

VHDL-AMS Essentiel - Table des matières

1. Conventions	2	6.5. Attributs sur des natures	8	13. Sous-programmes	14
2. Eléments de base du langage	2	6.6. Attributs sur des terminaux	8	13.1. Procédure	14
2.1. Identificateurs	2	6.7. Attributs sur des quantités	8	13.2. Fonction	14
2.2. Littéraux caractères (<i>character literals</i>)	2	7. Alias	8	13.3. Surcharge (<i>overloading</i>)	15
2.3. Littéraux chaînes de caractères (<i>string literals</i>)	2	8. Expressions et opérateurs	8	13.4. Sous-programmes dans un paquetage	15
2.4. Mots réservés	2	8.1. Opérateurs logiques	8	14. Instructions simultanées	15
2.5. Littéraux caractères (<i>character literals</i>)	2	8.2. Opérateurs relationnels	8	14.1. Instruction simultanée simple	15
2.6. Littéraux chaînes de caractères (<i>string literals</i>)	2	8.3. Opérateurs de décalage et de rotation	8	14.2. Instruction simultanée conditionnelle	16
2.7. Littéraux chaînes de bits (<i>bit string literals</i>)	2	8.4. Concaténation	8	14.3. Instruction simultanée sélective	16
2.8. Littéraux numériques (<i>numeric literals</i>)	2	8.5. Opérateurs arithmétiques	9	14.4. Instruction simultanée procédurale	16
2.9. Agrégats (<i>aggregates</i>)	2	9. Unités de conception (<i>design units</i>)	9	15. Spécification du pas limite d'intégration	16
2.10. Commentaires	3	9.1. Clause de contexte (<i>context clause</i>)	9	16. Instructions concurrentes	16
3. Types (<i>types</i>) et sous-types (<i>subtypes</i>)	3	9.2. Déclaration d'entité (<i>entity declaration</i>)	9	16.1. Processus (<i>process</i>)	16
3.1. Types numériques (<i>numeric types</i>)	3	9.3. Corps d'architecture (<i>architecture body</i>)	9	16.2. Affectation concurrente de signal	17
3.2. Types énumérés (<i>enumerated types</i>)	3	9.4. Déclaration de paquetage (<i>package declaration</i>)	10	16.3. Affectation concurrente conditionnelle de signal	17
3.3. Types physiques (<i>physical types</i>)	3	9.5. Corps de paquetage (<i>package body</i>)	10	16.4. Affectation concurrente sélective de signal	17
3.4. Types tableaux (<i>array types</i>)	3	9.6. Déclaration de configuration (<i>configuration decl.</i>)	10	16.5. Instance de composant (<i>component instance</i>)	18
3.5. Types enregistrements (<i>record types</i>)	4	10. Déclarations d'interface (<i>interface declarations</i>)	10	16.6. Génération d'instructions (<i>generate statements</i>)	18
3.6. Types accès (<i>access types</i>)	4	10.1. Constante d'interface (<i>interface constant</i>)	10	16.7. Instruction concurrente d'assertion	18
3.7. Types fichiers (<i>file types</i>)	4	10.2. Signal d'interface (<i>interface signal</i>)	10	16.8. Instruction concurrente break	18
4. Natures (<i>natures</i>) et sous-natures (<i>subnatures</i>)	5	10.3. Variable d'interface (<i>interface variable</i>)	11	17. Paquetages standard	19
4.1. Natures scalaires	5	10.4. Fichier d'interface (<i>interface file</i>)	11	17.1. Paquetage STANDARD	19
4.2. Natures composites	5	10.5. Terminal d'interface (<i>interface terminal</i>)	11	17.2. Paquetage TEXTIO	20
5. Objets (<i>objects</i>)	5	10.6. Quantité d'interface (<i>interface quantity</i>)	11	17.3. Paquetage STD_LOGIC_1164	21
5.1. Constantes (<i>constants</i>)	5	10.7. Association d'interface (<i>interface association</i>)	11	17.4. Paquetage MATH_REAL	22
5.2. Variables (<i>variables</i>)	5	11. Déclaration de composant	11	17.5. Paquetage ENERGY_SYSTEMS	26
5.3. Signaux (<i>signals</i>)	5	12. Instructions séquentielles	12	17.6. Paquetage ELECTRICAL_SYSTEMS	26
5.4. Fichiers (<i>files</i>)	6	12.1. Affectation de signal (<i>signal assignment</i>)	12	17.7. Paquetage MECHANICAL_SYSTEMS	26
5.5. Terminaux (<i>terminals</i>)	6	12.2. Instruction wait	12	17.8. Paquetage FLUIDIC_SYSTEMS	27
5.6. Quantités (<i>quantities</i>)	6	12.3. Affectation de variable (<i>variable assignment</i>)	12	17.9. Paquetage THERMAL_SYSTEMS	28
6. Attributs (<i>attributes</i>)	7	12.4. Instruction conditionnelle (<i>conditional signal ass.</i>)	12	17.10. Paquetage RADIANT_SYSTEMS	28
6.1. Attributs sur des (sous-)types scalaires	7	12.5. Instruction sélective (<i>selective signal assignment</i>)	12	17.11. Paquetages NUMERIC_BIT et NUMERIC_STD	29
6.2. Attributs sur des (sous-)types discrets	7	12.6. Instructions de boucle (<i>loop statements</i>)	13		
6.3. Attributs sur des tableaux	7	12.7. Instructions d'assertion et de rapport	13		
6.4. Attributs sur des signaux	7	12.8. Instruction break (<i>break statement</i>)	13		

VHDL-AMS Essentiel

1. Conventions

gras	mot réservé
ID	identificateur
[terme]	optionnel
{ terme }	répétition (zéro ou plus)
terme {...}	répétition (un ou plus)
terme terme	choix un parmi n
(),;	ponctuation telle quelle
noir	syntaxe VHDL 87/93/2001 (sauf exception)
vert	ajouté dans VHDL 93
bleu	ajouté dans VHDL-AMS 99 (=VHDL 93 +)
rouge	ajouté dans VHDL 2001
italique	mot anglais

2. Eléments de base du langage

2.1. Identificateurs

- Premier caractère alphabétique. Suivants alphabétiques ou numériques.
- Caractère "_" accepté sauf en début et en fin. Pas de caractères soulignés successifs.
- Longueur quelconque. Tous les caractères sont significatifs.
- Pas de distinction minuscules/majuscules.

Identificateurs légaux:

X F1234 VHDL1076 VhDL_1076 long_identificateur

Identificateurs légaux mais non distincts:

Alert ALERT alert

Identificateurs illégaux:

74LS00 Q_ A_Z _Q \$non no\$

2.2. Littéraux caractères (*character literals*)

Compatibles avec types prédéfinis BIT et CHARACTER.

Exemples: 'A' 'a' ' ' (espace) ''' (apostrophe) CR LF

2.3. Littéraux chaînes de caractères (*string literals*)

Compatibles avec types prédéfinis STRING, BIT_VECTOR.

Exemples:

"ceci est une chaîne de caractères" "avec un " apostrophe"

Note: "A" | 'A'.

2.4. Mots réservés

abs	access	across	after
aliasall	and	architecture	array
assert	attribute		
begin	block	body	break
buffer	bus		
case	component	configuration	constant
disconnect	downto		
else	elsif	end	entity
exit			
file	for	function	
generate	generic	group	guarded
if	impure	in	inertial
inout	is		
label	library	limit	linkage
literal	loop		
map	mod		
nand	nature	new	next
noise	nor	not	null
of	on	open	or
others	out		
package	port	postponed	procedural
procedure	process	protected	pure
quantity			
range	record	reference	register
reject	rem	report	return
rol	ror		
select	severity	signal	shared
sla	sll	spectrum	sra
srl	subnature	subtype	
terminal	then	through	to
tolerance	transport	type	
unaffected	units	until	use
variable			
wait	when	while	with
xnor	xor		

Un mot réservé encadré par "\" peut être utilisé comme identificateur. Exemple: \begin

2.5. Littéraux caractères (*character literals*)

Compatibles avec types prédéfinis BIT et CHARACTER.

Exemples: 'A' 'a' ' ' (espace) ''' (apostrophe) CR LF

2.6. Littéraux chaînes de caractères (*string literals*)

Compatibles avec types prédéfinis STRING, BIT_VECTOR.

Exemples:

"ceci est une chaîne de caractères" "avec un " apostrophe"

Note: "A" | 'A'.

2.7. Littéraux chaînes de bits (*bit string literals*)

Compatibles avec type prédéfini BIT_VECTOR.

Exemples (B/b: binaire, O/o: octal, X/x: hexadécimal):

"001110100000" B"001110100000" b"001110100000"

B"001_110_100_000" O"1640" o"1640" X"3A0" x"3a0"

X"3_A_0" = 928

2.8. Littéraux numériques (*numeric literals*)

Nombres entiers: 12 0 -99 1e6 -5E2 123_456

Nombres réels: 0.0 -4.56 1.076e3 2.5E2 3.141_593

Littéraux numériques dans d'autres bases (2 à 16 seulement):

2#110_1010# -- base 2

8#1640# -- base 8

16#CA# -- base 16

16#f.ff#e+2 -- exposant 16**2

10#1076#

2.9. Agrégats (*aggregates*)

Association de valeurs à des éléments d'un tableau (indices)

ou à des éléments d'un enregistrement (champs).

Exemples:

(1, 2, 3, 4, 5)

tableau de 5 éléments ou enregistrement de 5 champs;
association par position.

(Jour => 21, Mois => Sep, An => 1998)

(Mois => Sep, Jour => 21, An => 1998)

enregistrement à 3 champs; association par noms.

('1', '0', '0', '1', **others** => '0')

vecteur de bits = "10010000..."; association mixte.

VHDL-AMS Essentiel

((('X', '0', 'X'), ('0', '0', '0'), ('X', '0', '1'))
("X0X", "000", "X01"))

tableau à deux dimensions; association par position.

Note: Les associations par positions doivent précéder les associations par noms dans une association mixte.

L'association **others** doit être la dernière de l'agrégat.

2.10. Commentaires

Tout texte d'une ligne situé après les caractères "--".

Exemples:

pas un commentaire -- commentaire sur une ligne

-- commentaire sur

-- plusieurs

-- lignes

3. Types (types) et sous-types (subtypes)

Type: ensemble de valeurs et opérations associées.

Sous-type: type avec une restriction (contrainte) sur les valeurs du type de base. Ces restrictions peuvent être statiques (fixes) ou dynamiques (connues seulement au moment de la simulation). Un sous-type n'est pas un nouveau type, mais un type dérivé d'un type de base, donc compatible avec ce dernier.

Syntaxe: Déclaration de type et de sous-type

type nom-type [**is** définition-type] ;

subtype nom-sous-type **is**

[fonction-résolution]

nom-type [contrainte]

[**tolerance** nom-groupe-tolérance] ;

4(5) classes de types:

- Types **scalaires** (*scalar*): entier, réel, énuméré, physique.
- Types **composés** (*composite*): tableau, enregistrement.
- Types **accès** (*access*): accès (pointeur) à des objets d'un type donné.
- Types **fichiers** (*file*): fichier, séquence de valeurs d'un type donné.
- Types protégés (*protected types*): pour accès aux variables partagées (*shared variables*).

3.1. Types numériques (*numeric types*)

Types prédéfinis INTEGER et REAL:

type INTEGER **is range** *dépend de l'implantation*;

-- intervalle minimum [-2147483647,+2147483647]

type REAL **is range** *dépend de l'implantation*;

-- intervalle minimum [-1.0E38,+1.0E38]

-- minimum 64 bits selon IEEE Std 754 ou 854

Exemples de types numériques non prédéfinis:

-- types entiers

type byte **is range** 0 **to** 255; -- intervalle ascendant

type bit_index **is range** 31 **downto** 0; -- ou descendant

-- types réels:

type signal_level **is range** -15.0 **to** 15.0;

type probability **is range** 0.0 **to** 1.0;

Sous-types prédéfinis NATURAL et POSITIVE:

subtype NATURAL **is** INTEGER **range** 0 **to** INTEGER'HIGH;

subtype POSITIVE **is** INTEGER **range** 1 **to** INTEGER'HIGH;

Note: L'attribut 'HIGH désigne la plus grande valeur représentable pour le type annoté.

Note: La spécification d'un groupe de tolérance n'est possible que pour un sous-type du type REAL.

3.2. Types énumérés (*enumerated types*)

Ensembles ordonnés d'identificateurs ou de caractères distincts.

Types prédéfinis BIT, BOOLEAN et CHARACTER:

type BIT **is** ('0', '1'); -- caractères

type BOOLEAN **is** (FALSE, TRUE); -- identificateurs

type CHARACTER **is**

(...caractères imprimables et non imprimables...);

Exemples de types énumérés non prédéfinis:

type state **is** (idle, init, check, shift, add);

type mixed **is** (false, 'A', 'B', idle);

-- illustre la **surcharge** (*overloading*) et le mélange

-- de caractères et d'identificateurs

3.3. Types physiques (*physical types*)

Caractérisés par une **unité de base** (*base unit*), un intervalle de valeurs et une éventuelle collection de sous-unités.

Type prédéfini TIME:

type TIME **is range** *même que INTEGER*

units

fs; -- unité de base (femtoseconde)

ps = 1000 fs; -- picoseconde

ns = 1000 ps; -- nanoseconde

us = 1000 ns; -- microseconde

ms = 1000 us; -- milliseconde

sec = 1000 ms; -- seconde

min = 60 sec; -- minute

hr = 60 min; -- heure

end units;

Note: Le type TIME est essentiel pour la simulation. L'unité de base (1 fs) définit la **limite de résolution** (*resolution limit*) maximum de la simulation. Il est possible de définir une **unité secondaire** (*secondary unit*) pour la simulation qui est multiple de l'unité de base (ns par ex.), avec toutefois une perte possible de précision.

Note: Espace obligatoire entre le nombre et l'unité.

Sous-type prédéfini DELAY_LENGTH:

subtype DELAY_LENGTH **is** TIME

range 0 **to** INTEGER'HIGH;

3.4. Types tableaux (*array types*)

Collections de valeurs de même type de base sur un domaine d'indices (tableaux monodimensionnels) ou sur plusieurs domaines d'indices (tableaux multidimensionnels). Un domaine d'indices peut être **contraint** (*constrained*; intervalle d'indices défini) ou **non contraint** (*unconstrained*; intervalle d'indice ouvert).

Types tableaux prédéfinis BIT_VECTOR et STRING:

type BIT_VECTOR **is array** (NATURAL **range** <>) **of** BIT;

type STRING **is array** (NATURAL **range** <>)
of CHARACTER;

VHDL-AMS Essentiel

Type tableau prédéfini REAL_VECTOR:

```
type REAL_VECTOR is array (NATURAL range <> )  
  of REAL;
```

Note: La notation "<>" (*box*) dénote un tableau non contraint.

Exemples de types tableaux non prédéfinis:

```
type word is array (31 downto 0) of bit;  
  -- vecteur, intervalle descendant  
type address is NATURAL range 0 to 255;  
type memory is array (address) of word;  
  -- matrice, intervalle des lignes ascendant  
type truth_table is array (BIT, BIT) of BIT; -- matrice
```

Accès à un élément d'un tableau:

- Soit W un objet de type word. W(0), W(8), W(31) dénotent le bit respectif du mot. W(31 **downto** 16) dénote une *tranche* (*slice*), un sous-mot, constituée des 16 premiers bits du mot. W(0 **to** 15) est illégal ici (intervalle descendant seulement).
- Soit TT un objet de type truth_table. TT('0', '1') dénote l'élément de la première ligne et de la deuxième colonne de la matrice TT. TT('0')('1') est illégal ici.
- Soit M in objet de type memory. M(12)(15) dénote le bit 15 du mot d'adresse 12. M(12,15) est illégal ici.
- Utilisation d'agrégats. Exemple:

```
type charstr is array (1 to 4) of CHARACTER;
```

Accès par position: ('T', 'O', 'T', 'O')

Accès par nom: (1 => 'T', 2 => 'O', 3 => 'T', 4 => 'O')

Accès par défaut: (1 | 3 => 'T', **others** => 'O')

3.5. Types enregistrements (*record types*)

Collections de valeurs de types de bases potentiellement différents.

Exemple de type enregistrement:

```
type processor_operation is (op_load, op_add, ...);  
type mode is (none, indirect, direct, ...);  
type instruction is record
```

```
  opcode: processor_operation;  
  addrmode: mode;  
  op1, op2: INTEGER range 0 to 15;
```

```
end record instruction;
```

Accès à un *champ* (*field*) d'un enregistrement:

- Par sélection avec le caractère '.' (point). Soit inst un objet de type instruction, la notation inst.opcode fait référence au premier champ de l'enregistrement.
- Par agrégat. Exemples:
(opcode => op_add, addrmode => none, op1 => 2, op2 => 15)
(op_load, indirect, 7, 8)

3.6. Types accès (*access types*)

Pointeurs, références.

Type accès prédéfini LINE:

```
type LINE is access STRING;
```

Exemple de type accès non prédéfini:

```
type cell; -- déclaration incomplète de type  
type link is access cell;  
type cell is record  
  value: INTEGER;  
  succ: link;  
end record cell;
```

L'opérateur **new** permet d'allouer une zone mémoire:

```
new link -- crée un pointeur sur un enregistrement de type  
  -- cell et de valeur initiale (INTEGER'LEFT, null)  
new link'(15, null) -- initialisation explicite, l'agrégat doit  
  -- être qualifié (link')  
new link'(1, new link'(2, null)) -- allocation chaînée
```

Le littéral **null** dénote un pointeur sans référence.

Soit cellptr un objet de type link, la notation cellptr.all dénote le nom de l'enregistrement référencé. La notation sélective (ex.: cellptr.value, cellptr.succ) accède à un champ spécifique.

La procédure deallocat libère la place mémoire référencée et remet la valeur du pointeur à **null**:
deallocat (cellptr); -- cellptr = **null**

3.7. Types fichiers (*file types*)

Séquences de valeurs contenues dans un dispositif externe.

Type fichier prédéfini TEXT:

```
type TEXT is file of STRING;
```

Exemples de types fichiers non prédéfinis:

```
type str_file is file of STRING;  
type nat_file is file of NATURAL;
```

Opérations prédéfinies sur les fichiers:

```
procedure FILE_OPEN (file f: type-fichier;  
  external_name: in STRING;  
  open_kind: in FILE_OPEN_KIND := read_mode);  
procedure FILE_OPEN (status: out FILE_OPEN_STATUS;  
  file f: type-fichier;  
  external_name: in STRING;  
  open_kind: in FILE_OPEN_KIND := read_mode);  
procedure READ (file f: type-fichier;  
  value: out type-élément);  
function ENDFILE (file f: type-fichier) return BOOLEAN;  
procedure WRITE (file f: type-fichier;  
  value: in type-élément);  
procedure FILE_CLOSE (file f: type-fichier);
```

Note: En VHDL-87, l'ouverture et la fermeture d'un fichier sont implicites. Ouverture à la déclaration, fermeture lorsque la simulation quitte la zone de visibilité de la déclaration.

4. Natures (*natures*) et sous-natures (*subnatures*)

Nature: types des valeurs pouvant être accédées par l'intermédiaire des attributs d'un terminal.

Les attributs d'un terminal sont: le terminal de référence de la nature, le type "entre" (across type) et le type "dans" (through type) qui définiront indirectement les types des quantités de branches associées au terminal. Seuls les terminaux peuvent être déclarés comme appartenant à une nature. Une nature peut être scalaire (simple) ou composite.

Sous-nature: restriction (contrainte) sur les attributs d'une nature de base. En particulier, une sous-nature peut définir des groupes de tolérances pour les types **across** et **through** de la nature de base. Une sous-nature n'est pas une nouvelle nature, mais une nature dérivée d'une nature de base, donc compatible avec cette dernière.

Syntaxe: Déclaration de nature et de sous-nature

nature nom-nature **is** nature-scalaire | nature-composite ;

subnature nom-sous-nature **is**
nom-(sous-)nature [contrainte]
[**tolerance** nom-groupe-tolérance **across**
nom-groupe-tolérance **through**] ;

Note: Il n'existe pas de natures prédéfinies.

4.1. Natures scalaires

Définissent les types across et through et le nom du terminal de référence.

Syntaxe: Déclaration de nature scalaire

nature nom-nature **is**
nom-(sous-)type **across**
nom-(sous-)type **through**
nom-terminal **reference** ;

Exemple de nature scalaire pour les systèmes électriques:

subtype voltage **is** REAL **tolerance** "voltage";
subtype current **is** REAL **tolerance** "current";
nature electrical **is**
voltage **across**
current **through**
ground **reference**;

4.2. Natures composites

Définissent les attributs de nature pour des collections de terminaux sous la forme de tableaux ou d'enregistrements.

Syntaxe: Déclaration de nature composite

nature nom-nature **is array** (index) **of** nom-sous-nature;

nature nom-nature **is record**
nom-champ : nom-sous-nature ; { ... }
end record;

Exemples:

nature electrical_vector **is**
array (NATURAL range <>) **of** electrical;

nature electrical_bus **is record**
strobe : electrical;
databus: electrical_vector (0 to 7);
end record;

Note: Tous les champs d'une nature enregistrement doivent être d'une même nature scalaire ou dérivés de la même nature scalaire.

5. Objets (*objects*)

4(6) classes d'objets: constantes, variables, signaux, fichiers, terminaux, quantités. Une déclaration d'objet déclare un objet d'un certain type (d'une certaine nature pour un terminal).

5.1. Constantes (*constants*)

Une constante représente une valeur fixe qui ne peut être modifiée.

Syntaxe: Déclaration de constante

constant nom-constante {, ...} : sous-type
[:= expression] ;

Exemples:

constant PI: REAL := 3.141_593;
constant N: NATURAL := 4;
constant index_max: INTEGER := 10*N - 1;
constant delay: DELAY_LENGTH := 5 ns;
-- Utilisation d'aggrégats

constant null_bv: BIT_VECTOR(0 to 15) := (others => '0');
constant TT: truth_table := (others => (others => '0'));
constant instadd1: instruction :=
(opcode => op_add, addrmode => none,
op1 => 2, op2 => 15);
-- Constante à **valeur différée** (*deferred constant*)
-- seulement admise dans une déclaration de paquetage
-- sa valeur doit être définie dans le corps de paquetage
constant max_size: NATURAL;

5.2. Variables (*variables*)

Une variable représente une valeur modifiable par affectation (*assignment*). L'affectation a un effet immédiat.

Syntaxe: Déclaration de variable

[**shared**] **variable** nom-variable {, ...} : sous-type
[:= expression] ;

L'expression définit une valeur initiale. La valeur initiale par défaut d'une variable de type T est définie par T'LEFT.

Exemples:

variable count: NATURAL;
-- valeur initiale: count = 0 (= NATURAL'LEFT)
variable isHigh: BOOLEAN;
-- valeur initiale: isHigh = false (= BOOLEAN'LEFT)
variable currentState: state := idle; -- initialisation explicite
variable memory: bit_matrix(0 to 7, 0 to 1023);
-- si les éléments de la matrice sont du type BIT,
-- la valeur initiale est ((others => (others => '0'))

Le mot réservé **shared** permet la déclaration de **variables partagées** (*shared variables*), ou variables globales.

5.3. Signaux (*signals*)

Un signal représente une forme d'onde logique sous la forme d'une suite discrète de paires temps/valeur. L'affectation d'une valeur à un signal n'est **jamais** immédiate.

Syntaxe: Déclaration de signal

signal nom-signal {, ...} : sous-type [:= expression] ;

L'expression définit une valeur initiale. La valeur initiale par défaut d'un signal de type T est définie par T'LEFT. Un signal ne peut pas être d'un type fichier ou d'un type accès.

VHDL-AMS Essentiel

Exemples:

```
signal S: BIT_VECTOR(15 downto 0);
-- valeur initiale par défaut: (others => '0')
signal clk: BIT := '1'; -- valeur initiale explicite
```

5.4. Fichiers (files)

Une déclaration de fichier représente un stockage sur une mémoire de masse externe et dépendante du système d'exploitation.

Syntaxe: Déclaration de fichier

```
file nom-fichier : sous-type is [ in | out ] nom-externe ;
file nom-fichier { , ... } : sous-type
    [ [ open type-ouverture ] is nom-externe ;
```

Exemples:

```
type integer_file is file of INTEGER;
file file1: integer_file is "test.dat";
-- fichier local en mode lecture (in, défaut)
file file1: integer_file is out "test.dat";
-- mode écriture
file file1: integer_file;
-- fichier local en mode lecture (read_mode, défaut)
-- ouverture explicite requise (procédure FILE_OPEN)
file file2: integer_file is "test.dat";
-- mode lecture (défaut) et lien sur un nom externe
-- appel implicite de la procédure
-- FILE_OPEN(F => file2, external_name => "test.dat",
-- OPEN_KIND => read_mode)
file file3: integer_file open write_mode is "test.dat";
-- mode écriture avec lien sur nom externe
-- appel implicite de la procédure
-- FILE_OPEN(f => file3, external_name => "test.dat",
-- OPEN_KIND => write_mode)
```

5.5. Terminaux (terminals)

Un terminal ne prend pas de valeur, mais est d'une nature (§4) particulière. Il définit un point de connexion et de référence pour les quantités de branches. Les lois de conservation de l'énergie (lois de Kirchhoff pour les circuits électriques) doivent être satisfaites pour toutes les quantités de branches aux points de connexion définis par les terminaux.

Syntaxe: Déclaration de terminal

```
terminal nom-terminal { , ... } : nom-(sous-)nature ;
```

Exemple:

```
terminal t1, t2: electrical;
```

5.6. Quantités (quantities)

Représentent des fonctions à valeurs réelles du temps et correspondent à des inconnues dans le système d'équations induit par la description.

Trois variétés de quantités: les quantités libres (*free quantities*), les quantités de branches (*branch quantities*) et les quantités de sources (*source quantities*).

Syntaxe: Déclaration de quantité libre

```
quantity nom-quantité { , ... } : (sous-)type-réel
    [ := expression ] ;
```

Une quantité ne peut être que du type prédéfini REAL ou d'un type ou sous-type dérivé du type REAL. L'expression optionnelle doit s'évaluer à une valeur de type réel et définit la valeur initiale de la quantité. Par défaut, la valeur initiale d'une quantité est 0.0.

Note: La valeur initiale d'une quantité définie à sa déclaration n'est pas une condition initiale. Il s'agit plutôt d'une valeur de départ pour le calcul, usuellement itératif, de toutes les inconnues.

Note: Seules les quantités ont une valeur initiale par défaut égale à 0.0. Les autres objets de type réel (variables et signaux) ont toujours une valeur initiale par défaut égale à T'LEFT où T est le type de l'objet (p.ex. T'LEFT = -1.0e38 pour le type REAL).

Exemple de déclaration de quantités libres:

```
quantity q1, q2: real;
```

Syntaxe: Déclaration de quantité de branche

```
quantity [ nom-quantité { , ... } [ := expression ] across
    [ nom-quantité { , ... } [ := expression ] through ]
    terminal-plus [ to terminal-moins ] ;
```

Une quantité de branche définit les grandeurs "entre" et "dans" relativement à deux terminaux, le terminal plus et le terminal moins. Les types de quantités de branche déclarées sont définis par la nature associée aux terminaux (les deux terminaux doivent être de même nature).

Une quantité de branche "entre" (*across quantity*) est du type défini par l'aspect **across** de la nature.

Une quantité de branche "dans" (*through quantity*) est du type défini par l'aspect **through** de la nature.

Une déclaration de quantité de branche doit définir au moins une quantité **across** ou une quantité **through**.

L'expression optionnelle associée à une quantité de branche doit s'évaluer à une valeur réelle et définit la valeur initiale de la quantité (mais pas la condition initiale associée à la quantité). Le terminal moins est par défaut le terminal de référence de la nature du terminal plus.

La valeur d'une quantité de branche **across** entre le terminal plus T1 et le terminal moins T2 est

T1'REFERENCE - T2'REFERENCE.

Le sens du flot d'une quantité de branche **through** est du terminal plus au terminal moins.

La déclaration d'une quantité de branche **across** seule ne définit pas de branche topologique entre les terminaux plus et moins. La déclaration d'une quantité de branche **through** définit une branche topologique entre les terminaux plus et moins.

La déclaration de plusieurs quantités de branche **across** représentent des alias d'une même quantité (tension p.ex.).

La déclaration de plusieurs quantités de branche **through** définit plusieurs quantités différentes.

Exemples de déclarations de quantités de branche:

```
quantity v12 across i12 through t1 to t2;
quantity v12a, v12b across t1 to t2; -- alias
quantity i12a, i12b through t1 to t2; -- branches parallèles
quantity v1 across t1; -- v1 = t'REFERENCE (contre terre)
quantity i1 through t1; -- branche contre terre
```

Syntaxe: Déclaration de quantité de source spectrale et de bruit

```
quantity nom-quantité { , ... } : (sous-)type-réel
    spectrum expression-amplitude , expression-phase
    | noise expression-puissance ;
```

VHDL-AMS Essentiel

Une quantité de source spectrale définit un stimulus pour une analyse AC petits-signaux introduit par mot-clé **spectrum**. Le stimulus est défini par une expression pour le module et une expression pour la phase.

Exemple de déclaration d'une quantité de source spectrale:
quantity ac: voltage **spectrum** 1.0, 0.0;

Une quantité de source de bruit définit un stimulus pour une analyse de bruit introduit par le mot-clé **noise**. Le stimulus est défini par une expression pour la puissance de bruit.

Exemple de déclaration d'une quantité de source de bruit:
constant Kf: real := 1.0e-14;
quantity Flns: real **noise** Kf*Id/FREQUENCY;
-- bruit en 1/f (diode)

Les expressions de module, de phase et de puissance de bruit peuvent être n'importe quelle expression VHDL légale dont l'évaluation retourne une valeur du type de la quantité de source. En particulier, ces expressions peuvent inclure les noms d'autres quantités et faire appel à la fonction prédéfinie **FREQUENCY** qui retourne la valeur courante de la fréquence de simulation.

6. Attributs (*attributes*)

Un attribut représente une caractéristique associée à une **entité nommée** (*named entity*). Un certain nombre d'attributs sont prédéfinis pour des types, des intervalles, des valeurs, des signaux, des fonctions, **des natures, des terminaux, des quantités**.

Classes d'attributs: *T* (type), *V* (valeur), *I* (intervalle), *F* (fonction), *S* (signal), *Q* (quantité).

6.1. Attributs sur des (sous-)types scalaires

Soient *T* un type scalaire, *X* une valeur de ce type et *C* une chaîne de caractères:

T'LEFT (V) limite à gauche de *T*.
T'RIGHT (V) limite à droite de *T*.
T'LOW (V) limite inférieure de *T*.
T'HIGH (V) limite supérieure de *T*.

T'IMAGE(X) (F) représentation de *X* comme une chaîne de caractères.
T'VALUE(C) (F) valeur de type *T* représentée par *C*.

Exemples:

type address **is** INTEGER range 7 **downto** 0;
-- address'LOW = 0,
-- address'HIGH = 7
-- address'LEFT = 7
-- address'RIGHT = 0

6.2. Attributs sur des (sous-)types discrets

Soient *T* un type discret ou physique, *X* une expression de type *T* et *N* une expression de type entier:

T'POS(X) (F) position de *X* dans la déclaration de *T*.
T'VAL(N) (F) valeur de l'élément *X* à la position *N*.
T'SUCC(X) (F) prochaine valeur après *X* (dans l'ordre).
T'PRED(X) (F) valeur précédant *X* (dans l'ordre).

Exemples:

type level **is** ('U', '0', '1', 'Z');
-- level'POS('U') = 0
-- level'VAL(2) = '1'
-- level'SUCC('1') = 'Z'
-- level'PRED('0') = 'U'

6.3. Attributs sur des tableaux

Soient *A* un objet de type tableau et *N* une expression de type entier (défaut: *N*=1):

A'LEFT[(N)] (F) limite à gauche de l'intervalle [indice *N*].
A'RIGHT[(N)] (F) limite à droite de l'intervalle [indice *N*].
A'LOW[(N)] (F) limite inf. de l'intervalle [indice *N*].
A'HIGH[(N)] (F) limite sup. de l'intervalle [indice *N*].
A'RANGE[(N)] (I) **A'LEFT[(N)]** **to** **downto** **A'RIGHT[(N)]**.
A'REVERSE_RANGE[(N)] (I) **A'RIGHT[(N)]** **to** **downto** **A'LEFT[(N)]**.
A'LENGTH[(N)] (V) longueur de la dimension *N*.
A'ASCENDING[(N)] (V) **A'RIGHT[(N)]** >= **A'LEFT[(N)]**?

Exemples:

type word **is** array (31 **downto** 0) of BIT;
type memory **is** array (7 **downto** 0) of word;
variable mem: memory;
-- mem'LOW = 0
-- mem'HIGH = 7
-- mem'LEFT = 7
-- mem'RIGHT = 0
-- mem'RANGE = 7 **downto** 0
-- mem'REVERSE_RANGE = 0 **to** 7
-- mem'LENGTH = 8
-- mem'LENGTH(2) = 32
-- mem'RANGE(2) = 31 **downto** 0
-- mem'HIGH(2) = 31
-- mem'ASCENDING(1) = FALSE

6.4. Attributs sur des signaux

Soient *S* un objet de classe signal et *T* une expression de type TIME (défaut: *T* = 0 fs):

S'STABLE[(T)] (S) vaut TRUE si aucun événement ne s'est produit depuis *T*; FALSE sinon.
S'DELAYED[(T)] (S) signal équivalent à *S*, mais retardé de *T* unités de temps.
S'TRANSACTION (S) vaut '1' lorsque *S* est actif; FALSE sinon.
S'EVENT (F) retourne TRUE si un événement s'est produit sur *S*; FALSE sinon.
S'LAST_EVENT (F) retourne le temps (type TIME/REAL) écoulé depuis le dernier événement.
S'LAST_VALUE (F) retourne la valeur du signal lors du dernier événement sur le signal.

S'DRIVING (F) vaut TRUE si une transaction est définie sur *S*; FALSE sinon.

S'DRIVING_VALUE (F) valeur de la transaction.

S'RAMP[(TR,TF)]

(Q) quantité de même type que *S* dont la valeur suit celle de *S* mais avec une pente à chaque changement de valeur de *S*. *TR* (*TF*) définit un temps de montée (descente). Si *TF* est omis, *TF*=*TR*. Si *TR* et *TF* sont omis, la discontinuité induite est implicitement notifiée au simulateur.

S'SLEW[(SR,SF)]

(Q) quantité de même type que *S* dont la valeur suit celle de *S* mais avec une pente à chaque

VHDL-AMS Essentiel

changement de valeur de S. SR (SF) définit une pente de montée (descente). Si SF est omis, SF=SR. Si SR et SF sont omis, la discontinuité induite est implicitement notifiée au simulateur.

6.5. Attributs sur des natures

Soit N une (sous-)nature:

N'ACROSS (T) type **across** de la nature N.
N'THROUGH (T) type **through** de la nature N.

6.6. Attributs sur des terminaux

Soit T un terminal d'une nature N:

T'REFERENCE (Q) quantité "entre" (across);
terminal plus: T; terminal moins: terminal de référence de la nature N.

T'CONTRIBUTION (Q) quantité "dans" (through);
valeur représente la somme des contributions des quantités de branche "dans" incidentes au terminal T.

T'TOLERANCE (V) nom du groupe de tolérance de T.

6.7. Attributs sur des quantités

Soit Q une quantité:

Q'DOT (Q) dérivée temporelle première de Q.

Q'INTEG (Q) intégrale temporelle de Q de t = 0 à l'instant courant.

Q'DELAYED[(T)] (Q) Q(NOW - T), T ≤ 0.0; défaut: T = 0.0.

Q'ABOVE(E) (S) signal de type boolean valant TRUE si Q-E > 0.0 et valant FALSE si Q-E < 0.0; E est une expression dont l'évaluation est une valeur de même type que Q.

Q'LTF(NUM,DEN) (Q) transformée de Laplace de Q. NUM et DEN sont du type REAL_VECTOR (coeff. des polynômes). DEN(0) ≠ 0.0.

Q'ZOH(T,TD)] (Q) Q (TD + k*T), k ≤ 0 (échantillonneur idéal de période T avec maintien). TD: délai initial (0.0 par défaut).

Q'ZTF(NUM,DEN,T,TD)] (Q) transformée en z de Q. NUM et DEN sont du type REAL_VECTOR (coeff. des polynômes). DEN(0) ≠ 0.0. T et TD: voir Q'ZOH.

Q'SLEW[(SR[,SF)]

(Q) quantité de même type que Q dont la valeur suit celle de Q mais avec une pente à chaque changement de valeur de Q. SR (SF) définit une pente de montée (descente). Si SF est omis, SF=SR. Si SR et SF sont omis, la discontinuité induite est implicitement notifiée au simulateur.

Q'TOLERANCE (V) nom du groupe de tolérance de Q.

7. Alias

Une déclaration d'alias permet de définir des noms équivalents pour des **entités nommées** (named entity).

Syntaxe: Déclaration d'alias

alias nom-alias [: sous-type] **is** non-objet ;

Exemples:

variable real_number: BIT_VECTOR(0 to 31);

alias sign: bit **is** real_number(0);

alias mantissa: BIT_VECTOR(23 downto 0) **is** real_number(8 to 31);

8. Expressions et opérateurs

Expression = termes reliés par des opérateurs.

Opérateurs prédéfinis (dans l'ordre décroissant de leur niveau de précédance):

**	abs	not			
*	/	mod	rem		
+	-	(unaires)			
+	-	&			
sll	srl	sla	sra	rol	ror
=	/=	<	<=	>	>=
and	or	nand	nor	xor	xnor

8.1. Opérateurs logiques

and or nand nor xor xnor

Termes légaux: Objets ou expressions de types scalaires ou tableaux mono-dimensionnels de types BIT ou BOOLEAN.

Court-circuits (short-circuit operations) possibles pour types scalaires. L'opérande de droite n'est pas évalué si l'opérande

de gauche vaut '1' ou TRUE (**and**, **nand**) ou '0' ou FALSE (**or**, **nor**). Opérations élément par élément pour types tableaux.

8.2. Opérateurs relationnels

= /= < <= > >=

Termes légaux: Objets ou expressions de mêmes types. Les opérateurs "=" et "/=" ne peuvent pas avoir des opérandes de types fichiers. Les autres opérateurs doivent avoir des opérandes de types scalaires ou tableaux mono-dimensionnels à base de type discret (entier ou énuméré).

8.3. Opérateurs de décalage et de rotation

sll srl sla sra rol ror

Termes légaux: Objets ou expressions binaires avec opérande de gauche de type tableau mono-dimensionnel de types BIT ou BOOLEAN et opérande de droite de type entier.

Exemples:

b"10001010" sll 3 = b"01010000"

b"10001010" srl -2 = b"00100010" -- équivalent à srl 2

b"10001010" sll 0 = b"10001010"

b"10010111" srl 2 = b"00100101"

b"10010111" srl -6 = b"11000000" -- équivalent à srl 6

b"01001011" sra 3 = b"00001000"

b"10010111" sra 3 = b"11110010"

b"00001100" sla 2 = b"00110000"

b"00010001" sla 2 = b"01000111"

b"10010011" rol 1 = b"00100111"

b"10010011" ror 1 = b"11001001"

8.4. Concaténation

L'opérateur de concaténation "&" ne s'applique qu'aux tableaux mono-dimensionnels.

Exemples:

constant BY0: byte := b"0000" & b"0000"; -- "00000000"

constant C: BIT_VECTOR := BY0 & BY0; -- C'length = 8

variable A: BIT_VECTOR(7 downto 0);

A := '0' & A(A'LEFT downto 1); -- équivalent à A srl 1

A := A(A'LEFT-1 downto 0) & '0'; -- équivalent à A sll 1

VHDL-AMS Essentiel

8.5. Opérateurs arithmétiques

**	abs	not	
*	/	mod	rem
+	-	(unaires)	
+	-	&	

Termes légaux:

- **"**"** (multiplication), **"/"** (division): opérandes de types entier, réel ou physique.
- **mod** (modulo), **rem** (reste): opérandes de types entiers.
- **abs** (valeur absolue): opérande de type numérique.
- **"**"** (puissance): opérande de gauche de type entier ou réel, opérande de droite de type entier (si <0, l'opérande de gauche doit être de type réel).
- **"+"**, **"-"**: opérande(s) de type(s) entier ou réel.

9. Unités de conception (*design units*)

L'unité de conception est le plus petit module compilable séparément dans une **bibliothèque** (*library*). Cinq unités de conception: déclaration d'entité, corps d'architecture, déclaration de configuration, déclaration de paquetage et corps de paquetage.

9.1. Clause de contexte (*context clause*)

Définit une ou plusieurs bibliothèques et/ou un ou plusieurs chemins d'accès aux entités nommées disponibles.

Syntaxe: Clause de contexte

```
{ library nom-bibliothèque {, ...} ;  
  | use nom-sélectif {, ...} ; }
```

Exemple de déclaration de bibliothèques:

```
library IEEE, CMOS_LIB;
```

Les noms de bibliothèques déclarés sont des noms **logiques**. L'association avec des emplacements physiques (p.ex. des répertoires Unix) doit être faite dans l'environnement des outils VHDL utilisés.

Deux bibliothèques sont prédéfinies:

- **WORK**: Bibliothèque de travail dans laquelle les unités de conceptions compilées sont stockées.
- **STD**: Bibliothèque contenant uniquement les paquetages **STANDARD** et **TEXTIO**.

L'accès aux entités nommées d'une unité de conception est spécifié par un chemin de la forme:

nom-bibliothèque.nom-unité.nom-simple

la clause **use** permet d'éviter de devoir spécifier le chemin complet.

Exemples:

```
use IEEE.STD_LOGIC_1164;  
-- le préfixe STD_LOGIC_1164 est requis pour utiliser une  
-- entité nommée du paquetage STD_LOGIC_1164  
-- ex.: type STD_LOGIC_1164.STD_LOGIC  
use IEEE.STD_LOGIC_1164.all; -- préfixe non requis
```

La clause de contexte

```
library STD, WORK;  
use STD.STANDARD.all;
```

est implicitement déclarée avant toute unité de conception.

9.2. Déclaration d'entité (*entity declaration*)

Une déclaration d'entité définit l'interface (paramètres, signaux) d'un modèle de composant et des caractéristiques communes à toute réalisation (architecture) de ce modèle.

Syntaxe: Déclaration d'entité

```
[ clause-de-contexte ]  
entity nom-entité is  
  [ generic ( constante-interface {, ...} ) ; ]  
  [ port ( signal-interface {, ...} ) ; ]  
  { déclaration }  
[ begin  
  { instruction-concurrente } ]  
end [ entity ] [ nom-entité ] ;
```

Déclarations admises: type, sous-type, constante, signal, **variable partagée**, (sous-)nature, quantité, terminal, limitation du pas, fichier, alias, attribut, sous-programme (en-tête et corps), clause **use**.

Instructions concurrentes admises: instruction concurrente d'assertion, appel concurrent de procédure, processus. Ces deux dernières instructions doivent être **passives** (lecture des valeurs d'objets - signaux, **variables globales** - seulement).

Exemples:

```
entity ent is  
  generic (N: positive := 1);  
  port (s1, s2: in BIT; s3: out BIT_VECTOR(0 to N-1));  
begin  
  assert s1 /= s2 report "s1 /= s2" severity ERROR;  
end entity ent;  
entity dac is  
  generic (vref: REAL := 5.0);  
  port (signal din : in BIT;  
        terminal aout: electrical);  
end entity dac;  
entity multiplier is  
  generic (bound: REAL);  
  port (quantity qin1, qin2 : in REAL;  
        quantity qout : out REAL;  
        signal outOfBound: out BOOLEAN);  
end entity multiplier;
```

9.3. Corps d'architecture (*architecture body*)

Un corps d'architecture définit une réalisation d'un modèle de composant.

Syntaxe: Corps d'architecture

```
[ clause-de-contexte ]  
architecture nom-architecture of nom-entité is  
  { déclaration }  
[ begin  
  { instruction-concurrente | instruction-simultanée } ]  
end [ architecture ] [ nom-architecture ] ;
```

Déclarations admises: type, sous-type, constante, signal, **variable partagée**, (sous-)nature, quantité, terminal, limitation du pas, fichier, alias, attribut, sous-programme (en-tête et corps), clause **use**.

Instructions concurrentes **et simultanées** admises: toutes.

Exemples:

```
architecture arch of ent is  
  constant delay TIME := 10 ns;  
  signal s: BIT;
```

VHDL-AMS Essentiel

```
begin
  s <= s1 and s2;
  s3 <= (0 to N-2 => '0') & s after delay;
end architecture arch;
architecture sfg of multiplier is
begin
  qout == qin1 * qin2;
  outOfBound <= qout'above(bound)
    or not qout'above(-bound);
end architecture sfg;
```

9.4. Déclaration de paquetage (package declaration)

Une déclaration de paquetage groupe des déclarations qui peuvent être utilisées par d'autres unités de conception.

Syntaxe: Déclaration de paquetage

```
[ clause-de-contexte ]
package nom-paquetage is
{ déclaration }
end [ package ] [ nom-paquetage ] ;
```

Déclarations admises: type, sous-type, constante, signal, **variable partagée**, (sous-)nature, terminal, fichier, alias, attribut, sous-programme (en-tête seulement), clause **use**.

Exemple:

```
package pkg is
  constant max: INTEGER := 10;
  constant max_size: NATURAL; -- constante différée
  subtype bv10 is BIT_VECTOR(max-1 downto 0);
  procedure proc (A: in bv10; B: out bv10);
  function func (A, B: in bv10) return bv10;
end package pkg;
```

9.5. Corps de paquetage (package body)

Un corps de paquetage contient les définitions des déclarations incomplètes de la déclaration de paquetage correspondante.

Syntaxe: Corps de paquetage

```
[ clause-de-contexte ]
package body nom-paquetage is
{ déclaration }
end [ package body ] [ nom-paquetage ] ;
```

Déclarations admises: type, sous-type, constante, **variable partagée**, fichier, alias, attribut, sous-programme (en-tête et corps), clause **use**.

Exemple:

```
package body pkg is
  constant max_size: NATURAL := 200;
  procedure proc (A: in bv10; B: out bv10) is
  begin
    B := abs(A);
  end procedure proc;
  function func (A, B: in bv10) return bv10 is
    variable V: bv10;
  begin
    V := A and B;
    return (not(V));
  end function func;
end package body pkg;
```

9.6. Déclaration de configuration (configuration decl.)

Une déclaration de configuration définit les associations (*binding*) entre les instances de composants d'un modèle et les entités de conception (paires entité/architecture) actuelles. La déclaration de configuration peut prendre des formes assez complexes. On se limite ici à la version la plus courante.

Syntaxe: Déclaration de configuration

```
[ clause-de-contexte ]
configuration nom-configuration of nom-entité is
  for nom-architecture
  { for spécification-composant
    indication-association ;
  end for; }
  end for;
end [ configuration ] [ nom-configuration ] ;
```

Syntaxe: Spécification de composant

nom-instance {, ...} | **others** | **all** : nom-composant

Syntaxe: Indication d'association

```
use entity nom-entité [ ( nom-architecture ) ]
[ generic map ( liste-association-générique ) ]
[ port map ( liste-association-port ) ]
```

La **spécification de composant** identifie l'instance de composant concernée par la configuration.

Le mot réservé **others** dénote toutes les instances de composants non encore identifiées. Le mot réservé **all** dénote toutes les instances d'un même composant.

L'**indication d'association** (*binding indication*) définit les correspondances entre des instances de composants et les entités de conception actuelles. Elle définit aussi les correspondances entre paramètres génériques formels et actuels et entre ports formels et actuels.

Exemple:

```
configuration conf of ent2 is
  for arch2
    for c: comp use entity WORK.dff(a3)
      port map (clk, d, q, qb);
    end for;
    for all: comp2 use entity WORK.dff(a2)
      generic map (tprop => 2 ns)
      port map (clk, d, q, qb);
    end for;
  end for;
end configuration conf;
```

10. Déclarations d'interface (interface declarations)

Une déclaration d'interface est utilisée dans une déclaration d'entité, une déclaration de composant et une déclaration de sous-programme.

10.1. Constante d'interface (interface constant)

Une constante d'interface peut être un paramètre générique ou un argument de sous-programme.

Syntaxe: Déclaration de constante d'interface

```
[ constant ] nom-constante {, ... } : [ in ] (sous-)type
[ := expression ]
```

10.2. Signal d'interface (interface signal)

Un signal d'interface peut être un port ou un argument de sous-programme.

Syntaxe: Déclaration de signal d'interface

[**signal**] nom-signal { , ... } : [**in** | **out** | **inout** | **buffer**]
(sous-)type [:= expression]

Le mode d'un signal d'interface peut être:

- **in**: le signal ne peut qu'être lu. C'est le seul mode admis pour un argument de fonction.
- **out**: le signal ne peut qu'être écrit (une valeur lui est affectée).
- **inout**: le signal peut être lu et écrit. Le signal peut être affecté de plusieurs sources différentes.
- **buffer**: le signal peut être lu et écrit. Le signal ne peut être affecté que par une seule source. Ce mode n'est pas admis pour un argument de procédure.

10.3. Variable d'interface (*interface variable*)

Une variable d'interface ne peut être qu'un argument de sous-programme.

Syntaxe: Déclaration de variable d'interface

[**variable**] nom-variable { , ... } : [**in** | **out** | **inout**]
(sous-)type [:= expression]

Le mode d'une variable d'interface peut être:

- **in**: la variable ne peut qu'être lue. C'est le seul mode admis pour un argument de fonction.
- **out**: la variable ne peut qu'être écrite (une valeur lui est affectée).
- **inout**: la variable peut être lue et écrite. Elle peut être affectée de plusieurs sources.

10.4. Fichier d'interface (*interface file*)

Une fichier d'interface ne peut être qu'un argument de sous-programme.

Syntaxe: Déclaration de fichier d'interface

file nom-fichier { , ... } : (sous-)type

Note: VHDL-87 déclare un fichier d'interface comme une variable d'interface de mode **in** (le fichier est lu) ou **out** (le fichier est écrit).

10.5. Terminal d'interface (*interface terminal*)

Un terminal d'interface ne peut être qu'un port.

Syntaxe: Déclaration de terminal d'interface

terminal nom-terminal { , ... } : (sous-)nature

Exemple: voir §9.2.

10.6. Quantité d'interface (*interface quantity*)

Unq quantité d'interface ne peut être qu'un port.

Syntaxe: Déclaration de quantité d'interface

quantity nom-quantité { , ... } : [**in** | **out**] (sous-)type-réel
[:= expression]

Le mode d'une quantité d'interface peut être:

- **in**: la valeur de la quantité est seulement lue (pas d'équation associée).
- **out**: une équation est associée à la quantité.

L'expression optionnelle doit s'évaluer à une valeur de type réel et définit la valeur initiale de la quantité. Par défaut, la valeur initiale d'une quantité est 0.0.

Exemples: voir §9.2.

10.7. Association d'interface (*interface association*)

Une association d'interface établit une correspondance entre une **partie formelle** (*formal part*) et une **partie effective** (*actual part*).

Syntaxe: Association d'interface

[partie formelle =>] partie-effective

La partie formelle est le nom d'une déclaration d'interface. La partie effective peut être une expression, un nom de signal, un nom de variable ou le mot-clé **open**.

L'**association par nom** (*named association*) utilise une partie formelle explicite. L'**association par position** (*positional association*) n'utilise que la partie effective. La partie formelle correspondante est déduite de la position de l'élément associé dans la liste d'interface.

11. Déclaration de composant

Une déclaration de composant (*component declaration*) ne définit pas une nouvelle unité de conception, mais définit plutôt l'**empreinte** (*template, socket*) d'une entité de conception qui doit être instanciée dans le modèle. Cette empreinte correspond à l'interface de l'entité de conception (paramètres génériques et ports).

Syntaxe: Déclaration de composant

component nom-composant [**is**]
[**generic** (constante-interface { , ... }) ;]
[**port** (signal-interface { , ... }) ;]
end component [**nom-composant**] ;

Exemples:

```
component flipflop is
  generic (Tprop, Tsetup, Thold: TIME := 0 ns);
  port (clk, d: in BIT; q: out BIT);
end component flipflop;

component capacitor is
  generic (val: REAL);
  port (terminal tp, tn: electrical);
end entity capacitor;

component multiplier is
  generic (bound: REAL);
  port (quantity qin1, qin2 : in REAL;
        quantity qout : out REAL;
        signal outOfBound: out BOOLEAN);
end component multiplier;
```

12. Instructions séquentielles

Une instruction séquentielle (*sequential statement*) ne peut apparaître que dans un processus, dans un corps de sous-programme ou dans un corps d'instruction simultanée procédurale.

12.1. Affectation de signal (*signal assignment*)

Pas admise dans un corps d'instruction simultanée procédurale.

Syntaxe: Affectation de signal

[**étiquette** :] nom-signal <= [mode-délai] forme-onde ;

Le mode de délai permet de spécifier un mode *inertiel* ou un mode *transport*. Le mode inertiel est le mode par défaut.

Syntaxe: Mode de délai

transport | [**reject** temps-rejection] **inertial**

Le temps de rejection est une valeur de type TIME. Il doit être positif et plus petit ou égal au délai inertiel.

Syntaxe: Forme d'onde

valeur | expression [**after** expression-temps] { , ... }

L'expression temps est par défaut égale à 0 ns. Les expressions temps d'une forme d'onde doivent être spécifiées par valeurs croissantes des temps. Un signal ne prend jamais sa nouvelle valeur immédiatement, mais seulement après un certain délai. Ce délai est celui spécifié par l'expression temps. C'est un **délai delta** (*delta delay*) si l'expression temps est égale à 0 ns. Le délai inertiel est défini par l'expression temps de la première clause **after** de la forme d'onde.

Exemples:

```
A <= B after 5 ns, not B after 10 ns; -- délai inertiel de 5 ns
A <= inertial B after 5 ns;
A <= reject 2 ns inertial B after 5 ns;
A <= transport B after 5 ns;
S <= A xor B after 5 ns;
```

12.2. Instruction wait

Permet de synchroniser un processus avec son environnement. Pas admise dans un corps d'instruction simultanée procédurale.

Syntaxe: Instruction wait

```
[ étiquette : ] wait
[ on nom-signal { , ... } ]
[ until expression-boléenne ]
[ for expression-temps ] ;
```

L'instruction **wait** seule stoppe un processus indéfiniment.

La forme **wait on** synchronise un processus sur un événement sur un ou plusieurs signaux. Ces signaux définissent une **liste de sensibilité** (*sensitivity list*). Exemple:

```
wait on S1, S2;
```

La forme **wait until** synchronise un processus sur une condition. Exemple:

```
wait until clk = '1'; -- équivalent à:
wait on clk until clk = '1';
```

Si l'instruction contient une liste de sensibilité explicite, la condition est seulement testée lorsqu'un événement survient sur l'un des signaux de la liste. Exemple:

```
wait on reset until clk = '1';
-- le signal clk ne fait pas partie de la liste de sensibilité
```

Si l'expression booléenne ne contient pas de signal, l'instruction est équivalente à un **wait** seul, donc le processus est suspendu indéfiniment! Exemple (où stop est une variable):

```
wait until stop = TRUE; -- équivalent à wait seul
```

La forme **wait for** permet de spécifier un temps de suspension pour un processus. Exemple:

```
wait for 15 ns;
```

Si l'instruction contient en plus une liste de sensibilité ou une condition, le processus peut être réactivé plus tôt. Exemple:

```
wait until trigger = '1' for 20 ns; -- est équivalent à:
wait on trigger until trigger = '1' for 20 ns;
```

12.3. Affectation de variable (*variable assignment*)

L'affectation d'une valeur à une variable est dénotée par ":=". La variable prend sa nouvelle valeur immédiatement.

Syntaxe: Affectation de variable

[**étiquette** :] nom-variable := expression ;

Exemples:

```
count := (count + 1)/2;
currentState := init;
```

12.4. Instruction conditionnelle (*conditional signal ass.*)

Syntaxe: Affectation séquentielle conditionnelle de signal

```
[ étiquette : ] if expression-boléenne then
    instruction-séquentielle
{ elsif expression-boléenne then
    instruction-séquentielle }
[ else
    instruction-séquentielle ]
end if [ étiquette : ] ;
```

Exemple:

```
if A > B then
    Max := A;
elsif A < B then
    Max := B;
else
    Max := INTEGER'LOW;
end if;
```

12.5. Instruction sélective (*selective signal assignment*)

Syntaxe: Affectation séquentielle sélective de signal

```
[ étiquette : ] case expression is
    when choix { | choix } => instruction-séquentielle { ... }
    { when choix { | choix } => instruction-séquentielle { ... } }
end case [ étiquette : ] ;
```

Voir §16.4 au sujet de l'expression de sélection et des choix. Une clause **when others** finale est requise si tous les choix possibles ne sont pas énumérés précédemment.

VHDL-AMS Essentiel

Exemple:

```
case int is -- int est du type INTEGER
  when 0      => V := 4; S <= '1' after 5 ns;
  when 1 | 2 | 7 => V := 6; S <= '1' after 10 ns;
  when 3 to 6  => V := 8; S <= '1' after 15 ns;
  when 9       => null; -- pas d'opération
  when others  => V := 0; S <= '0'; --autres cas
end case;
```

12.6. Instructions de boucle (loop statements)

Syntaxe: Instructions de boucle

```
[ étiquette : ]
[ while condition | for identificateur in intervalle ] loop
  { instruction-séquentielle }
end loop [ étiquette ] ;
```

L'indice d'une boucle **for** n'a pas besoin d'être déclaré et il est visible dans le corps de la boucle.

Les deux instructions **exit** et **next** permettent de contrôler le comportement de la boucle. L'instruction **next** (resp. **exit**) stoppe l'itération courante, éventuellement sous condition, et continue la boucle à l'itération suivante (resp. à la première instruction après la boucle). Le saut peut être aussi fait à l'instruction dont l'étiquette est indiquée.

Syntaxe: Instructions next et exit

```
[ étiquette : ] next | exit
[ étiquette-boucle ] [ when expression-booléenne ] ;
```

Exemples:

<pre>-- boucle infinie: loop wait until clk = '1'; q <= d after 5 ns; end loop; -- boucle while: while i < str'LENGTH and str(i) /= ' ' loop next when i = 5; -- saut à l'indice suivant i := i + 1; end loop;</pre>	<pre>-- boucle générale avec sortie: L: loop exit L when value = 0; value := value / 2; end loop L; -- boucle for: L1: for i in 15 downto 0 loop L2: for j in 0 to 7 loop exit L1 when i = j; -- saut à l'indice suivant -- dans la boucle L1 tab(i,j) := i*j + 5; end loop L2; end loop L1;</pre>
--	--

12.7. Instructions d'assertion et de rapport

L'instruction d'assertion (*assert statement*) et l'instruction de rapport (*report statement*) permettent de vérifier des conditions et de produire des messages si ces conditions ne sont pas vérifiées.

Syntaxe: Instruction d'assertion

```
[ étiquette : ] assert condition
[ report message ] [ severity niveau-sévérité ] ;
```

Niveaux de sévérité: NOTE, WARNING, ERROR, FAILURE.

Exemples:

```
assert initial_value <= max_value;
-- message produit par défaut: "Assertion violation"
-- niveau de sévérité par défaut: ERROR
assert char >= '0' and char <= '9'
  report "Nombre " & input_str & " incorrect"
  severity WARNING;
assert FALSE
  report "Message toujours produit" severity NOTE;
```

Syntaxe: Instruction de rapport

```
[ étiquette : ] report message [ severity niveau-sévérité ] ;
```

Exemple:

```
report "Message toujours produit";
-- niveau de sévérité par défaut: NOTE
```

Note: Le comportement du simulateur en fonction du niveau de sévérité n'est pas défini. Chaque simulateur fournit un moyen d'indiquer le niveau de sévérité à partir duquel la simulation doit s'arrêter (normalement ERROR).

Note: L'usage d'instructions d'assertion ou de rapport dans le corps d'une instruction simultanée procédurale est admis, mais leurs nombres d'exécutions dépend du simulateur utilisé.

12.8. Instruction break (break statement)

Notifie l'existence d'une discontinuité de la solution des équations induites par les instructions simultanées. Permet aussi de spécifier des conditions initiales sur des quantités.

Syntaxe: Instruction break

```
[ étiquette : ] break [ liste-éléments ] [ when condition ] ;
```

La liste des éléments liste-éléments spécifie les quantités concernées par l'instruction **break**.

Syntaxe: Liste des éléments de l'instruction break

[for nom-quantité use] nom-quantité => expression { , ... }

La quantité spécifiée dans la clause **for ... use** est la quantité de sélection. L'autre quantité spécifiée est la quantité de break. La quantité de sélection est par défaut identique à la quantité de break.

Note: En l'absence d'une quantité de sélection, la quantité de break ne peut être que le préfixe Q d'une quantité implicite Q'DOT apparaissant dans le modèle ou une quantité implicite Q'INTEG.

L'instruction **break** n'est exécutée que si la condition mentionnée dans la clause **when** est vraie ou s'il n'y a pas de condition.

13. Sous-programmes

Deux types de sous-programmes: les procédures et les fonctions. Chacune d'elles encapsule une série d'instructions séquentielles. Une procédure est une instruction à part entière, alors qu'une fonction est une expression retournant un résultat.

13.1. Procédure

Syntaxe: Déclaration de procédure

```
procedure nom-procédure [ ( liste-param-interface ) ] is
{ déclaration }
begin
{ instruction-séquentielle }
end [ procedure ] [ nom-procédure ] ;
```

Déclarations locales admises: type, sous-type, constante, variable et sous-programme. Contrairement à un processus, les déclarations sont élaborées à nouveau (p. ex. les variables sont recrées) à chaque appel de procédure.

Paramètres d'interface admis: constante, variable, signal et fichier. Le §10 donne la syntaxe des paramètres d'interface.

L'**appel de procédure** (*procedure call*) est une instruction séquentielle ou concurrente selon l'endroit où il est effectué.

Syntaxe: Appel de procédure

```
[ étiquette : ]
nom-procédure [ ( liste-association-paramètres ) ] ;
```

La syntaxe de la liste d'association des paramètres a été donnée au §10.7.

Exemple:

```
procedure p (f1: in t1; f2: in t2; f3: out t3; f4: in t4 := v4) is
begin
...
end procedure p;
-- appels possibles (a = paramètre effectif):
p(a1, a2, a3, a4);
p(f1 => a1, f2 => a2, f4 => a4, f3 => a3);
p(a1, a2, f4 => open, f3 => a3);
p(a1, a2, a3);
```

L'**appel concurrent de procédure** (*concurrent procedure call*) est équivalent à un processus contenant le même appel de procédure et sensible aux paramètres signaux effectifs de mode **in** ou **inout**.

Exemple:

```
-- déclaration:
procedure proc (signal S1, S2: in BIT;
               constant C1: INTEGER := 5) is
begin
...
end procedure proc;
-- appel concurrent:
Appel_Proc: proc(S1, S2, C1);
-- processus équivalent:
Appel_Proc: process
begin
  proc(S1, S2, C1);
  wait on S1, S2;
end process Appel_Proc;
```

13.2. Fonction

Une fonction calcule et retourne un résultat qui peut être directement utilisé dans une expression.

Syntaxe: Déclaration de fonction

```
[ pure | impure ]
function nom-fonction
[ ( liste-paramètres-interface ) ] return (sous-)type is
{ déclaration }
begin
{ instruction-séquentielle }
end [ function ] [ nom-fonction ] ;
```

Déclarations admises: type, sous-type, constante, variables et sous-programme. Contrairement à un processus, les déclarations sont élaborées à nouveau (p. ex. les variables sont recrées) à chaque appel de fonction.

Paramètres d'interface admis: constante, variable, signal et fichier. Le §10 donne la syntaxe des paramètres d'interface. Une fonction n'admet que des paramètres en mode **in** (c'est le mode par défaut).

Une fonction retourne la valeur calculée au moyen d'une instruction **return**. L'exécution d'une instruction **return** termine l'appel de la fonction.

Syntaxe: Instruction return

```
[ étiquette : ] return expression ;
```

L'instruction **return** n'est pas admise dans le corps d'une instruction simultanée procédurale.

Une fonction est dite **pure** si elle ne fait aucune référence à une variable ou à un signal déclaré dans l'environnement dans lequel la fonction est appelée. En d'autres mots, une fonction pure n'a pas d'effets de bord. C'est le type de fonction par défaut. Le mot réservé **impure** permet de définir des fonctions avec effets de bord.

L'**appel de fonction** (*function call*) n'est pas une instruction en elle-même, mais une expression.

Syntaxe: Appel de fonction

```
[ étiquette : ] nom-fonction [ ( liste-association-param ) ] ;
```

La syntaxe de la liste d'association des paramètres a été donnée au §10.7.

Exemple:

```
function limit (value, min, max, gain: INTEGER)
return INTEGER is
  variable val: INTEGER;
begin
  if value > max then
    val := max;
  elsif value < min then
    val := min;
  else
    val := value;
  end if;
  return gain * val;
end function limit;
```

-- Usages:

```
new_value := limit(value, min => 10, max => 100, gain => 2);
new_speed := old_speed + scale * limit(error, -10, +10, 2);
```

13.3. Surcharge (overloading)

La surcharge est un mécanisme permettant de définir plusieurs sous-programmes ayant la même fonction et le même nom, mais agissant sur des paramètres de nombres et de types différents.

Exemples:

```
-- trois procédures de conversion vers un nombre entier:
procedure convert (r: in REAL; result: out INTEGER) is ...
procedure convert (b: in BIT; result: out INTEGER) is ...
procedure convert (bv: in BIT_VECTOR;
    result: out INTEGER) is ...

-- trois fonctions de calcul du minimum:
function min (a, b: in INTEGER) return INTEGER is ...
function min (a: in REAL; b: in REAL) return REAL is ...
function min (a, b: in BIT) return BIT is ...
```

L'appel de la procédure ou de la fonction définira quel sous-programme il faut exécuter en fonction des paramètres actuels, plus précisément en fonction de leur nombre, leurs types, et l'ordre dans lequel ils sont déclarés.

Il est aussi possible de surcharger des opérateurs prédéfinis tels que "+", "-", **and**, **or**, etc.

Exemple:

```
type logic4 is ('0', '1', 'X', 'Z');
function "and" (a, b: in logic4) return logic4 is ...
function "or" (a, b: in logic4) return logic4 is ...
```

13.4. Sous-programmes dans un paquetage

Un sous-programme placé dans un paquetage peut être décomposé en deux parties. Une partie, la déclaration de sous-programme, se trouve dans la déclaration de paquetage. L'autre partie, le corps de sous-programme se trouve dans le corps de paquetage.

Exemple d'un paquetage définissant un sous-type vecteur de 32 bits et ses opérations associées:

```
package bv32_pkg is
    subtype word32 is BIT_VECTOR(31 downto 0);
```

```
procedure add (
    a, b: in word32;
    result: out word32; overflow: out BOOLEAN);
function "<" (a, b: in word32) return BOOLEAN;
... -- autres déclarations de sous-programmes
end package bv32_pkg;
package body bv32_pkg is
    procedure add (
        a, b: in word32;
        result: out word32; overflow: out BOOLEAN) is
        ... -- déclarations locales
    begin
        ... -- corps de la procédure
    end procedure add;
    function "<" (a, b: in word32) return BOOLEAN is
        ... -- déclarations locales
    begin
        ... -- corps de la fonction
    end function "<";
    ... -- autres corps de sous-programmes
end package body bv32_pkg;
```

14. Instructions simultanées

Une instruction simultanée (simultaneous statement) représente une équation qui contraint les valeurs de quantités. Cinq formes d'instructions simultanées: l'instruction simultanée simple, l'instruction simultanée conditionnelle, l'instruction simultanée sélective, l'instruction simultanée procédurale et l'instruction simultanée nulle. Une instruction simultanée ne peut apparaître que dans un corps d'architecture.

14.1. Instruction simultanée simple

Syntaxe: Instruction simultanée simple

```
[ étiquette : ]
expression == expression [ tolerance groupe-tolérance ] ;
```

Elle définit une relation entre quantités qui doit être satisfaite pour n'importe quelle valeur du temps (variable indépendante). Les expressions à gauche et à droite du signe "==" sont des expressions VHDL légalles s'évaluant à des valeurs réelles.

Les expressions peuvent inclure des quantités, des constantes, des littéraux, des signaux est des appels de fonctions.

Une instruction simultanée simple doit au moins inclure une quantité.

Une référence à une quantité composite de type tableau ou enregistrement définit un ensemble d'instructions simultanées simples à raison d'une instruction par indice du tableau ou champ de l'enregistrement.

L'instruction est symétrique: les expressions à gauche et à droite du signe "==" peuvent être échangées sans que cela change les résultats de la simulation.

Une spécification de tolérance optionnelle permet de modifier le groupe de tolérance par défaut associé à l'instruction. Un groupe de tolérance est une chaîne de caractères. Si l'instruction est de la forme "Q == expression", le groupe de tolérance associé à la quantité Q (dérivée de son sous-type) est celui de l'instruction. Sinon, si l'instruction est de la forme "expression == Q", le groupe de tolérance associé à la quantité Q est celui de l'instruction. Sinon, l'instruction n'est associée à aucun groupe de tolérance et une spécification de tolérance locale doit exister.

Exemple:

```
entity diode is
    generic (IS0: REAL := 1.0e-14; TAU, RD: REAL := 0.0);
    port (terminal anode, cathode: electrical);
end entity diode;
architecture simple of diode is
    constant VT: REAL := 0.0258;
    quantity vd across id, ic through anode to cathode;
    quantity qd: REAL; -- groupe de tolérance anonyme
begin
    id == IS0 * (exp(vd - RD*id)/VT - 1.0);
    qd == TAU * id tolerance "charge";
    ic == qd'dot;
end architecture simple;
```

14.2. Instruction simultanée conditionnelle

Syntaxe: Instruction simultanée conditionnelle

```
[ étiquette : ]
if condition use
    { instruction-simultanée }
elsif condition use
    { instruction-simultanée } }
else
    { instruction-simultanée } }
end use [ étiquette ] ;
```

Le nombre d'instructions simultanées apparaissant dans chaque branche de l'instruction peut être quelconque. Il n'est pas nécessaire qu'il soit le même pour chaque branche.

14.3. Instruction simultanée sélective

Syntaxe: Instruction simultanée sélective

```
[ étiquette : ]
case expression use
    when choix { | ... } => { instruction-simultanée }
    { when choix { | ... } => { instruction-simultanée } }
end case [ étiquette ] ;
```

Le nombre d'instructions simultanées apparaissant dans chaque branche de l'instruction peut être quelconque. Il n'est pas nécessaire qu'il soit le même pour chaque branche.

14.4. Instruction simultanée procédurale

L'instruction simultanée procédurale offre un style séquentiel de notation pour exprimer des équations différentielles algébriques.

Syntaxe: Instruction simultanée procédurale

```
[ étiquette : ]
procedural [ is ]
    { déclaration }
begin
    { instruction-séquentielle }
end procedural [ étiquette ] ;
```

Les déclarations locales suivantes sont admises: type, sous-type, constante, variable, alias, sous-programme (en-tête et corps), clause **use**.

Note: Les variables locales ne conservent pas leur valeur entre les évaluations successives des instructions séquentielles.

Les instructions séquentielles suivantes ne sont pas admises dans le corps d'une instruction simultanée procédurale: instruction **return**, instruction **wait**, instruction **break**, affectation de signal.

Les quantités référencées dans le corps de l'instruction simultanée procédurale sont localement traitées comme des variables.

Le manuel de référence de VHDL-AMS définit un ensemble de transformations telles que toute instruction simultanée procédurale puisse se ramener à une instruction simultanée simple incluant un appel de fonction dont le corps évalue la séquence des instructions de l'instruction simultanée procédurale.

Exemple:

```
architecture proc of diode is
    constant VT: REAL := 0.0258;
    quantity vd across id, ic through anode to cathode;
    quantity qd: REAL;
begin
    P1: procedural
    begin
        id := IS0 * (exp(vd - RD*id)/VT - 1.0);
        qd := TAU * id;
        ic := qd'dot;
    end procedural P1;
end architecture proc;
```

-- instruction simultanée simple équivalente:
(id, qd, ic) == P1_func(vd, id, ic, qd, qd'dot);

15. Spécification du pas limite d'intégration

La spécification **limit** permet de définir le temps maximum entre deux calculs successifs des valeurs des quantités du modèle.

Syntaxe: Spécification limit

limit nom-quantité { , ... } : type **with** expression ;
limit others | **all with** expression ;

L'expression doit s'évaluer à une valeur réelle positive et non nulle. Les mots-clés **others** et **all** permettent d'affecter une limite à respectivement toutes les quantités non encore affectées ou à toutes les quantités du modèle.

Une spécification **limit** ne peut apparaître que dans la même région que la (les) quantité(s) concernée(s), c'est-à-dire une déclaration d'entité pour une quantité d'interface ou la région déclarative d'une architecture pour une quantité libre ou de branche.

16. Instructions concurrentes

Une instruction concurrente (*concurrent statement*) ne peut apparaître que dans une déclaration d'entité ou un corps d'architecture. Toute instruction concurrente possède une forme séquentielle équivalente utilisant un processus.

16.1. Processus (process)

Un processus définit une portion de code dont les instructions sont exécutées en séquence dans l'ordre donné. Chaque processus s'exécute de manière asynchrone par rapport aux autres processus et aux instances de composants.

Syntaxe: Processus

```
[ étiquette : ]
[ postponed ] process [ ( nom-signal { , ... } ) ] [ is ]
    { déclaration }
begin
    { instruction-séquentielle }
end [ postponed ] process [ étiquette ] ;
```

La liste optionnelle de signaux entre parenthèses après le mot réservé **process** définit la **liste de sensibilité** (*sensitivity list*) du processus. Un événement sur l'un de ces signaux a pour conséquence une activation du processus et une exécution de ses instructions.

Un processus ayant une liste de sensibilité ne peut pas contenir d'instructions **wait**. Equivalence:

```
process (S1, S2, ...)
begin
    -- instr. séquentielles
end process;

process
begin
    -- instr. séquentielles
    wait on S1, S2, ...;
end process;
```

Déclarations admises: type, sous-type, constante, variable, fichier, alias, sous-programme (en-tête et corps), clause **use**. Les variables locales conservent leur valeur d'une activation du processus à une autre.

Instructions séquentielles admises: assertion, conditionnelle (**if**), de sélection (**case**), de boucle (**loop**, **next**, **exit**), affectation de signal et de variable, de synchronisation (**wait**), appel de procédure.

Le mot réservé **postponed** définit un processus qui ne s'exécute que lorsque tous les processus qui n'ont pas cet attribut ont exécuté leur dernier cycle delta.

16.2. Affectation concurrente de signal

Trois types d'affectation concurrente de signal existent: l'affectation simple, l'affectation conditionnelle et l'affectation sélective.

Syntaxe: Affectation concurrente de signal simple

```
[ étiquette : ] [ postponed ]
    nom-signal <= [ mode-délai ] forme-onde ;
```

Syntaxe: Processus équivalent de l'affectation concurrente de signal simple

```
[ étiquette : ] [ postponed ]
process ( signaux-de-la-forme-d'onde )
begin
    nom-signal <= [ mode-délai ] forme-onde ;
end process [ étiquette ] ;
```

Éléments de l'affectation de signal simple: voir §12.1.

Exemple:
S <= A **xor** B **after** 5 ns;

```
-- processus équivalent:
process (A, B)
begin
    S <= A xor B after 5 ns;
end process;
lbl: data <= "11" after 2 ns, "01" after 4 ns, "00" after 10 ns;
-- délai inertiel de 2 ns; processus équivalent:
labeled: process begin
    data <= "11" after 2 ns, "01" after 4 ns, "00" after 10 ns;
    wait; -- le processus est stoppé indéfiniment
end process labeled;
```

16.3. Affectation concurrente conditionnelle de signal

L'affectation concurrente conditionnelle (*conditional signal assignment statement*) permet d'assigner différentes valeurs ou formes d'ondes en fonction d'une condition.

Syntaxe: Affectation concurrente conditionnelle de signal

```
[ étiquette : ] [ postponed ] nom-signal <= [ mode-délai ]
    { forme-onde when expression-boulienne else }
    forme-onde [ when expression-boulienne ] ;
```

Syntaxe: Processus équivalent de l'affectation concurrente conditionnelle de signal

```
[ étiquette : ] [ postponed ]
process ( signaux-des-formes-d'ondes-et-des-conditions )
begin
    if expression-boulienne then
        affectation-signal-simple ;
    elsif expression-boulienne then
        affectation-signal-simple ;
    else
        affectation-signal-simple ;
    end if;
end process [ étiquette ] ;
```

Exemple:

```
A <= B after 10 ns when Z = '1' else C after 15 ns;
-- processus équivalent:
process (B, C, Z)
begin
    if Z = '1' then
        A <= B after 10 ns;
```

```
else
    A <= C after 15 ns;
end if;
end process;
```

La partie **else** peut être omise ou utiliser le mot clé **unaffected**.

Exemples:

```
S <= '1' when cond;
S <= '1' when cond else unaffected;
```

16.4. Affectation concurrente sélective de signal

L'affectation concurrente de signal sélective (*selective signal assignment statement*) permet d'assigner différentes valeurs ou formes d'ondes en fonction de la valeur d'une expression de sélection.

Syntaxe: Affectation concurrente sélective de signal

```
[ étiquette : ] [ postponed ]
with expression select
    nom-signal <= [ mode-délai ]
    { forme-onde when choix { | choix } , }
    forme-onde when choix { | choix } ;
```

Syntaxe: Processus équivalent de l'affectation concurrente sélective de signal

```
[ étiquette : ] [ postponed ]
process ( signaux-de-l'expression-de-sélection )
begin
    case expression is
        when choix { | choix } => forme-onde ;
        { when choix { | choix } => forme-onde ; }
    end case;
end process [ étiquette ] ;
```

L'expression de sélection est de type discret ou d'un type tableau mono-dimensionnel. Les choix peuvent prendre l'une des formes suivantes:

- Des littéraux chaînes de caractères, des littéraux chaînes de bits ou des expressions constantes du même type.
- Des intervalles de valeurs discrètes.
- Le mot-clé **others** spécifie tous les choix possibles non spécifiés dans les choix précédents.

VHDL-AMS Essentiel

Exemple:

```
with muxval select
  S <= A after 5 ns when "00",
    B after 10 ns when "01" | "10",
    C after 15 ns when others;
-- processus équivalent:
process (A, B, C, muxval)
begin
  case muxval is
    when "00"      => S <= A after 5 ns;
    when "01" | "10" => S <= B after 10 ns;
    when others    => S <= C after 15 ns;
  end case;
end process;
```

La partie **when others** peut utiliser le mot clé **unaffected**.

Exemple:

```
with exp select
  S <= '1' when ADD,
  S <= '0' when SUB,
  unaffected when others;
```

16.5. Instance de composant (*component instance*)

Une instance de composant crée une copie d'un composant préalablement déclaré (§11) et définit les connexions de ce composant avec le reste du modèle.

Syntaxe: Instance de composant

```
nom-instance : [ component ] nom-composant
[ generic map ( liste-association-param-génériques ) ]
[ port map ( liste-association-ports ) ] ;
```

Exemple:

```
c_add: addn generic map (N => 8)
port map (A8, B8, sum => S8, cout => open);
```

L'**instanciation directe** (*direct instantiation*) ne requiert pas de déclaration préalable du composant instancié.

Syntaxe: Instanciation directe

```
nom-instance : entity nom-entité [ ( nom-architecture ) ]
[ generic map ( liste-association-param-génériques ) ]
[ port map ( liste-association-ports ) ] ;
```

Si le nom d'architecture n'est pas spécifié, l'architecture la plus récemment analysée (*most recently analyzed*, MRA) est utilisée.

Exemple:

```
c_add: entity WORK.addn(str)
generic map (N => 8)
port map (A8, B8, sum => S8, cout => open);
```

16.6. Génération d'instructions (*generate statements*)

La génération d'instructions concurrentes peut être faite de manière itérative ou conditionnelle. La génération a lieu à l'élaboration, avant le début de la simulation.

Syntaxe: Instruction generate

```
étiquette :
for identificateur in intervalle
| if expression_booléenne generate
  [ { déclaration-locale }
begin ]
  { instruction-concurrente | instruction-simultanée }
end generate [ étiquette ] ;
```

Les déclarations locales admises sont les mêmes que pour le corps d'architecture (§9.3). Elles sont dupliquées en fonction du type d'instruction.

Les instructions concurrentes **ou simultanées** sont dupliquées en fonction du type d'instruction.

L'instruction de génération itérative (**for**) génère autant d'instances des instructions spécifiées que de valeurs prises par l'identificateur. L'identificateur n'a pas besoin d'être déclaré.

L'instruction de génération conditionnelle (**if**) ne génère des instances des instructions spécifiées que si l'expression booléenne s'évalue à la valeur TRUE.

Exemples:

```
Gen: for i in 1 to N generate
  First: if i = 1 generate
    C1: comp port map (CLK, D => A, Q => S(i));
    S2(i) <= S(i) after 10 ns;
  end generate First;
```

```
Int: if i > 1 and i < N generate
  C1: comp port map (CLK, D => S(i-1), Q => S(i));
  S2(i) <= S(i-1) after 10 ns;
end generate Int;
Last: if i = N generate
  CN: comp port map (CLK, D => S(i-1), Q => B);
  S2(i) <= S(i-1) after 10 ns;
end generate Last;
end generate Gen;
```

16.7. Instruction concurrente d'assertion

L'instruction concurrente d'assertion (*concurrent assert statement*) a la même syntaxe que son équivalent séquentiel (§12.7).

Syntaxe: Processus équivalent de l'instruction concurrente d'assertion

```
[ étiquette : ] process
begin
  [ étiquette : ] assert condition
  [ report message ] [ severity niveau-sévérité ] ;
  wait [ on signaux-de-la-condition ] ;
end process;
```

Si la condition ne contient pas de référence à un signal, le processus équivalent est stoppé définitivement après l'exécution de l'instruction d'assertion.

16.8. Instruction concurrente break

Syntaxe: Instruction concurrente break

```
[ étiquette : ] break [ liste-éléments ] [ liste-sensibilité ]
[ when condition ] ;
```

Syntaxe: Processus équivalent de l'instruction concurrente break

```
[ étiquette : ] process
begin
  break [ liste-éléments ] [ when condition ] ;
  wait [ on liste-sensibilité ] ;
end process [ étiquette ] ;
```

Voir le §12.8 pour les détails des éléments de l'instruction.

VHDL-AMS Essentiel

La liste de sensibilité du processus équivalent inclut en plus tout nom de signal apparaissant dans la condition.

Exemple:

```
ground_dis: break v => -v when not s'above(0.0);
-- processus équivalent:
ground_dis: process
begin
    break v => -v when not s'above(0.0);
    wait on s'above(0.0);
end process ground_dis;
```

17. Paquetages standard

Seuls les paquetages STANDARD et TEXTIO font partie de la bibliothèque STD prédéfinie.

17.1. Paquetage STANDARD

Le paquetage STANDARD est implicitement visible dans n'importe quelle entité de conception (§9.1).

package STANDARD is

```
type BOOLEAN is (FALSE, TRUE);
-- logical operators that return BOOLEAN:
-- "and", "or", "nand", "nor", "xor", "xnor", "not"
-- relational operators that return BOOLEAN:
-- "=", "/=", "<", "<=", ">", ">="
```

```
type BIT is ('0', '1');
-- logical operators that return BIT:
-- "and", "or", "nand", "nor", "xor", "xnor", "not"
-- relational operators that return BOOLEAN:
-- "=", "/=", "<", "<=", ">", ">="
```

```
type CHARACTER is (
    NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
    BS, HT, LF, VT, FF, CR, SO, SI,
    DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
    CAN, EM, SUB, ESC, FSP, GSP, RSP, USP,
    ' ', '!', '"', '#', '$', '%', '&', "'",
    '(', ')', '*', '+', ',', '-', '.', '/',
    '0', '1', '2', '3', '4', '5', '6', '7',
    '8', '9', ':', ';', '<', '=', '>', '?',
    '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
    'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
    'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
    'X', 'Y', 'Z', '[', '\', ']', '^', '_',
    '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
    'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
    'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
    'x', 'y', 'z', '{', '|', '}', '~', DEL,
    C128, C129, C130, C131, C132, C133, C134, C135,
    C136, C137, C138, C139, C140, C141, C142, C143,
    C144, C145, C146, C147, C148, C149, C150, C151,
    C152, C153, C154, C155, C156, C157, C158, C159,
```

```
' ', '!', '¢', '£', '¤', '¥', '¦', '§',
'¨', '©', 'ª', «, '¬', '®', '¯',
'°', '±', '²', '³', '´', 'µ', '¶', '·',
'¸', '¹', 'º', '»', '¼', '½', '¾', '¿',
'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç',
'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î', 'Ï',
'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×',
'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß',
'à', 'á', 'â', 'ä', 'å', 'æ', 'ç',
'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï',
'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷',
'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ');
-- relational operators that return BOOLEAN:
-- "=", "/=", "<", "<=", ">", ">="
```

```
type SEVERITY_LEVEL is
    (NOTE, WARNING, ERROR, FAILURE);
-- relational operators that return BOOLEAN:
-- "=", "/=", "<", "<=", ">", ">="
```

```
type INTEGER is range dépend de l'implantation;
subtype NATURAL is INTEGER
    range 0 to INTEGER'HIGH;
subtype POSITIVE is INTEGER
    range 1 to INTEGER'HIGH;
-- relational operators that return BOOLEAN:
-- "=", "/=", "<", "<=", ">", ">="
-- arithmetic operators that return INTEGER:
-- "****", "**", "/", "+", "-", "abs", "rem", "mod"
```

```
type REAL is range -dépend de l'implantation;
-- relational operators that return BOOLEAN:
-- "=", "/=", "<", "<=", ">", ">="
-- arithmetic operators that return REAL:
-- "****", "**", "/", "+", "-", "abs"
```

```
type TIME is range dépend de l'implantation;
units
    fs;
    ps = 1000 fs;
    ns = 1000 ps;
    us = 1000 ns;
    ms = 1000 us;
    sec = 1000 ms;
    min = 60 sec;
```

VHDL-AMS Essentiel

```
hr = 60 min;
end units;
```

```
subtype DELAY_LENGTH is TIME range 0 to TIME'HIGH;
```

```
-- relational operators that return boolean:
-- "=", "/=", "<", "<=", ">", ">="
-- arithmetic operators that return time:
-- "*", "+", "-", "abs"
-- "/" returns time or integer
```

```
type DOMAIN_TYPE is (QUIESCENT_DOMAIN,
                     TIME_DOMAIN, FREQUENCY_DOMAIN);
```

```
-- relational operators that return boolean:
-- "=", "/=", "<", "<=", ">", ">="
```

```
signal DOMAIN: DOMAIN_TYPE
        := QUIESCENT_DOMAIN;
```

```
pure function NOW return DELAY_LENGTH;
```

```
pure function NOW return REAL;
```

```
function FREQUENCY return REAL;
```

```
type REAL_VECTOR is
```

```
    array (NATURAL range <>) of REAL;
```

```
type STRING is array (POSITIVE range <>)
                of CHARACTER;
```

```
-- relational operators that return BOOLEAN:
```

```
-- "=", "/=", "<", "<=", ">", ">="
```

```
-- concatenation operator returns STRING: "&"
```

```
type BIT_VECTOR is array (NATURAL range <>) of BIT;
```

```
-- logical operators that return BIT_VECTOR:
```

```
-- "and", "or", "nand", "nor", "xor", "xnor", "not"
```

```
-- "sll", "srl", "sla", "sra", "rol", "ror"
```

```
-- relational operators that return BOOLEAN:
```

```
-- "=", "/=", "<", "<=", ">", ">="
```

```
-- concatenation operator returns BIT_VECTOR: "&"
```

```
type FILE_OPEN_KIND is
```

```
    (READ_MODE, WRITE_MODE, APPEND_MODE);
```

```
type FILE_OPEN_STATUS is
```

```
    (OPEN_OK, STATUS_ERROR, NAME_ERROR,
     MODE_ERROR);
```

```
-- relational operators that return BOOLEAN:
```

```
-- "=", "/=", "<", "<=", ">", ">="
```

```
attribute FOREIGN: STRING;
end package STANDARD;
```

17.2. Paquetage TEXTIO

Le paquetage TEXTIO requiert l'usage de la clause de contexte suivante:

```
use STD.TEXTIO.all;
```

```
package TEXTIO is
```

```
-- type definitions for text I/O:
```

```
type LINE is access STRING;
```

```
type TEXT is file of STRING;
```

```
type SIDE is (RIGHT, LEFT);
```

```
subtype WIDTH is NATURAL;
```

```
-- standard text files:
```

```
file INPUT: TEXT is in "std_input";
```

```
file OUTPUT: TEXT is out "std_input";
```

```
file INPUT: TEXT open READ_MODE is "std_input";
```

```
file OUTPUT: TEXT open WRITE_MODE is "std_output";
```

```
-- input routines for standard types:
```

```
procedure READLINE (f: in TEXT; l: out LINE);
```

```
procedure READLINE (file f: TEXT; l: out LINE);
```

```
procedure READ (l: inout LINE; value: out BIT);
```

```
procedure READ (l: inout LINE; value: out BIT;
```

```
                good: out BOOLEAN);
```

```
procedure READ (l: inout LINE; value: out BIT_VECTOR);
```

```
procedure READ (l: inout LINE; value: out BIT_VECTOR;
```

```
                good: out BOOLEAN);
```

```
procedure READ (l: inout LINE; value: out BOOLEAN);
```

```
procedure READ (l: inout LINE; value: out BOOLEAN;
```

```
                good: out BOOLEAN);
```

```
procedure READ (l: inout LINE; value: out CHARACTER);
```

```
procedure READ (l: inout LINE; value: out CHARACTER;
```

```
                good: out BOOLEAN);
```

```
procedure READ (l: inout LINE; value: out INTEGER);
```

```
procedure READ (l: inout LINE; value: out INTEGER;
```

```
                good: out BOOLEAN);
```

```
procedure READ (l: inout LINE; value: out REAL);
```

```
procedure READ (l: inout LINE; value: out REAL;
```

```
                good: out BOOLEAN);
```

```
procedure READ (l: inout LINE; value: out STRING);
procedure READ (l: inout LINE; value: out STRING;
                good: out BOOLEAN);
```

```
procedure READ (l: inout LINE; value: out TIME);
```

```
procedure READ (l: inout LINE; value: out TIME;
                good: out BOOLEAN);
```

```
-- output routines for standard types:
```

```
procedure WRITELINE (f: out TEXT; l: in LINE);
```

```
procedure WRITELINE (file f: TEXT; l: in LINE);
```

```
procedure WRITE (l: inout LINE; value: in BIT;
                justified: in SIDE := RIGHT;
                field: in WIDTH := 0);
```

```
procedure WRITE (l: inout LINE; value: in bit_vector;
                justified: in SIDE := RIGHT;
                field: in WIDTH := 0);
```

```
procedure WRITE (l: inout LINE; value: in BOOLEAN;
                justified: in SIDE := RIGHT;
                field: in WIDTH := 0);
```

```
procedure WRITE (l: inout LINE; value: in CHARACTER;
                justified: in SIDE := RIGHT;
                field: in WIDTH := 0);
```

```
procedure WRITE (l: inout LINE; value: in INTEGER;
                justified: in SIDE := RIGHT;
                field: in WIDTH := 0);
```

```
procedure WRITE (l: inout LINE; value: in REAL;
                justified: in SIDE := RIGHT;
                field: in WIDTH := 0;
                digits: in NATURAL := 0);
```

```
procedure WRITE (l: inout LINE; value: in STRING;
                justified: in SIDE := RIGHT;
                field: in WIDTH := 0);
```

```
procedure WRITE (l: inout LINE; value: in TIME;
                justified: in SIDE := RIGHT;
                field: in WIDTH := 0;
```

```
                unit: in TIME := ns);
```

```
end package TEXTIO;
```

Note: VHDL-87 définit en plus la fonction ENDLINE:

```
function ENDLINE (l: LINE) return BOOLEAN;
```


VHDL-AMS Essentiel

VHDL-93 ne supporte plus cette fonction. Elle est remplacée par l'expression `L'LENGTH = 0`.

17.3. Paquetage STD_LOGIC_1164

Le paquetage STD_LOGIC_1164 est le standard IEEE 1164 définissant les types, les opérateurs et les fonctions nécessaires à une modélisation détaillée au niveau du transistor. Il est basé sur un système de valeurs logiques à 3 états (zéro, un et inconnu (*unknown*)) et 3 niveaux de forces (fort, faible et haute impédance). Un niveau fort représente l'effet d'une source active telle qu'une source de tension; un niveau faible représente l'effet d'une source résistive telle qu'une résistance pull-up ou un transistor de transmission; un niveau haute impédance représente l'effet d'une source désactivée. Le niveau fort domine le niveau faible qui domine lui-même le niveau haute impédance. Les 3 états sont complétés par l'état non initialisé (représentant le fait qu'une valeur n'a jamais été assignée à un signal) et l'état *dont-care* ou indéfini.

L'utilisation du paquetage nécessite les déclarations de contexte suivantes:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

package STD_LOGIC_1164 **is**

```
-- logic state system (unresolved)
type STD_ULOGIC is ('U', -- Uninitialized
                    'X', -- Forcing Unknown
                    '0', -- Forcing 0
                    '1', -- Forcing 1
                    'Z', -- High Impedance
                    'W', -- Weak Unknown
                    'L', -- Weak 0
                    'H', -- Weak 1
                    '-'); -- Don't care

-- unconstrained array of STD_ULOGIC for use with the
-- resolution function
type STD_ULOGIC_VECTOR is
    array (NATURAL range <>) of STD_ULOGIC;
-- resolution function
function RESOLVED (s : STD_ULOGIC_VECTOR)
    return STD_ULOGIC;
-- *** industry standard logic type ***
```

```
subtype STD_LOGIC is RESOLVED STD_ULOGIC;
-- unconstrained array of STD_LOGIC for use in declaring
-- signal arrays
type STD_LOGIC_VECTOR is
    array (NATURAL range <>) of STD_LOGIC;

-- common subtypes
subtype X01 is RESOLVED STD_ULOGIC
    range 'X' to '1'; -- ('X','0','1')
subtype X01Z is RESOLVED STD_ULOGIC
    range 'X' to 'Z'; -- ('X','0','1','Z')
subtype UX01 is RESOLVED STD_ULOGIC
    range 'U' to '1'; -- ('U','X','0','1')
subtype UX01Z is RESOLVED STD_ULOGIC
    range 'U' to 'Z'; -- ('U','X','0','1','Z')

-- overloaded logical operators
function "and" (l: STD_ULOGIC; r: STD_ULOGIC)
    return UX01;
function "nand" (l: STD_ULOGIC; r: STD_ULOGIC)
    return UX01;
function "or" (l: STD_ULOGIC; r: STD_ULOGIC)
    return UX01;
function "nor" (l: STD_ULOGIC; r: STD_ULOGIC)
    return UX01;
function "xor" (l: STD_ULOGIC; r: STD_ULOGIC)
    return UX01;
function "xnor" (l: STD_ULOGIC; r: STD_ULOGIC)
    return UX01;
function "not" (l: STD_ULOGIC) return UX01;

-- vectorized overloaded logical operators
function "and" (l,r: STD_LOGIC_VECTOR)
    return STD_LOGIC_VECTOR;
function "and" (l,r: STD_ULOGIC_VECTOR)
    return STD_ULOGIC_VECTOR;
function "nand" (l,r: STD_LOGIC_VECTOR)
    return STD_LOGIC_VECTOR;
function "nand" (l,r: STD_ULOGIC_VECTOR)
    return STD_ULOGIC_VECTOR;
function "or" (l,r: STD_LOGIC_VECTOR)
    return STD_LOGIC_VECTOR;
function "or" (l,r: STD_ULOGIC_VECTOR)
    return STD_ULOGIC_VECTOR;
function "nor" (l,r: STD_LOGIC_VECTOR)
```

```
    return STD_LOGIC_VECTOR;
(l,r: STD_ULOGIC_VECTOR)
return STD_ULOGIC_VECTOR;
function "xor" (l,r: STD_LOGIC_VECTOR)
    return STD_LOGIC_VECTOR;
(l,r: STD_ULOGIC_VECTOR)
return STD_ULOGIC_VECTOR;
function "xnor" (l,r: STD_LOGIC_VECTOR)
    return STD_ULOGIC_VECTOR;
(l,r: STD_ULOGIC_VECTOR)
return STD_ULOGIC_VECTOR;
function "not" (l: STD_LOGIC_VECTOR)
    return STD_LOGIC_VECTOR;
(l: STD_ULOGIC_VECTOR)
return STD_ULOGIC_VECTOR;

-- conversion functions
function TO_BIT (s: STD_ULOGIC; xmap: BIT := '0')
    return bit;
function TO_BITVECTOR (s: STD_LOGIC_VECTOR;
    xmap: bit := '0') return BIT_VECTOR;
function TO_BITVECTOR (s: STD_ULOGIC_VECTOR;
    xmap: bit := '0') return BIT_VECTOR;
function TO_STDULOGIC (b: BIT) return STD_ULOGIC;
function TO_STDLOGICVECTOR (b: BIT_VECTOR)
    return STD_LOGIC_VECTOR;
function TO_STDLOGICVECTOR
    (s: STD_ULOGIC_VECTOR)
    return STD_LOGIC_VECTOR;
function TO_STDLOGICVECTOR (b: BIT_VECTOR)
    return STD_ULOGIC_VECTOR;
function TO_STDULOGICVECTOR
    (s: STD_LOGIC_VECTOR)
    return STD_ULOGIC_VECTOR;

-- strength strippers and type converters
function TO_X01 (s: STD_LOGIC_VECTOR)
    return STD_LOGIC_VECTOR;
function TO_X01 (s: STD_ULOGIC_VECTOR)
    return STD_ULOGIC_VECTOR;
function TO_X01 (s: STD_ULOGIC) return X01;
function TO_X01 (b: BIT_VECTOR)
    return STD_LOGIC_VECTOR;
function TO_X01 (b: BIT_VECTOR)
    return STD_ULOGIC_VECTOR;
```

VHDL-AMS Essentiel

```

function TO_X01 (b: BIT) return X01;
function TO_X01Z (s: STD_LOGIC_VECTOR)
    return STD_LOGIC_VECTOR;
function TO_X01Z (s: STD_ULOGIC_VECTOR)
    return STD_ULOGIC_VECTOR;
function TO_X01Z (s: STD_ULOGIC) return X01Z;
function TO_X01Z (b: BIT_VECTOR)
    return STD_LOGIC_VECTOR;
function TO_X01Z (b: BIT_VECTOR)
    return STD_ULOGIC_VECTOR;
function TO_X01Z (b: BIT) return X01Z;
function TO_UX01 (s: STD_LOGIC_VECTOR)
    return STD_LOGIC_VECTOR;
function TO_UX01 (s: STD_ULOGIC_VECTOR)
    return STD_ULOGIC_VECTOR;
function TO_UX01 (s: STD_ULOGIC) return UX01;
function TO_UX01 (b: BIT_VECTOR)
    return STD_LOGIC_VECTOR;
function TO_UX01 (b: BIT_VECTOR)
    return STD_ULOGIC_VECTOR;
function TO_UX01 (b: BIT) return UX01;

-- edge detection
function RISING_EDGE (signal s: STD_ULOGIC)
    return BOOLEAN;
function FALLING_EDGE (signal s: STD_ULOGIC)
    return BOOLEAN;

-- object contains an unknown
function IS_X (s: STD_ULOGIC_VECTOR)
    return BOOLEAN;
function IS_X (s: STD_LOGIC_VECTOR)
    return BOOLEAN;
function IS_X (s: STD_ULOGIC) return BOOLEAN;

end package STD_LOGIC_1164;

```

Le standard définit une fonction de résolution RESOLVED qui utilise une table de résolution réalisant les règles suivantes:

- S'il existe une seule source, le signal résolu prends la valeur de cette source.
- Si le tableau de sources est vide, le signal résolu vaut 'Z'.
- Une valeur de source forte ('X', '0' ou '1') domine une valeur de source faible ('W', 'L' ou 'H').
- Deux sources de même forces mais de valeurs différentes

produisent une valeur résolue inconnue de même force ('X' ou 'W').

- La valeur haute impédance 'Z' est toujours dominée par des valeurs fortes et faibles.
- La résolution d'une valeur *don't care* '-' avec n'importe quelle autre valeur donne la valeur 'X'.
- La résolution d'une valeur 'U' avec n'importe quelle autre valeur donne la valeur 'U'. Ceci permet de détecter les signaux qui n'ont pas été initialisés correctement.

La fonction de résolution RESOLVED est définie de la manière suivante:

```

type stdlogic_table is
    array (STD_ULOGIC, STD_ULOGIC) of STD_ULOGIC;
constant resolution_table: stdlogic_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')); -- -

```

```

function RESOLVED (s : STD_ULOGIC_VECTOR)
    return STD_ULOGIC is
    variable result : STD_ULOGIC := 'Z'; -- state default
begin
    -- the test for a single driver is essential otherwise the
    -- loop would return 'X' for a single driver of 'L' and that
    -- would conflict with the value of a single driver unresolved
    -- signal.
    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;

```

17.4. Paquetage MATH_REAL

Le paquetage MATH_REAL est défini par le standard IEEE 1076.2. Il fournit les déclarations de constantes réelles et de routines mathématiques à valeurs réelles.

L'utilisation du paquetage nécessite les déclarations de contexte suivantes:

```

library IEEE;
use IEEE.MATH_REAL.all;

```

package MATH_real is

```

constant MATH_E: real -- Value of e
    := 2.71828_18284_59045_23536;
constant MATH_1_OVER_E: real -- Value of 1/e
    := 0.36787_94411_71442_32160;
constant MATH_PI: real -- Value of pi
    := 3.14159_26535_89793_23846;
constant MATH_2_PI: real -- Value of 2*pi
    := 6.28318_53071_79586_47693;
constant MATH_1_OVER_PI: real -- Value of 1/pi
    := 0.31830_98861_83790_67154;
constant MATH_PI_OVER_2: real -- Value of pi/2
    := 1.57079_63267_94896_61923;
constant MATH_PI_OVER_3: real -- Value of pi/3
    := 1.04719_75511_96597_74615;
constant MATH_PI_OVER_4: real -- Value of pi/4
    := 0.78539_81633_97448_30962;
constant MATH_3_PI_OVER_2: real -- Value of 3*pi/2
    := 4.71238_89803_84689_85769;
constant MATH_LOG_OF_2: real -- Natural log of 2
    := 0.69314_71805_59945_30942;
constant MATH_LOG_OF_10: real -- Natural log of 10
    := 2.30258_50929_94045_68402;
constant MATH_LOG2_OF_E: real -- Log base 2 of e
    := 1.44269_50408_88963_4074;
constant MATH_LOG10_OF_E: real -- Log base 10 of e
    := 0.43429_44819_03251_82765;
constant MATH_SQRT_2: real -- square root of 2
    := 1.41421_35623_73095_04880;
constant MATH_1_OVER_SQRT_2: real -- 1/SQRT(2)
    := 0.70710_67811_86547_52440;
constant MATH_SQRT_PI: real -- Square root of pi
    := 1.77245_38509_05516_02730;
constant MATH_DEG_TO_RAD: real

```

VHDL-AMS Essentiel

```

:= 0.01745_32925_19943_29577;
-- Conversion factor from degree to radian
constant MATH_RAD_TO_DEG: real
:= 57.29577_95130_82320_87680;
-- Conversion factor from radian to degree

```

```

function SIGN (X: in real) return real;
-- Returns 1.0 if X > 0.0; 0.0 if X = 0.0; -1.0 if X < 0.0
-- Special values: None
-- Domain: X in real
-- Error conditions: None
-- Range: ABS(SIGN(X)) <= 1.0
-- Notes: None

```

```

function CEIL (X: in real) return real;
-- Returns smallest INTEGER value (as real) not less than X
-- Special values: None
-- Domain: X in real
-- Error conditions: None
-- Range: CEIL(X) is mathematically unbounded
-- Notes: Implementations have to support at least the
-- domain ABS(X) < real(INTEGER'HIGH)

```

```

function FLOOR (X: in real) return real;
-- Returns largest integer value (as real) not greater than X
-- Special values: FLOOR(0.0) = 0.0
-- Domain: X in real
-- Error conditions: None
-- Range: FLOOR(X) is mathematically unbounded
-- Notes: Implementations have to support at least the
-- domain ABS(X) < real(INTEGER'HIGH)

```

```

function ROUND (X: in real) return real;
-- Rounds X to the nearest integer value (as real). If X is
-- halfway between two integers, rounding is away from 0.0
-- Special values: ROUND(0.0) = 0.0
-- Domain: X in real
-- Error conditions: None
-- Range: ROUND(X) is mathematically unbounded
-- Notes: Implementations have to support at least the
-- domain ABS(X) < real(INTEGER'HIGH)

```

```

function TRUNC (X: in real) return real;
-- Truncates X towards 0.0 and returns truncated value
-- Special values: TRUNC(0.0) = 0.0

```

```

-- Domain: X in real
-- Error conditions: None
-- Range: TRUNC(X) is mathematically unbounded
-- Notes: Implementations have to support at least the
-- domain ABS(X) < real(INTEGER'HIGH)

```

```

function "MOD" (X, Y: in real) return real;
-- Returns floating point modulus of X/Y, with the same sign
-- as Y, and absolute value less than the absolute value of
-- Y, and for some INTEGER value N the result satisfies th
-- relation  $X = Y \cdot N + \text{MOD}(X, Y)$ 
-- Special values: None
-- Domain: X in real; Y in real and  $Y \neq 0.0$ 
-- Error conditions: Error if  $Y = 0.0$ 
-- Range:  $\text{ABS}(\text{MOD}(X, Y)) < \text{ABS}(Y)$ 
-- Notes: None

```

```

function REALMAX (X, Y : in real) return real;
-- Returns the algebraically larger of X and Y
-- Special values: REALMAX(X,Y) = X when X = Y
-- Domain: X in real; Y in real
-- Error conditions: None
-- Range: REALMAX(X,Y) is mathematically unbounded
-- Notes: None

```

```

function REALMIN (X, Y : in real) return real;
-- Returns the algebraically smaller of X and Y
-- Special values: REALMIN(X,Y) = X when X = Y
-- Domain: X in real; Y in real
-- Error conditions: None
-- Range: REALMIN(X,Y) is mathematically unbounded
-- Notes: None

```

```

procedure UNIFORM (
  variable SEED1, SEED2: inout positive;
  variable X: out real);
-- Returns, in X, a pseudo-random number with uniform
-- distribution in the open interval (0.0, 1.0)
-- Special values: None
-- Domain:  $1 \leq \text{SEED1} \leq 2147483562$ ;
--  $1 \leq \text{SEED2} \leq 2147483398$ 
-- Error conditions:
-- Error if SEED1 or SEED2 outside of valid domain
-- Range:  $0.0 < X < 1.0$ 
-- Notes:

```

```

-- a) The semantics for this function are described by the
-- algorithm published by Pierre L'Ecuyer in
-- "Communications of the ACM," vol. 31, no. 6, June 1988,
-- pp. 742-774.
-- The algorithm is based on the combination of two
-- multiplicative linear congruential generators for 32-bit
-- platforms.
-- b) Before the first call to UNIFORM, the seed values
-- (SEED1, SEED2) have to be initialized to values in the
-- range [1, 2147483562] and [1, 2147483398] respectively.
-- The seed values are modified after each call to UNIFORM.
-- c) This random number generator is portable for 32-bit
-- computers, and it has a period of  $\sim 2.30584 \cdot (10^{**18})$  for
-- each set of seed values.
-- d) For information on spectral tests for the algorithm, refer
-- to the L'Ecuyer article.

```

```

function SQRT (X: in real) return real;
-- Returns square root of X
-- Special values: SQRT(0.0) = 0.0, SQRT(1.0) = 1.0
-- Domain:  $X \geq 0.0$ 
-- Error conditions: Error if  $X < 0.0$ 
-- Range:  $\text{SQRT}(X) \geq 0.0$ 
-- Notes: The upper bound of the reachable range of SQRT
-- is approximately given by  $\text{SQRT}(X) \leq \text{SQRT}(\text{real'HIGH})$ 

```

```

function CBRT (X: in real) return real;
-- Returns cube root of X
-- Special values:
--  $\text{CBRT}(0.0) = 0.0$ ,  $\text{CBRT}(1.0) = 1.0$ ,  $\text{CBRT}(-1.0) = -1.0$ 
-- Domain: X in real
-- Error conditions: None
-- Range: CBRT(X) is mathematically unbounded
-- Notes: The reachable range of CBRT is approximately
-- given by:  $\text{ABS}(\text{CBRT}(X)) \leq \text{CBRT}(\text{real'HIGH})$ 

```

```

function "^^" (X: in integer; Y: in real) return real;
-- Returns Y power of X ==>  $X^{**Y}$ 
-- Special values:
--  $X^{**0.0} = 1.0$ ;  $X \neq 0$ 
--  $0^{**Y} = 0.0$ ;  $Y > 0.0$ 
--  $X^{**1.0} = \text{real}(X)$ ;  $X \geq 0$ 
--  $1^{**Y} = 1.0$ 
-- Domain:
--  $X > 0$ 

```

VHDL-AMS Essentiel

```
-- X = 0 for Y > 0.0
-- X < 0 for Y = 0.0
-- Error conditions:
--   Error if X < 0 and Y /= 0
--   Error if X = 0 and Y <= 0.0
-- Range: X**Y >= 0.0
-- Notes: The upper bound of the reachable range for "" is
--   approximately given by: X**Y <= real'HIGH
```

```
function "" (X: in real; Y: in real) return real;
-- Returns Y power of X ==> X**Y
-- Special values:
--   X**0.0 = 1.0; X /= 0
--   0**Y = 0.0; Y > 0.0
--   X**1.0 = real(X); X >= 0
--   1.0**Y = 1.0
-- Domain:
--   X > 0.0
--   X = 0.0 for Y > 0.0
--   X < 0.0 for Y = 0.0
-- Error conditions:
--   Error if X < 0.0 and Y /= 0
--   Error if X = 0.0 and Y <= 0.0
-- Range: X**Y >= 0.0
-- Notes: The upper bound of the reachable range for "" is
--   approximately given by: X**Y <= real'HIGH
```

```
function EXP (X: in real) return real;
-- Returns e**X; where e = MATH_E
-- Special values:
--   EXP(0.0) = 1.0
--   EXP(1.0) = MATH_E
--   EXP(-1.0) = MATH_1_OVER_E
--   EXP(X) = 0.0 for X <= -LOG(real'HIGH)
-- Domain: X in real such that EXP(X) <= real'HIGH
-- Error conditions: Error if X > LOG(real'HIGH)
-- Range: EXP(X) >= 0.0
-- Notes: The usable domain of EXP is approximately given
-- by: X <= LOG(real'HIGH)
```

```
function LOG (X: in real) return real;
-- Returns natural logarithm of X
-- Special values: LOG(1.0) = 0.0, LOG(MATH_E) = 1.0
-- Domain: X > 0.0
-- Error conditions: Error if X <= 0.0
```

```
-- Range: LOG(X) is mathematically unbounded
-- Notes: The reachable range of LOG is approximately
-- given by: LOG(0+) <= LOG(X) <= LOG(real'HIGH)
```

```
function LOG2 (X: in real) return real;
-- Returns logarithm base 2 of X
-- Special values: LOG2(1.0) = 0.0, LOG2(2.0) = 1.0
-- Domain: X > 0
-- Error conditions: Error if X <= 0.0
-- Range: LOG2(X) is mathematically unbounded
-- Notes: The reachable range of LOG2 is approximately
-- given by: LOG2(0+) <= LOG2(X) <= LOG2(real'HIGH)
```

```
function LOG10 (X: in real) return real;
-- Returns logarithm base 10 of X
-- Special values: LOG10(1.0) = 0.0, LOG10(10.0) = 1.0
-- Domain: X > 0.0
-- Error conditions: Error if X <= 0.0
-- Range: LOG10(X) is mathematically unbounded
-- Notes: The reachable range of LOG10 is approximately
-- given by: LOG10(0+) <= LOG10(X) <= LOG10(real'HIGH)
```

```
function LOG (X: in real; BASE: in real) return real;
-- Returns logarithm base BASE of X
-- Special values:
--   LOG(1.0, BASE) = 0.0, LOG(BASE, BASE) = 1.0
-- Domain: X > 0.0, BASE > 0.0, BASE /= 1.0
-- Error conditions:
--   Error if X <= 0.0
--   Error if BASE <= 0.0 or if BASE = 1.0
-- Range: LOG(X, BASE) is mathematically unbounded
-- Notes:
-- a) When BASE > 1.0, the reachable range of LOG is
--   approximately given by:
--   LOG(0+, BASE) <= LOG(X, BASE) <= LOG(real'HIGH,
--   BASE)
-- b) When 0.0 < BASE < 1.0, the reachable range of LOG is
--   approximately given by:
--   LOG(real'HIGH, BASE) <= LOG(X, BASE) <= LOG(0+,
--   BASE)
```

```
function SIN (X: in real) return real;
-- Returns sine of X; X in radians
-- Special values:
```

```
-- SIN(X) = 0.0 for X = k*MATH_PI
-- SIN(X) = 1.0 for X = (4*k+1)*MATH_PI_OVER_2
-- SIN(X) = -1.0 for X = (4*k+3)*MATH_PI_OVER_2
-- where k is an integer
-- Domain: X in real
-- Error conditions: None
-- Range: ABS(SIN(X)) <= 1.0
-- Notes: For larger values of ABS(X), degraded accuracy is
-- allowed
```

```
function COS (X: in real) return real;
-- Returns cosine of X; X in radians
-- Special values:
--   COS(X) = 0.0 for X = (2*k+1)*MATH_PI_OVER_2
--   COS(X) = 1.0 for X = (2*k)*MATH_PI
--   COS(X) = -1.0 for X = (2*k+1)*MATH_PI
-- where k is an INTEGER
-- Domain: X in real
-- Error conditions: None
-- Range: ABS(COS(X)) <= 1.0
-- Notes: For larger values of ABS(X), degraded accuracy is
-- allowed
```

```
function TAN (X: in real) return real;
-- Returns tangent of X; X in radians
-- Special values:
--   TAN(X) = 0.0 for X = k*MATH_PI, where k is an integer
-- Domain: X in real and
--   X /= (2*k+1)*MATH_PI_OVER_2,
-- where k is an INTEGER
-- Error conditions:
--   Error if X = ((2*k+1) * MATH_PI_OVER_2)
-- where k is an integer
-- Range: TAN(X) is mathematically unbounded
-- Notes: For larger values of ABS(X), degraded accuracy is
-- allowed
```

```
function ARCSIN (X: in real) return real;
-- Returns inverse sine of X
-- Special values:
--   ARCSIN(0.0) = 0.0
--   ARCSIN(1.0) = MATH_PI_OVER_2
--   ARCSIN(-1.0) = -MATH_PI_OVER_2
-- Domain: ABS(X) <= 1.0
-- Error conditions: Error if ABS(X) > 1.0
```


VHDL-AMS Essentiel

```
-- Range: ABS(ARCSIN(X)) <= MATH_PI_OVER_2
-- Notes: None
```

```
function ARCCOS (X: in real) return real;
-- Returns inverse cosine of X
-- Special values:
--   ARCCOS(1.0) = 0.0
--   ARCCOS(0.0) = MATH_PI_OVER_2
--   ARCCOS(-1.0) = MATH_PI
-- Domain: ABS(X) <= 1.0
-- Error conditions: Error if ABS(X) > 1.0
-- Range: 0.0 <= ARCCOS(X) <= MATH_PI
-- Notes: None
```

```
function ARCTAN (Y : in real) return real;
-- Returns the value of the angle in radians of the point
-- (1.0, Y), which is in rectangular coordinates
-- Special values: ARCTAN(0.0) = 0.0
-- Domain: Y in real
-- Error conditions: None
-- Range: ABS(ARCTAN(Y)) <= MATH_PI_OVER_2
-- Notes: None
```

```
function ARCTAN (Y : in real; X: in real) return real;
-- Returns the principal value of the angle in radians of
-- the point (X, Y), which is in rectangular coordinates
-- Special values:
--   ARCTAN(0.0, X) = 0.0 if X > 0.0
--   ARCTAN(0.0, X) = MATH_PI if X < 0.0
--   ARCTAN(Y, 0.0) = MATH_PI_OVER_2 if Y > 0.0
--   ARCTAN(Y, 0.0) = -MATH_PI_OVER_2 if Y < 0.0
-- Domain:
--   Y in real, X in real, X /= 0.0 when Y = 0.0
-- Error conditions: Error if X = 0.0 and Y = 0.0
-- Range: -MATH_PI < ARCTAN(Y,X) <= MATH_PI
-- Notes: None
```

```
function SINH (X: in real) return real;
-- Returns hyperbolic sine of X
-- Special values: SINH(0.0) = 0.0
-- Domain: X in real
-- Error conditions: None
-- Range: SINH(X) is mathematically unbounded
-- Notes: The usable domain of SINH is approximately given
-- by:   ABS(X) <= LOG(real'HIGH)
```

```
function COSH (X: in real) return real;
-- Returns hyperbolic cosine of X
-- Special values: COSH(0.0) = 1.0
-- Domain: X in real
-- Error conditions: None
-- Range: COSH(X) >= 1.0
-- Notes: The usable domain of COSH is approximately
-- given by:   ABS(X) <= LOG(real'HIGH)
```

```
function TANH (X: in real) return real;
-- Returns hyperbolic tangent of X
-- Special values: TANH(0.0) = 0.0
-- Domain: X in real
-- Error conditions: None
-- Range: ABS(TANH(X)) <= 1.0
-- Notes: None
```

```
function ARCSINH (X: in real) return real;
-- Returns inverse hyperbolic sine of X
-- Special values: ARCSINH(0.0) = 0.0
-- Domain: X in real
-- Error conditions: None
-- Range: ARCSINH(X) is mathematically unbounded
-- Notes: The reachable range of ARCSINH is approximately
-- given by: ABS(ARCSINH(X)) <= LOG(real'HIGH)
```

```
function ARCCOSH (X: in real) return real;
-- Returns inverse hyperbolic cosine of X
-- Special values: ARCCOSH(1.0) = 0.0
-- Domain: X >= 1.
-- Error conditions: Error if X < 1.0
-- Range: ARCCOSH(X) >= 0.0
-- Notes: The upper bound of the reachable range of
-- ARCCOSH is approximately given by:
--   ARCCOSH(X) <= LOG(real'HIGH)
```

```
function ARCTANH (X: in real) return real;
-- Returns inverse hyperbolic tangent of X
-- Special values: ARCTANH(0.0) = 0.0
-- Domain: ABS(X) < 1.0
-- Error conditions: Error if ABS(X) >= 1.0
-- Range: ARCTANH(X) is mathematically unbounded
-- Notes: The reachable range of ARCTANH is
-- approximately given by:
```

```
--   ABS(ARCTANH(X)) < LOG(real'HIGH)
end MATH_REAL;
```

17.5. Paquetage ENERGY_SYSTEMS

Le paquetage ENERGY_SYSTEMS contient des déclarations pour la modélisation de systèmes à conservation d'énergie (systèmes conservatifs).

Il ne s'agit pas encore d'un paquetage standard IEEE et le contenu donné ici est une proposition pour le futur standard valable pour les outils de Mentor Graphics (AdvanceMS).

Il faut déclarer les clauses de contexte suivantes:

```
library IEEE_PROPOSED;
use IEEE_PROPOSED.ENERGY_SYSTEMS.all;
```

-- Version 2.1

library IEEE;

use IEEE.MATH_REAL.all;

package ENERGY_SYSTEMS is

-- subtype declarations

subtype ENERGY is REAL
tolerance "DEFAULT_ENERGY";

subtype POWER is REAL
tolerance "DEFAULT_POWER";

subtype PERIODICITY is REAL
tolerance "DEFAULT_PERIODICITY";

-- common scaling factors

constant ATTO : REAL := 1.0e-18;

constant FEMTO : REAL := 1.0e-15;

constant PICO : REAL := 1.0e-12;

constant NANO : REAL := 1.0e-9;

constant MICRO : REAL := 1.0e-6;

constant MILLI : REAL := 1.0e-3;

constant CENTI : REAL := 1.0e-2;

constant DENCI : REAL := 1.0e-1;

constant DEKA : REAL := 1.0e+1;

constant HECTO : REAL := 1.0e+2;

constant KILO : REAL := 1.0e+3;

constant MEGA : REAL := 1.0e+6;

constant GIGA : REAL := 1.0e+9;

constant TERA : REAL := 1.0e+12;

-- permittivity of vacuum <FARADS/METER>

constant EPS0 : REAL := 8.8542e-12;

-- permeability of vacuum <HENRIES/METER>

constant MU0 : REAL := 4.0e-6 * MATH_PI;

-- electron charge <COULOMB>

constant Q : REAL := 1.60218e-19;

-- Boltzmann constant <JOULES/KELVIN>

constant K : REAL := 1.38066e-23;

-- Relative permittivity of silicon

constant EPS_SI : REAL := 11.8;

-- Relative permittivity of silicon dioxide

constant EPS_SIO2 : REAL := 3.9;

-- Young's Modulus for silicon <PASCALS>

constant E_SI : REAL := 190.0e+9;

-- Young's Modulus for silicon dioxide <PASCALS>

constant E_SIO2 : REAL := 73.0e+9;

-- Poisson's Ratio for silicon <100> orientation

constant NU_SI : REAL := 0.28;

-- Acceleration due to gravity <METERS/SECOND_SQRD>

constant GRAV : REAL := 9.81;

-- attribute declarations

constant SYMBOL : STRING

end package ENERGY_SYSTEMS;

17.6. Paquetage ELECTRICAL_SYSTEMS

Le paquetage ELECTRICAL_SYSTEMS contient des déclarations pour la modélisation de systèmes électriques et magnétiques.

Il ne s'agit pas encore d'un paquetage standard IEEE et le contenu donné ici est une proposition pour le futur standard valable pour les outils de Mentor Graphics (AdvanceMS).

Il faut déclarer les clauses de contexte suivantes:

```
library IEEE_PROPOSED;
use IEEE_PROPOSED.ELECTRICAL_SYSTEMS.all;
```

-- Version 2.1

library IEEE;

use IEEE.ENERGY_SYSTEMS.all;

package ELECTRICAL_SYSTEMS is

-- subtype declarations

subtype VOLTAGE is REAL
tolerance "DEFAULT_VOLTAGE";

subtype CURRENT is REAL
tolerance "DEFAULT_CURRENT";

subtype CHARGE is REAL
tolerance "DEFAULT_CHARGE";

subtype RESISTANCE is REAL
tolerance "DEFAULT_RESISTANCE";

subtype CAPACITANCE is REAL
tolerance "DEFAULT_CAPACITANCE";

subtype MMF is REAL **tolerance** "DEFAULT_MMF";

subtype FLUX is REAL **tolerance** "DEFAULT_FLUX";

subtype INDUCTANCE is REAL
tolerance "DEFAULT_INDUCTANCE";

-- nature declarations

nature ELECTRICAL is
VOLTAGE across
CURRENT through
ELECTRICAL_REF reference;

nature ELECTRICAL_VECTOR is
array (NATURAL range <>) **of** ELECTRICAL;

nature MAGNETIC is
MMF across
FLUX through
MAGNETIC_REF reference;

nature MAGNETIC_VECTOR is
array (NATURAL range <>) **of** MAGNETIC;

-- attribute declarations

attribute SYMBOL **of** VOLTAGE: **subtype** is "Volt";

attribute SYMBOL **of** CURRENT: **subtype** is "Ampere";

attribute SYMBOL **of** CHARGE: **subtype** is "Coulomb";

attribute SYMBOL **of** RESISTANCE: **subtype** is "Ohm";

attribute SYMBOL **of** CAPACITANCE: **subtype** is "Farad";

attribute SYMBOL **of** MM: **subtype** is "Ampere-Turns";

attribute SYMBOL **of** FLUX: **subtype** is "Weber";

attribute SYMBOL **of** INDUCTANCE: **subtype** is "Henry";

end package ELECTRICAL_SYSTEMS;

17.7. Paquetage MECHANICAL_SYSTEMS

Le paquetage MECHANICAL_SYSTEMS contient des déclarations pour la modélisation de systèmes mécaniques linéaires et rotationnels.

Il ne s'agit pas encore d'un paquetage standard IEEE et le contenu donné ici est une proposition pour le futur standard valable pour les outils de Mentor Graphics (AdvanceMS).

Il faut déclarer les clauses de contexte suivantes:

```
library IEEE_PROPOSED;
use IEEE_PROPOSED.MECHANICAL_SYSTEMS.all;
```

-- Version 2.1

library IEEE;

use IEEE.ENERGY_SYSTEMS.all;

package MECHANICAL_SYSTEMS is

-- subtype declarations

subtype DISPLACEMENT is REAL

VHDL-AMS Essentiel

```

    tolerance "DEFAULT_DISPLACEMENT";
    subtype FORCE is REAL tolerance "DEFAULT_FORCE";
    subtype VELOCITY is REAL
        tolerance "DEFAULT_VELOCITY";
    subtype ACCELERATION is REAL
        tolerance "DEFAULT_ACCELERATION";
    subtype MASS is REAL tolerance "DEFAULT_MASS";
    subtype STIFFNESS is REAL
        tolerance "DEFAULT_STIFFNESS";
    subtype DAMPING is REAL
        tolerance "DEFAULT_DAMPING";
    subtype ANGLE is REAL tolerance "DEFAULT_ANGLE";
    subtype TORQUE is REAL
        tolerance "DEFAULT_TORQUE";
    subtype ANGULAR_VELOCITY is REAL
        tolerance "DEFAULT_ANGULAR_VELOCITY";
    subtype ANGULAR_ACCEL is REAL
        tolerance "DEFAULT_ANGULAR_ACCEL";
    subtype MMOMENT_I is REAL
        tolerance "DEFAULT_MMOMENT_I";
    -- nature declarations
    nature TRANSLATIONAL is
        DISPLACEMENT across
        FORCE through
        TRANSLATIONAL_REF reference;
    nature TRANSLATIONAL_VECTOR is
        array (NATURAL range <=>) of TRANSLATIONAL;
    nature TRANSLATIONAL_V is
        VELOCITY across
        FORCE through
        TRANSLATIONAL_V_REF reference;
    nature TRANSLATIONALV_VECTOR is
        array (NATURAL range <=>) of TRANSLATIONAL_V;
    nature ROTATIONAL is
        ANGLE across
        TORQUE through
        ROTATIONAL_REF reference;
    nature ROTATIONAL_VECTOR is
        array (NATURAL range <=>) of ROTATIONAL;
    nature ROTATIONAL_V is
        ANGULAR_VELOCITY across
        TORQUE through
        ROTATIONAL_V_REF reference;
    nature ROTATIONALV_VECTOR is
        array (NATURAL range <=>) of ROTATIONAL_V;

```

```

    -- attribute declarations
    attribute SYMBOL of DISPLACEMENT: subtype is
        "Meter";
    attribute SYMBOL of FORCE: subtype is "Newton";
    attribute SYMBOL of VELOCITY: subtype is
        "Meters/Second";
    attribute SYMBOL of ACCELERATION: subtype is
        "Meters/Second^2";
    attribute SYMBOL of MASS: subtype is "Kilogram";
    attribute SYMBOL of STIFFNESS: subtype is
        "Newton/Meter";
    attribute SYMBOL of DAMPING: subtype is
        "Newton Seconds/Meter";
    attribute SYMBOL of ANGLE: subtype is "Radian";
    attribute SYMBOL of TORQUE: subtype is
        "Newton Meters";
    attribute SYMBOL of ANGULAR_VELOCITY: subtype is
        "Radians/Second";
    attribute SYMBOL of ANGULAR_ACCEL: subtype is
        "Radians/Second^2";
    attribute SYMBOL of MMOMENT_I: subtype is
        "Kilogram Meters^2";
    end package MECHANICAL_SYSTEMS;

```

17.8. Paquetage FLUIDIC_SYSTEMS

Le paquetage FLUIDIC_SYSTEMS contient des déclarations pour la modélisation de systèmes fluidiques.

Il ne s'agit pas encore d'un paquetage standard IEEE et le contenu donné ici est une proposition pour le futur standard valable pour les outils de Mentor Graphics (AdvanceMS).

Il faut déclarer les clauses de contexte suivantes:

```

    library IEEE_PROPOSED;
    use IEEE_PROPOSED.FLUIDIC_SYSTEMS.all;

```

```

    -- Version 2.1
    library IEEE;
    use IEEE.ENERGY_SYSTEMS.all;
    package FLUIDIC_SYSTEMS is
    -- subtype declarations
    subtype PRESSURE is REAL
        tolerance "DEFAULT_PRESSURE";
    subtype VFLOW_RATE is REAL
        tolerance "DEFAULT_VFLOW_RATE";
    subtype VOLUME is REAL

```

```

    tolerance "DEFAULT_VOLUME";
    subtype DENSITY is REAL
        tolerance "DEFAULT_DENSITY";
    subtype VISCOSITY is REAL
        tolerance "DEFAULT_VISCOSITY";
    subtype FRESISTANCE is REAL
        tolerance "DEFAULT_FRESISTANCE";
    subtype FCAPACITANCE is REAL
        tolerance "DEFAULT_FCAPACITANCE";
    subtype INERTANCE is REAL
        tolerance "DEFAULT_INERTANCE";
    -- nature declarations
    nature FLUIDIC is
        PRESSURE across
        VFLOW_RATE through
        FLUIDIC_REF reference;
    nature FLUIDIC_VECTOR is
        array (NATURAL range <=>) of FLUIDIC;
    -- attribute declarations
    attribute SYMBOL of PRESSURE: subtype is "Pascal";
    attribute SYMBOL of VFLOW_RATE: subtype is
        "Meters^3/Second";
    attribute SYMBOL of DENSITY: subtype is
        "Kilogram/Meter^3";
    attribute SYMBOL of VISCOSITY: subtype is
        "Newton Second/Meter^2";
    attribute SYMBOL of VOLUME: subtype is "Meter^3";
    attribute SYMBOL of FRESISTANCE: subtype is
        "Newton Second/Meter^5";
    attribute SYMBOL of FCAPACITANCE: subtype is
        "Meter^5/Newton";
    attribute SYMBOL of INERTANCE: subtype is
        "Newton Second^2/Meter^5";
    end package FLUIDIC_SYSTEMS;

```

17.9. Paquetage THERMAL_SYSTEMS

Le paquetage THERMAL_SYSTEMS contient des déclarations pour la modélisation de systèmes thermiques.

Il ne s'agit pas encore d'un paquetage standard IEEE et le contenu donné ici est une proposition pour le futur standard valable pour les outils de Mentor Graphics (AdvanceMS).

Il faut déclarer les clauses de contexte suivantes:

```
library IEEE_PROPOSED;
use IEEE_PROPOSED.THERMAL_SYSTEMS.all;
```

-- Version 2.1

```
library IEEE;
use IEEE.ENERGY_SYSTEMS.all;
package THERMAL_SYSTEMS is
-- subtype declarations
subtype TEMPERATURE is REAL
tolerance "DEFAULT_TEMPERATURE";
subtype HEAT_FLOW is REAL
tolerance "DEFAULT_HEAT_FLOW";
-- nature declarations
nature THERMAL is
TEMPERATURE across
HEAT_FLOW through
THERMAL_REF reference;
nature THERMAL_VECTOR is
array (NATURAL range <>) of THERMAL;
-- attribute declarations
attribute SYMBOL of TEMPERATURE: subtype is
"Kelvin";
attribute SYMBOL of HEAT_FLOW: subtype is
"Joules/Second";
end package THERMAL_SYSTEMS;
```

17.10. Paquetage RADIANT_SYSTEMS

Le paquetage RADIANT_SYSTEMS contient des déclarations pour la modélisation de systèmes rayonnants.

Il ne s'agit pas encore d'un paquetage standard IEEE et le contenu donné ici est une proposition pour le futur standard valable pour les outils de Mentor Graphics (AdvanceMS).

Il faut déclarer les clauses de contexte suivantes:

```
library IEEE_PROPOSED;
use IEEE_PROPOSED.RADIANT_SYSTEMS.all;
```

-- Version 2.1

```
library IEEE;
use IEEE.ENERGY_SYSTEMS.all;
package RADIANT_SYSTEMS is
-- subtype declarations
subtype ILLUMINANCE is REAL
tolerance "DEFAULT_ILLUMINANCE";
subtype OPTIC_FLUX is REAL
tolerance "DEFAULT_OPTIC_FLUX";
-- nature declarations
nature RADIANT is
ILLUMINANCE across
OPTIC_FLUX through
RADIANT_REF reference;
nature RADIANT_VECTOR is
array (NATURAL range <>) of RADIANT;
-- attribute declarations
attribute SYMBOL of ILLUMINANCE: subtype is
"Candela";
attribute SYMBOL of OPTIC_FLUX: subtype is "Lumen";
end package RADIANT_SYSTEMS;
```

VHDL-AMS Essentiel

17.11. Paquetages NUMERIC_BIT et NUMERIC_STD

Le paquetage NUMERIC_BIT (NUMERIC_STD) est défini par le standard IEEE 1076.3. Il définit les types numériques et les fonctions arithmétiques à utiliser avec les outils de synthèse.

L'utilisation du paquetage nécessite les déclarations de contexte suivantes:

library IEEE;

use IEEE.NUMERIC_BIT.all;

library IEEE;

use IEEE.STD_LOGIC_1164.all;

use IEEE.NUMERIC_STD.all;

Types définis:

type UNSIGNED is array (NATURAL range <>) of BIT;

type SIGNED is array (NATURAL range <>) of BIT;

type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;

type SIGNED is array (NATURAL range <>) of STD_LOGIC;

Sous-programmes définis (types des arguments: *U*= UNSIGNED, *S*=SIGNED, *I*=INTEGER, *N*=NATURAL, *B*=BIT, *S(U)*L=STD_(U)LOGIC, *S(U)*LV=STD_(U)LOGIC_VECTOR):

Type	Name	Argument(s)(Type(s))	Result subtype	Result
Arithmetic operators				
F	"abs"	ARG (S)	signed(ARG'length-1 downto 0)	abs value of vector ARG
	"-"	ARG (S)	signed(ARG'length-1 downto 0)	unary minus operation on vector ARG
	"+"	L (U), R (U)	unsigned(max(L'length, R'length)-1 downto 0)	adds two vectors that may be of different lengths
		L (S), R (S)	signed(max(L'length, R'length)-1 downto 0)	
		L (U), R (N)	unsigned(L'length-1 downto 0)	adds a vector (integer) L, with an integer (vector) R
		L (N), R (U)	unsigned(R'length-1 downto 0)	
		L (I), R (S)	signed(R'length-1 downto 0)	
		L (S), R (I)	signed(L'length-1 downto 0)	
	"-"	L (U), R (U)	unsigned(max(L'length, R'length)-1 downto 0)	subtracts two vectors that may be of different lengths
		L (S), R (S)	signed(max(L'length, R'length)-1 downto 0)	subtracts a vector (integer) L, from an integer (vector) R
		L (U), R (N)	unsigned(L'length-1 downto 0)	
		L (N), R (U)	unsigned(R'length-1 downto 0)	
		L (I), R (S)	signed(R'length-1 downto 0)	
		L (S), R (I)	signed(L'length-1 downto 0)	
	"*"	L (U), R (U)	unsigned((L'length+R'length-1) downto 0)	multiplies two vectors that may possibly be of different lengths
		L (S), R (S)	signed((L'length+R'length-1) downto 0)	
		L (U), R (N)	unsigned((L'length+L'length-1) downto 0)	multiplies a vector (integer) L with an integer (vector) R; R (L) is converted to a vector before multiplication
		L (N), R (U)	unsigned((R'length+R'length-1) downto 0)	
		L (I), R (S)	signed((R'length+R'length-1) downto 0)	
		L (S), R (I)	signed((L'length+L'length-1) downto 0)	

VHDL-AMS Essentiel

Type	Name	Argument(s)(Type(s))	Result subtype	Result
F	"/" ^a	L (U), R (U)	unsigned(L'length-1 downto 0)	divides two vectors
		L (S), R (S)	signed(L'length-1 downto 0)	
		L (U), R (N)	unsigned(L'length-1 downto 0)	divides vector (integer) L by an integer (vector) R; result is truncated to vector'length
		L (N), R(U)	unsigned(R'length-1 downto 0)	
		L (S), R (I)	signed(L'length-1 downto 0)	
		L (I), R (S)	signed(R'length-1 downto 0)	
	"rem" ^a	L (U), R (U)	unsigned(R'length-1 downto 0)	computes "L rem R"
		L (S), R (S)	signed(R'length-1 downto 0)	
		L (U), R (N)	unsigned(L'length-1 downto 0)	computes "L rem R"; result is truncated to vector'length
		L (N), R(U)	unsigned(R'length-1 downto 0)	
		L (S), R (I)	signed(L'length-1 downto 0)	
		L (I), R (S)	signed(R'length-1 downto 0)	
	"mod" ^a	L (U), R (U)	unsigned(R'length-1 downto 0)	computes "L mod R"
		L (S), R (S)	signed(R'length-1 downto 0)	
		L (U), R (N)	unsigned(L'length-1 downto 0)	computes "L mod R"; result is truncated to vector'length
		L (N), R(U)	unsigned(R'length-1 downto 0)	
		L (S), R (I)	signed(L'length-1 downto 0)	
		L (I), R (S)	signed(R'length-1 downto 0)	
Comparison operators				
F	">" "<" "<=" ">=" "=" "/="	L (U), R (U)	boolean	computes "L r R", where r is a relational operator; vectors may possibly be of different lengths ^b
		L (S), R (S)		
		L (U), R (N)		computes "L r R", where r is a relational operator ^b
		L (N), R(U)		
		L (S), R (I)		
		L (I), R (S)		
		Shift and rotate operators		
F	shift_left	ARG (U), COUNT (N)	unsigned(ARG'length-1 downto 0)	shifts left vector ARG COUNT times; vacated positions are filled with '0'; COUNT leftmost bits are lost
		ARG (S), COUNT (N)	signed(ARG'length-1 downto 0)	
	shift_right	ARG (U), COUNT (N)	unsigned(ARG'length-1 downto 0)	shifts right vector ARG COUNT times; vacated positions are filled with '0'; COUNT rightmost bits are lost
		ARG (S), COUNT (N)	signed(ARG'length-1 downto 0)	

VHDL-AMS Essentiel

Type	Name	Argument(s)(Type(s))	Result subtype	Result
F	rotate_left	ARG (U), COUNT (N)	unsigned(ARG'length-1 downto 0)	rotates left vector ARG COUNT times
		ARG (S), COUNT (N)	signed(ARG'length-1 downto 0)	
	rotate_right	ARG (U), COUNT (N)	unsigned(ARG'length-1 downto 0)	rotates right vector ARG COUNT times
		ARG (S), COUNT (N)	signed(ARG'length-1 downto 0)	
	"sl" ^C	ARG (U), COUNT (N)	unsigned(ARG'length-1 downto 0)	shift_left(ARG, COUNT)
		ARG (S), COUNT (N)	signed(ARG'length-1 downto 0)	
	"slr" ^C	ARG (U), COUNT (N)	unsigned(ARG'length-1 downto 0)	shift_right(ARG, COUNT)
		ARG (S), COUNT (N)	signed(ARG'length-1 downto 0)	signed(shift_right(unsigned(ARG), COUNT))
	"rol" ^C	ARG (U), COUNT (N)	unsigned(ARG'length-1 downto 0)	rotate_left(ARG, COUNT)
		ARG (S), COUNT (N)	signed(ARG'length-1 downto 0)	
	"ror" ^C	ARG (U), COUNT (N)	unsigned(ARG'length-1 downto 0)	rotate_right(ARG, COUNT)
		ARG (S), COUNT (N)	signed(ARG'length-1 downto 0)	
Resize functions				
F	resize	ARG (U), NEW_SIZE (N)	unsigned(NEW_SIZE-1 downto 0)	resizes vector ARG to NEW_SIZE; when enlarging, new (leftmost) bit positions are filled with '0'; when truncating, leftmost bits are lost
		ARG (S), NEW_SIZE (N)	signed(NEW_SIZE-1 downto 0)	resizes vector ARG to NEW_SIZE; when enlarging, new (leftmost) bit positions are filled with sign bit ARG'left; when truncating, sign bit is retained with the rightmost part
Conversion functions				
F	to_integer	ARG (U)	natural	unsigned to non negative integer
		ARG (S)	integer	signed to integer
	to_unsigned	ARG (N), SIZE (N)	unsigned(SIZE-1 downto 0)	non negative to unsigned of specified size
	to_signed	ARG (I), SIZE (N)	signed(SIZE-1 downto 0)	integer to signed of specified size
Logical operators				
F	"not"	L (U)	unsigned(L'length-1 downto 0)	bitwise inversion
		L (S)	signed(L'length-1 downto 0)	
	"and"	L (U), R (U)	unsigned(L'length-1 downto 0)	bitwise AND
		L (S), R (S)	signed(L'length-1 downto 0)	
	"or"	L (U), R (U)	unsigned(L'length-1 downto 0)	bitwise OR
		L (S), R (S)	signed(L'length-1 downto 0)	

VHDL-AMS Essentiel

Type	Name	Argument(s)(Type(s))	Result subtype	Result
F	"nand"	L (U), R (U)	unsigned(L'length-1 downto 0)	bitwise NAND
		L (S), R (S)	signed(L'length-1 downto 0)	
	"nor"	L (U), R (U)	unsigned(L'length-1 downto 0)	bitwise NOR
		L (S), R (S)	signed(L'length-1 downto 0)	
	"xor"	L (U), R (U)	unsigned(L'length-1 downto 0)	bitwise XOR
		L (S), R (S)	signed(L'length-1 downto 0)	
	"xnor"	L (U), R (U)	unsigned(L'length-1 downto 0)	bitwise XNOR
		L (S), R (S)	signed(L'length-1 downto 0)	
Edge detection functions				
F	rising_edge ^d	S (B)	boolean	TRUE if event on signal S and new value is '1'
	falling_edge ^d			TRUE if event on signal S and new value is '0'
Match functions ^e				
F	std_match	L (SUL), R (SUL)	boolean	bitwise compare; treats value '-' (don't care) as matching any other STD_ULOGIC value
		L (U), R (U)		
		L (S), R (S)		
		L (SULV), R (SULV)		
		L (SLV), R (SLV)		
Translation functions ^e				
F	to_01	S (U), XMAP (SL)	unsigned(S'range)	bitwise translation; '1' 'H' -> '1' '0' 'L' -> '0' others -> XMAP (= '0' by default)
		S (S), XMAP (SL)	signed(S'range)	

- a. If 2nd argument is zero, a severity level of ERROR is issued.
- b. Operands are treated as binary integers.
- c. Predefined in VHDL for ARG of type bit_vector.
- d. Defined in STD_LOGIC_1164 package for S of type std_ulogic.
- e. Only in package NUMERIC_STD.