14
3.2.1. Het scalaire type	15
3.2.1.1. Het opsommingstype (Enumeration type)	15
3.2.1.2. Het geheel-getaltype (integer type)	16
3.2.1.3. Het fysische type	16
3.2.1.4. Het bewegende-kommatype (floating point type)	17
3.2.2. Het samengestelde type	17
3.2.2.1. Het array-type	18
3.2.2.2. Het record-type	19
3.2.3. Concatenatie	20
3.2.4. Aggregaten	20
3.2.5. Partities van arrays	21
3.2.6. Het access-type	22
3.2.7. Het bestandstype (file type)	22
3.3. Extended Data Types	23
3.3.1. IEEE Standard Logic Type	23
3.3.2. Resolved en unresolved types	24
3.3.3. Subtypes en resolutiefuncties	25
3.4. Objecten in VHDL	26
3.5. Packages	26
3.5.1. Package declaratie	27
3.5.2. Package body	27
3.5.3. Voorbeeld	27
3.6. Libraries	28
3.7. Attributen	30
3.8. Voorgedefinieerde Operatoren	31

<b>3.9. Sequentiële Statements</b>	<b>32</b>
3.9.1. Toekenningsbevel voor variabelen	33
3.9.2. Toekenningsbevel voor signalen	33
3.9.3. Variabelen vs. signalen	33
3.9.4. IF statement	34
3.9.5. CASE Statement	35
3.9.6. For-lus	36
3.9.7. While lus.	37
3.9.8. Lus met exit conditie	38
3.9.9. WAIT statement	39
3.9.10. ASSERT statement.	40
3.9.11. Procedure oproep.	41
3.9.12. RETURN statement.	41
<b>3.10. Concurrent Statements</b>	<b>41</b>
3.10.1. Eenvoudig concurrent toekenningsbevel	41
3.10.2. Voorwaardelijk toekenningsbevel.	42
3.10.3. Selectief toekenningsbevel.	43
3.10.4. Het concurrent ASSERTION-bevel.	44
3.10.5. De concurrent PROCEDURE-oproep.	45
<b>3.11. RTL-Style</b>	<b>46</b>
3.11.1. Het PROCESS-commando	46
3.11.2. Het combinatorisch proces	47
3.11.2.1. Sensitivity-lijst.	47
3.11.2.2. WAIT Statement.	47
3.11.3. Het synchrone proces	48
3.11.4. Finite State Machines and VHDL	49
3.11.4.1. Algemene structuur.	49
3.11.4.2. State encoding.	50
3.11.4.3. Machine van Moore.	51
3.11.4.4. Mealy.	54
<b>3.12. Geavanceerde technieken</b>	<b>56</b>
3.12.1. Het gebruik van GENERICS.	56
3.12.2. Het GENERATE statement.	57
<b>4. Simulatie.</b>	<b>59</b>
<b>4.1. Tijdsmodel in VHDL.</b>	<b>59</b>
4.1.1. Waveforms en drivers.	59
4.1.2. Delay modellen.	60
4.1.2.1. Transport delay	60
4.1.2.2. Inertial delay	62
4.1.2.3. Delta delay	65
<b>4.2. Verloop van een VHDL-simulatie</b>	<b>66</b>
<b>4.3. Testbenches</b>	<b>68</b>
<b>4.4. FILE I/O</b>	<b>70</b>