Syntaxe VHDL

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

ABS, AFTER, AND, ARCHITECTURE, ASSERT, ATTRIBUTS.

Abs

Voir les opérateurs.

After

AFTER spécifie un temps de réponse d'un signal par rapport à son événement.
Deux modes de délai sont possibles INERTIAL (défaut) ou TRANSPORT. Le mode INERTIAL ne propage pas les impulsions plus courtes que le temps de propagation.

Syntaxe

```
sigal_name <= [INERTIAL/TRANSPORT] expression AFTER
time_expression
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY nor2 IS
    PORT (a,b : IN std_logic;
    z : OUT std_logic);
END nor2;

ARCHITECTURE behavior OF nor2 IS
BEGIN
    dly0: PROCESS(a,b)
    BEGIN
        z <= a NOR b AFTER 10 ns; -- temps de propagation est de
        10ns
    END PROCESS dly0;
END behavior;
```

And

Voir les opérateurs.

Architecture

Déclare une architecture comportementale ou structurelle d'une entité. Ceci permet de définir les relations entre les entrées/sorties d'une fonction.

Syntaxe

```
ARCHITECTURE architecture_name OF entity_name IS
    [subprogram_declaration] -- zone de déclaration
    [subprogram_body]
    [type_declarations]
    [constant_declarations]
    [signal_declarations]
    .
    .
    .
BEGIN
    [block_statement]
    [process_statement]
    [concurrent_signal_assignement_statement]
    [generate_statement]
    .
    .
END [architecture_name];
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY fulladd IS
    PORT (x,y,cin : IN bit;
    sum, cout : OUT bit);
END fulladd;
```

```
ARCHITECTURE behavior OF fulladd IS
BEGIN
    sum <= x XOR y XOR cin;
    cout <= (x AND y) OR (x AND cin) OR (y AND cin);
END behavior;
```

Assert

ASSERT est une instruction de déboggage.
Test si une condition est TRUE. Dans le cas contraire, reporte un message sur l'écran du simulateur.
ASSERT est très utile pour vérifier les violations de timing.
L'option de SEVERITE permet de définir le niveau d'importance de l'alerte. Voici la liste des niveaux :
NOTE, WARNING, ERROR, FAILURE. Le dernier niveau arrête normalement la simulation.

Guide des niveaux de sévérité :
- Note : utilisé pour information seulement
  - "Note : Chargement de données d'un fichier"
- Warning : utilisé pour fournir une information sur une erreur en instance
  - "Warning : Détection d'un pic"
- Error : utilisé pour information seulement
  - "Error : Violation du temps d'initialisation"
- Failure : raporte une grosse erreur
  - "Failure : Ligne RESET instable"

Syntaxe

ASSERT condition
[REPORT string] [SEVERITY severity_level];

Exemple

```
check_setup: PROCESS (clk, d)
BEGIN
    IF (clk'EVENT AND clk='1') THEN -- test si front montant de
    clk
        ASSERT d'STABLE(setup_time) -- regarde si d est stable
        pendant setup_time
            REPORT "Setup Violation..." -- affiche un message
            d'avertissement si pas de stabilité
            SEVERITY WARNING;
    END IF;
END PROCESS check_setup;
```

Attributs

Les attributs permettent d'ajouter des informations supplémentaires à un signal, variable ou un composant.

Syntaxe

```
object_name'attribut_name
```

Syntaxe des attributs pour le type ARRAY ou scalaire

```
X'HIGH élément le plus grand ou borne maximale
X'LOW élément le plus petit ou borne minimale
X'LEFT élément de gauche ou borne de gauche
X'RIGHT élément de droite ou borne de droite
```

Syntaxe des attributs pour les objets déclarés dans un ARRAY ou un SUBTYPE

```
x'RANGE élément range de x
x'REVERSE_RANGE élément range de x dans l'ordre inverse
x'LENGTH x'HIGH - x'LOW + 1 (integer)
```

Syntaxe des attributs pour SIGNAL

```
x'EVENT retourne TRUE si x change d'état
x'ACTIVE retourne TRUE si x a changé durant le dernier interval
x'LAST_EVENT retourne une valeur temporelle depuis le dernier
changement de x
x'LAST_ACTIVE retourne une valeur temporelle depuis la dernière
transition de x
x'LAST_VALUE retourne la dernière valeur de x
```

Ces attributs créent un nouveau SIGNAL

```
x'DELAYED(t) crée un signal du type de x retardé par t
x'STABLE(t) retourne TRUE si x n'est pas modifié pendant le temps
t
x'QUIET(t) crée un signal logique à TRUE si x n'est pas modifié
pendant un temps t
x'TRANSACTION crée un signal logique qui bascule lorsque x est
change d'état
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY d_flop IS
    PORT (d,clk,clr, set : IN std_logic;
                q : OUT std_logic);
END d_flop;

ARCHITECTURE behavior OF d_flop IS -- fonctionnement d'une bascule
D
CONSTANT clrit: std_logic := '0';
CONSTANT setit: std_logic := '1';
BEGIN
    PROCESS (clk, clr, set) -- liste de sensibilité
    BEGIN
        IF (clk='1') AND (clk'EVENT) THEN -- TRUE sur un front
        montant de clk
            IF (clr = '0') THEN -- si clr est à 0
                q <= clrit; -- clr actif, q à 0
```

```
            ELSIF (set = '0') THEN -- si set est à 0
                q <= setit; -- set actif, q à 1
            ELSE
                q <= d; -- si ni clr ni set, alors q = d
            END IF;
        END IF;
    END PROCESS;
END behavior;
```


B

C

CASE, CASSE, COMMENTAIRE, COMPONENT, CONFIGURATION, CONSTANT.

Case

On utilise CASE pour permettre le choix entre plusieurs actions. Cette instruction est très utile dans les machines d'états.
En fin de liste, on peut ajouter WHEN OTHERS qui permet de donner une action à tous les choix qui n'ont pu être trouvés dans la liste.

Syntaxe

```
CASE expression IS
    WHEN choices => sequence_of_statements;
    {WHEN choices => sequence_of_statements;}
END CASE;
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY handshk IS
    PORT (clk,req : IN std_logic;
```

```
            ack : OUT std_logic);
END handshk;

ARCHITECTURE behavior OF handshk IS
CONSTANT st0 : std_logic_vector(0 TO 1) := "00";
CONSTANT st1 : std_logic_vector(0 TO 1) := "01";
CONSTANT st2 : std_logic_vector(0 TO 1) := "10";
CONSTANT st3 : std_logic_vector(0 TO 1) := "11";
SIGNAL state : std_logic_vector(0 TO 1) := "00";
BEGIN
    checkreq: PROCESS
    BEGIN
        WAIT UNTIL clk = '1'; -- attend un front montant de clk
        CASE state IS -- state est la variable du choix
            WHEN st0 => -- si state = st0 soit '00'
                IF req = '1' THEN -- si req = '1'
                    state <= st1; -- on incrémente la machine
                    d'état
                END IF;
            WHEN st1 => -- si state = st1
                state <= st2; -- la machine d'état passe à st2
            WHEN st2 =>
                state <= st3;
            WHEN st3 =>
                state <= st0; -- la machine d'état est bouclée
                vers st0
            WHEN OTHERS => NULL;
        END CASE;
    END PROCESS checkreq;

    ackit: PROCESS (state) -- ce PROCESS gère un bit de
    propagation
    BEGIN
        CASE state IS
            WHEN st2 | st3 => -- si state vaut st2 ou st3
                ack <= '1'; -- passer ack à 1
            WHEN OTHERS => -- dans tous les autres cas de state
                ack <= '0'; -- laisser ack à 0
        END CASE;
    END PROCESS ackit;
END behavior;
```

Casse

La casse n'est pas sensible. Il n'y a pas de différenciation minuscule/majuscule.

Commentaire

Un commentaire débute par deux traits (signe moins) : --

Exemple

```
sum <= x XOR y; -- OU EXCLUSIF de x et y.
```

Component

COMPONENT permet d'écrire d'un programme sous la forme structurelle. En effet, il faut déclarer et définir les composants de la fonction puis dans le corps de l'architecture, il suffit de les connecter en suivant un schéma structurel.
Cette méthode d'écriture permet aussi la description hiérarchique.
Quatre blocs sont à distinguer dans une telle définition :

- rescencement de tous les composants du schéma. Ceci se fait par la syntaxe COMPONENT ... END COMPONENT ;
- affectation des liaisons du schéma aux broches des composants. On utilise PORT MAP. Cette phase correspond à réaliser la NETLIST du schéma ;
- attribution des circuits à leur ENTITY. Cette phase permet de contrôler le modèle comportemental qui va déterminer du fonctionnement du composant. Cette phase est réalisée avec CONFIGURATION ;
- définition comportementale de chaque composant associé. Ici on procédera à une simple structure d'écriture basée autour d'ENTITY et d'ARCHITECTURE.

Syntaxe
Déclaration

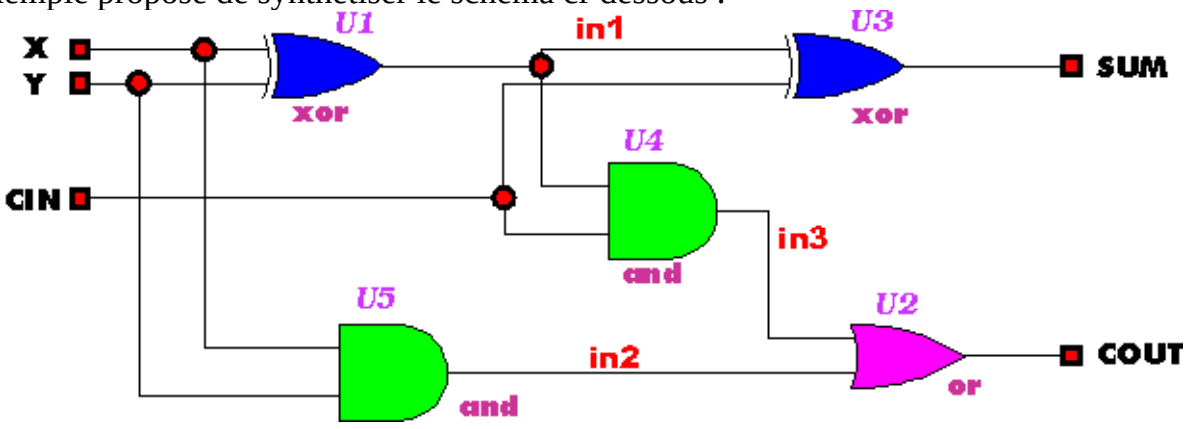
```
COMPONENT COMPONENT_name
    [local_generic_clause]
    [local_port_clause]
END COMPONENT;
```

Définition du brochage

```
instantiation_label: COMPONENT_name
[GENERIC MAP (local_generic_list)]
[PORT MAP local_port_list]
```

Exemple

L'exemple propose de synthétiser le schéma ci-dessous :




```
ENTITY fulladd IS
    PORT (x,y,cin : IN bit; sum,cout : OUT bit);
END fulladd;


ARCHITECTURE struct OF fulladd IS -- 1° rescencement de tous les
composants du schéma
```

```
    COMPONENT xor2 -- déclaration du OU-EXCLUSIF
        PORT (a, b : IN bit;
                q: OUT bit);
    END COMPONENT;

    COMPONENT or2 -- déclaration du OU
        PORT (a, b : IN bit;
                q: OUT bit);
    END COMPONENT;

    COMPONENT and2 -- déclaration du ET
        PORT (a,b : IN bit;
                q: OUT bit);
    END COMPONENT;
    SIGNAL in1, in2, in3: bit;

    BEGIN -- 2° affectation des liaisons du schéma aux broches des
    composants
    u1: xor2
        PORT MAP(x, -- affectation par nom
                y, -- on retrouve les signaux du schéma et les
                signaux du composant
                in1);

    u2: or2
        PORT MAP(in2, -- affectation par position
                in3, -- c'est l'ordre de passage qui détermine
                des connections
                cout);

    u3: xor2
        PORT MAP(a => in1,b => cin,q=>sum);

    u4: and2
        PORT MAP(a => in1,b => cin,q=>in3);

    u5: and2 -- il faut passer tous les composants en revue
        PORT MAP(a => x,b => y, q=>in2);
END struct;

CONFIGURATION fulladd_config OF fulladd IS -- 3° attribution des
circuits à leur ENTITY
FOR struct
    FOR u1,u3: xor2 USE ENTITY WORK.xor2; -- u1 et u3 héritent du
    comportement de xor2_arch défini plus loin
    END FOR;

    FOR u2: or2 USE ENTITY WORK.or2; -- affectation de u2
    END FOR;

    FOR u4,u5: and2 USE ENTITY WORK.and2;-- affectation de u4 et
    u5
    END FOR;

END FOR;
END fulladd_config;
```

```
-------------------------------- contenu d'une autre bibliothèque

ENTITY and2 IS
    PORT (a,b : IN bit;q : OUT bit);
END and2;

ARCHITECTURE behv OF and2 IS
BEGIN
    q<=a and b;
END behv;


ENTITY or2 IS
    PORT (a,b : IN bit;q : OUT bit);
END or2;

ARCHITECTURE behv OF or2 IS
BEGIN
    q<=a or b;
END behv;


ENTITY xor2 IS
    PORT (a,b : IN bit;q : OUT bit);
END xor2;

ARCHITECTURE behv OF xor2 IS
BEGIN
    q<=((not a) and b) or (a and (not b));
END behv;
```

Configuration

CONFIGURATION s'utilise en combinaison avec COMPONENT.

Syntaxe

```
CONFIGURATION configuration_name OF entity_name IS
        FOR architecture_name
            {use_clause}
            {component_configuration}
        END FOR;
END [configuration_name];
```

Constantes

Les constantes sont utilisées pour référencer une valeur ou un type spécifique.
Elles permettent une meilleur lecture et maintenance d'un source.
De tous types, elles sont utilisées dans les entités, architectures ou packets.

Syntaxe

```
CONSTANT const_name {, const_name} : type := value;
```

Exemple

```
constant BUS_WIDTH : integer := 8;
```

D

DELAY.

Delay

Voir AFTER.

E

ENTITY, EXIT.

Entity

Déclare une entité visible de l'extérieur.

Syntaxe

```
ENTITY entity_name IS
    [generic_declarations]
    [port_declarations]
    .
    .
    .
END [entity_name];
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

```
ENTITY fulladd IS
    PORT (x,y,cin : IN bit;
    sum, cout : OUT bit);
END ulladd;
```

Exit

EXIT se combine avec FOR, WHILE et LOOP pour sortir d'une boucle.

Syntaxe

```
EXIT [loop_label] [WHEN condition];
```

F

FOR, FUNCTION.

For

FOR est une instruction de bouclage. Elle s'utilise avec LOOP.
On place la condition de fin de sortie dans l'instruction même contrairement au LOOP seul où on est obligé d'avoir recourt à la commande EXIT. On peut toutefois utiliser EXIT ou NEXT.
Le bouclage se fait autant de fois que discrete_range l'indique.

Syntaxe

```
[loop_label:] FOR index_variable IN discrete_range LOOP
        sequence_of_statements
END LOOP [loop_label];
```

Exemple

```
USE ieee.std_logic_1164.all;
ENTITY clock IS
    PORT (clk1 : INOUT std_logic := '0');
END clock;
ARCHITECTURE behv OF clock IS
CONSTANT pulse_time : time := 25ns;
BEGIN
    clkit: PROCESS
    VARIABLE a : integer := 1;
```

```
        BEGIN
            loop1: FOR a IN 1 TO 10 LOOP -- la variable de boucle est
            a de 1 à 10
                WAIT FOR pulse_time; -- attend la valeur de
                pulse_time
                clk1 <= NOT clk1; -- complémente clk1
            END LOOP loop1;
        END PROCESS clkit;
    END behv;
```

Function

Les FONCTIONS sont des blocs d'instructions qui retournent une valeur. Les fonctions peuvent être appelées de différents endroits.
Les paramètres passés sont du type IN et les objets de class SIGNAL ou CONSTANT.
On peut déclarer des variables locales à l'intérieur d'une fonction. Ces variables perdent leur valeur à chaque sortie de la fonction et sont initialisées à chaque appel.
Une fonction peut être déclarée dans un PACKAGE ou dans le corps d'une ARCHITECTURE.
Une fonction autorise qu'un traitement séquentiel (pas de PROCESS et WAIT) ni d'assignation d'un signal.
Une fonction doit comporter au moins une instruction RETURN.
Voir aussi PROCEDURE et PACKAGE.

Syntaxe
Déclaration

```
    FUNCTION function_name
        {(parameter_list)}
        RETURN type_name;
```

Corps de la fonction

```
    FUNCTION function_name
        {(parameter_list)}
        RETURN type_name IS
    BEGIN
        [block_statement]
        [generate_statement]
    END [function_name];
```

Exemple

```
    LIBRARY ieee;
    USE ieee.std_logic_1164.all;
    ENTITY nor2 IS
        GENERIC (tphl : time := 10.0ns; -- déclaration des temps de
        propagation
                tplh : time := 15.0ns);
        PORT (a,b : IN std_logic;
                z : OUT std_logic);
    END nor2;

    ARCHITECTURE behavior OF nor2 IS
```

```
BEGIN
    dly: PROCESS(a,b)
    VARIABLE newstate : std_logic;
    BEGIN
        newstate := a NOR b;
        z <= newstate AFTER delay(a NOR b,tplh,tphl); -- appel de
        la fonction delay
    END PROCESS dly;
END behavior;

LIBRARY ieee; -- déclaration de la fonction
USE ieee.std_logic_1164.all;
USE work.myfuncs.all; -- on déclare la bibliothèque utilisateur
myfuncs
PACKAGE myfuncs IS
    FUNCTION delay(newval : IN std_logic; -- entête de delay
        delay01 : IN time; -- les paramètres n'ont pas le même
        nom, c'est l'ordre qui compte
    delay10 : IN time) RETURN time; -- retourne un nombre de type
    time
END myfuncs;

PACKAGE BODY myfuncs IS
    FUNCTION delay(newval : IN std_logic; -- corps de delay
        delay01 : IN time;
    delay10 : IN time) RETURN time;
BEGIN
    CASE newval IS
        WHEN '0' => RETURN delay01; -- retourne la valeur delay01
        WHEN '1' => RETURN delay10; -- retourne la valeur delay10
        WHEN OTHERS => IF (delay01 > delay10) THEN -- retourne la
        plus grande valeur
            RETURN delay01;
        ELSE;
            RETURN delay10;
        END IF;
    END CASE;
END delay;
END myfuncs;
```

G

GENERIC.

Generic

La section GENERIC_DECLARATIONS dans l'entête ENTITY permet de définir des paramètres exploitables dans l'architecture.
Cette méthode d'écriture permet une maintenance plus aisée.

Syntaxe

```
ENTITY entity_name IS
    [generic_declarations]
    [port_declarations]
END [entity_name];
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY nor2 IS
    GENERIC (tphlmin, tplhmin: time := 3.0ns; -- déclaration des
    temps de propagation
        tphltyp : time := 8.0ns;
        tplhtyp : time := 6.5ns;
        tphlmax, tplhmax: time := 13.0ns);
    PORT (a,b : IN std_logic;
        z : OUT std_logic);
END nor2;

ARCHITECTURE behavior OF nor2 IS
BEGIN
    dly0: PROCESS(a,b);
    BEGIN
        IF a = '0' AND b = '0' THEN -- il faut résoudre tous les
        cas puisque tplhtyp < > tphltyp
            z <= '1' AFTER tplhmin;
        ELSIF a = '0' AND b = '1' THEN
            z <= '0' AFTER tphlmin;
        ELSIF a = '1' AND b = '0' THEN
            z <= '0' AFTER tphlmin;
        ELSIF a = '1' AND b = '1' THEN
            z <= '0' AFTER tphlmin;
        END IF;
    END PROCESS dly0;
END behavior;
```

H

I

IF...THEN, INERTIAL.

IF ... THEN

La combinaison IF THEN permet d'effectuer un test d'une expression logique. La suite du déroulement dépend du résultat.
Deux imbrications sont possibles : ELSIF suivi d'un autre test et ELSE qui contrôle les résultats FALSE.

Syntaxe

```
IF condition THEN
    sequence_of_statements
{ELSIF condition THEN
    sequence_of_statements}
[ELSE
    sequence_of_statements]
END IF;
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY mux IS
    PORT (a,b,sel : IN std_logic;
            y : OUT std_logic);
END mux;
ARCHITECTURE behavior OF mux IS
BEGIN
    PROCESS (sel,a,b) -- liste de sensibilité
    BEGIN
        IF sel = '0' THEN -- test si sel vaut 0
            y <= a; -- si a=0 reproduit a sur y
        ELSIF sel = '1' THEN -- test si sel vaut 1
            y <= b; -- si a=1 reproduit b sur y
        ELSE
            y <= 'X'; -- si sel n'est ni à 1 ni à 0, on affecte X
            à y
        END IF;
    END PROCESS;
END behavior;
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY por IS
    PORT (int0,int1 : IN std_logic := '1';
            p1_1,p1_2 : OUT std_logic := '0'
            reset : INOUT std_logic := '0');
END por;
ARCHITECTURE behv OF por IS
BEGIN
    pwrit: PROCESS
    VARIABLE init : boolean := true; -- déclare une variable init
    initialisée à TRUE
    BEGIN
        IF init = true THEN -- teste init
            WAIT FOR 150ns -- si init est TRUE, temporise 150ns
            reset <= '1'; -- positionne reset à 1
            init := false; -- après le premier passage, la
            variable init est passée à FALSE
        ELSE -- si init n'est pas TRUE
            WAIT FOR 150ns; -- temporise 150ns
            reset <= '0'; -- met reset à 0. On suppose
            l'initialisation terminée
        END IF;
    END PROCESS pwrit;

    ckint: PROCESS -- ce PROCESS est déclanché suivant la liste
    définie par WAIT ON
    BEGIN
        IF reset = '1' THEN -- si reset est à 1
            IF int0 = '1' THEN -- si int0 est à 0
                p1_1 <= '1'; -- place p1_1 à 1
            ELSIF int1 = '1' THEN -- si int1 est à 1
                p1_2 <= '1'; -- place p1_2 à 1
            END IF;
        END IF;
        WAIT ON int0, int1, reset; -- attend un changement d'état
        d'un de ces signaux
    END PROCESS ckint;
END behv;
```

Inertial

Voir AFTER.

J

K

L

LIBRARY, LOOP.

Library

Déclare une liste de bibliothèques. Cette instruction s'utilise avec USE.
Ll'instruction LIBRARY doit apparaître au début du fichier source.
Les bibliothèques WORK et STD n'ont pas besoin d'être déclarées. Chaque développement incorpore implicitement les lignes suivantes :

```
LIBRARY WORK;
LIBRARY STD;
LIBRARY STD.STANDARD.all;
```

Syntaxe

```
LIBRARY library_name {, library_name};
USE library_name.package_name.declarative_unit;
```

Exemple

```
LIBRARY mylib;
USE mylib.math.all;
```

Loop

LOOP est une instruction de bouclage. Elle peut s'utiliser seule ou avec FOR ou WHILE.
Ici, elle est utilisée seule pour répéter une série d'instructions. Dans ce cas (lorsqu'elle est utilisée seule), la condition de fin de boucle est précisée par l'instruction EXIT ou NEXT por

passer à l'itération sauivante.

Syntaxe

```
[loop_label:] LOOP
        sequence_of_statements
END LOOP [loop_label];
```

Exemple

```
PROCESS (a) -- positionne z(a) à 1
    VARIABLE i : INTEGER RANGE 0 TO 4; -- déclare une variable
    entière variant de 0 à 4
BEGIN
    z <= "0000"; -- positione z à 0000
    i := 0; -- initialise i
    LOOP -- marque le début de la boucle
        EXIT WHEN i = 4; -- condition pour sortir de la boucle
        IF (a = i) THEN -- teste si a = i
            z(i) <= '1'; -- met z indice i à 1
        END IF;
        i := i + 1; -- incrémente i
    END LOOP; -- fin de la boucle
END PROCESS;
```

M

MOD.

Mod

Voir les opérateurs.

N

NAND, NEXT, NOR, NOT, NULL.

Nand

Voir les opérateurs.

Next

NEXT se combine avec FOR, WHILE et LOOP pour passer à l'itération suivante.

Syntaxe

```
NEXT [loop_label] [WHEN condition];
```

Nor

Voir les opérateurs.

Not

Voir les opérateurs.

Null

Null est l'instruction qui n'effectue pas d'action.

Syntaxe

```
NULL;
```

O

OPERATEURS, OR.

Opérateurs

Le tableau suivant résume les opérateurs VHDL.

Classe	Symbole	Fonction	Definit pour

Opérateurs divers *Classe de plus haute priorité*	not ** abs	complément exponentiel valeur absolue	bit, booléen entier, réel numérique
Opérateurs multiplicatifs	* / mod rem	multiplication division modulo reste	numérique entier
Signe (unaire)	+ -	positif négatif	numérique
Opérateurs additifs (binaire)	+ - &	addition soustraction concaténation	numérique 1 dimension
Opérateurs relationnels	= /= < > <= >=	égal différent inférieur supérieur inférieur ou égal supérieur ou égal	tous les types retournent un booléen Scalaire retourne un booléen
Opérateurs logiques (binaire) *Classe de plus faible priorité*	and or nand nor xor	et logique ou logique et non logique ou non logique ou exclusif	bit booléen vecteur

Les opérateurs sont définis pour des classes particulières.

La classe des opérateurs Divers

Ces opérateurs permettent la complémentarité, la fonction exponentielle et la recherche de la valeur absolue d'un nombre.
Ils retournent une valeur de même type.

La classe des opérateurs Logiques

Les opérateurs logiques doivent éffectuer une opération sur les types bit, booléen, bit_vector, tableau linéaire de booléens, std_logic et std_logic_vector. Ils retournent une valeur de même type.

Exemple

```
SIGNAL a,b,c : bit_vector(4 DOWNTO 0);
SIGNAL d,e : bit_vector(2 DOWNTO 0);
SIGNAL out1,out2 : bit_vector(4 DOWNTO 0);

out1 <= a AND b;
out1 <= (a AND b) OR c; -- l'utilisation des parenthèses permet de
certifier l'opération exacte

c <= b AND d; -- interdit car les types sont différents
e <= d OR 2; -- interdit car les types sont différents
```

La classe des opérateurs Relationels

Les opérateurs égalité et inégalité sont définis pour tous les types. Ils retournent un booléen.

Les autres opérateurs relationels sont définis pour tous les types scalaires et pour toutes les dimensions. Ils retournent également un booléen.

Exemple

```
'0' = '0'; -- évalué à TRUE
"101" < "110"; -- évalué à TRUE

constant ARR1 :bit_vector := "0011";
constant ARR2 :bit_vector := "01";
-- (ARR1 < ARR2) will return true car la comparaison s'effectue
bit à bit de gauche à droite.
```

La classe des opérateurs Additifs

Les additions et soustractions sont permises pour les opérandes numériques. Les opérandes doivent être de même type.
La concaténation est autorisée pour les tableaux à une dimension.

Exemple

```
CONSTANT s1 : string(1 TO 4):="ABCD";
CONSTANT s2 : string(1 TO 4):="EFGH";
CONSTANT s3 : string(1 TO 8):=s1 & s2; -- "ABCDEFGH"
```

Exemple

```
ENTITY concat IS
    PORT (a,b,c : IN std_logic;
             d : OUT std_logic_vector( 2 DOWNTO 0));
END concat;
ARCHITECTURE behav OF concat IS
BEGIN
    d <= NOT(a) & NOT(b) & NOT(c); -- d est un vecteur formé des
    compléments de a, b et c
END behav;
```

La classe Signe

Cette petite classe permet de spécifier le signe d'un opérateur.

Exemple

```
a/-b; -- illégal
a/(-b); -- légal
```

La classe des opérateurs Multiplicatifs

Cette classe permet des opérandes de types différents. Les exemples suivants illustrent cette

particularité.

Exemple

```
9 ns * 8 = 72ns; -- le résultat est du type physique
9 ns / 3 = 3; -- le résultat est du type numérique
9 ns * 8 ns -> l'opération provoque une erreur de compilation
```

Or

Voir les opérateurs.

P

PACKAGE, PORT, PORT MAP, PROCEDURE, PROCESS.

Package

PACKAGE déclare une fonction ou une procédure.
Le nom défini par PACKAGE BODY doit être le même que celui déclaré par PACKAGE.
Voir aussi FUNCTION et PROCEDURE.

Syntaxe
Déclaration

```
PACKAGE package_name IS
    [type_declaration]
    [subtype_declaration]
    [constant_declaration]
    .
    .
    [component_declaration]
END [package_name];
```

Corps du PACKAGE

```
PACKAGE BODY package_name IS
    [type_declaration]
    [subtype_declaration]
    [constant_declaration]
    .
    .
    [subprogram_declaration]
```

```
END [package_name];
```

Exemple

```
PACKAGE math IS -- déclaration de l'entête du package
FUNCTION minval (CONSTANT a,b : IN integer)
RETURN integer;
FUNCTION maxval (CONSTANT a,b : IN integer)
RETURN integer;
CONSTANT maxint: integer:=16#FFFF#;
END math;

PACKAGE BODY math IS -- déinition du corp du package
FUNCTION minval (CONSTANT a,b : IN integer) RETURN integer IS
BEGIN
    IF a < b THEN
        RETURN a;
    ELSE
        RETURN b;
    END IF;
END minval;
FUNCTION maxval (CONSTANT a,b : IN integer) RETURN integer IS
BEGIN
    IF a > b THEN
        RETURN a;
    ELSE
        RETURN b;
    END IF;
END maxval;
```

Port

PORT décrit un signal externe et visible d'une entité. Tous les PORT ont un mode ainsi qu'un type. Les PORT permettent la communication de l'architecture.
Le mode par défaut est IN.

Syntaxe

```
PORT port_name {, port_name} : mode
    type [(index_range [, index_range])] [:= default_value];
    [SIGNAL] port_name {, port_name} : mode
    type [(index_range [, index_range])] [:= default_value];
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY fulladd IS
    PORT (x,y,cin : IN bit;
    sum, cout : OUT bit);
END fulladd;
```

Port Map

PORT MAP est utilisé pour associer les broches d'un COMPONENT avec les signaux du montage.
Deux méthodes d'affectation sont possibles :

- par l'association des noms ;
- par l'association des positions (la plus courante).

Syntaxe

```
PORT MAP (pin_name => signal_name
        {, pin_name => signal_name});

PORT MAP (signal_name {, signal_name});
```

Procedure

Les PROCEDURES sont des blocs d'instructions qui permettent d'éffectuer des calculs et qui peuvent être appelées de différents endroits.
Les paramètres passés sont du type IN, OUT et INOUT et les objets de classe SIGNAL, CONSTANT ou VARIABLE. Le mode par défaut du type IN est de classe CONSTANT. OUT et INOUT sont considérés comme de classe VARIABLE. En fait, le paramètre CONSTANT IN peut être associé avec un signal, variable, constante ou une expression lorsque la procédure est appelée.
Une procédure peut être déclarée dans un PACKAGE ou dans le corps d'une ARCHITECTURE.
On peut déclarer des variables locales à l'intérieur d'une procédure. Ces variables perdent leur valeur à chaque sortie de la procédure et sont initialisées à chaque appel.
Bien qu'une PROCEDURE ne retourne pas de valeur contrairement à une FONCTION, on peut déclarer dans son entête plusieurs paramètres de type OUT ou INOUT et ainsi récupérés des valeurs de sortie.
Une procédure peut contenir un WAIT sauf si elle est appelée par un PROCESS avec une liste de sensibilité ou depuis une FONCTION.
Voir aussi FUNCTION et PACKAGE.

Syntaxe
Déclaration

```
PROCEDURE procedure_name
    {(parameter_list)};
```

Corps de la procédure

```
PROCEDURE procedure_name
    {(parameter_list)}
BEGIN
    [block_statement]
    [generate_statement]
```

```
END [procedure_name];
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.myfuncs.all; -- on déclare la bibliothèque utilisateur
myfuncs
ENTITY parity_add IS
    PORT (din : std_logic_vector(7 DOWNTO 0);
            douto, doute : OUT std_logic_vector(8 DOWNTO 0));
END parity_add;

ARCHITECTURE behv OF parity_add IS
BEGIN
    PROCESS(din)
    VARIABLE dintemp : std_logic_vector(7 DOWNTO 0);
    VARIABLE dptempo,dptempe : std_logic_vector(8 DOWNTO 0);
    BEGIN
        dintemp := din;
        addparity(dintemp,dptempo,dptempe);
        douto <= dptempo;
        doute <= dptempe;
    END PROCESS;
END behv;

LIBRARY ieee; -- déclaration de la procédure
USE ieee.std_logic_1164.all;
PACKAGE myfuncs IS
    PROCEDURE addparity(VARIABLE d : IN std_logic_vector(7 DOWNTO
    0);
        do, de : OUT std_logic_vector(8 DOWNTO 0)); -- les
        paramètres n'ont pas le même nom, c'est l'ordre qui
        compte
END myfuncs;

PACKAGE BODY myfuncs IS
    PROCEDURE addparity(VARIABLE d IN std_logic_vector(7 DOWNTO
    0); -- corps de la procédure
        do, de : OUT std_logic_vector(8 DOWNTO 0)) IS -- do et de
        sont les valeurs de sorties de la procédure
        VARIABLE temp : std_logic;
        BEGIN
            temp := '0';
            loop1: FOR i IN 7 DONWTO 0 LOOP
                temp := temp XOR d(i);
            END LOOP loop1;
            de := temp & d;
            do := NOT temps & d;
    END addparity;
END myfuncs;
```

Process

Un PROCESS est une déclaration concurrente dans une architecture. Un PROCESS peut être

sensible à une liste de signaux ou à un WAIT.
Les PROCESS contiennent seulement des déclarations séquentielles qui sont exécutées dans l'ordre spécifié.

Syntaxe

```
[label]: PROCESS [(sensitive_signal_name {,
sensitive_signal_name})]
    [constant_declarations]
    [variable_declarations]
BEGIN
    [sequential_statements]
END PROCESS [label];
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY nor2 IS
    PORT (a,b : IN std_logic:='0';
        qn : OUT std_logic);
END nor2;

ARCHITECTURE proc_behv OF nor2 IS
BEGIN
    ex1: PROCESS(a,b) -- ce PROCESS est décrit avec une liste de
    sensibilité
    BEGIN
        qn <= a NOR b;
    END PROCESS ex1;

    ex2: PROCESS -- ce PROCESS n'a pas de liste de sensibilité
    BEGIN
        qn <= a NOR b;
        WAIT ON a,b; -- WAIT permet d'attendre un événement sur a
        ou b
    END PROCESS ex2;
END proc_behv;
```

Exemple

```
SENSE_PROC: PROCESS (CLK)
begin
    if CLK'event and CLK='1' then
        Q2 <= D2;
    END IF;
END PROCESS SENSE_PROC;

WAIT_PROC: PROCESS
begin
    WAIT UNTIL CLK'event and CLK='1';
    Q1 <= D1;
END PROCESS WAIT_PROC;
```

Q

R

REM, RETURN.

Rem

Voir les opérateurs.

Return

Voir FUNCTION.

S

SELECT, SIGNAL, SUBTYPE.

Select

Dans la fonctionnalité, SELECT est équivalent à l'instruction CASE. SELECT ne peut toutefois affecter qu'une seule variable.

Syntaxe

```
[label:] WITH expression SELECT
            target <= waveform1 WHEN choices 1
                                {waveform1 WHEN choices N},
                                waveform1 WHEN OTHERS;
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY mux IS
    PORT (a,b,sel : IN std_logic;
            y : OUT std_logic);
END mux;
ARCHITECTURE behavior OF mux IS
BEGIN
    muxex: WITH sel SELECT
        y <= a WHEN '0',-- y = a si sel = 0 sinon
                    b WHEN '1', -- y = b si sel = 1 sinon
                    'X' WHEN OTHERS; -- y est indéterminé
END behavior;
```

Signal

SIGNAL déclare un signal permettant la communication entre les états concurrents à l'intérieur d'une architecture.
Dans la définition, on doit spécifier un type au signal et l'on peut attribuer une valeur par défaut.

Syntaxe

```
SIGNAL sig_name {, sig_name} : signal_type [:=initial_value];
```

On assigne une valeur à un signal en suivant ce format :

```
sig_name <= expression;
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY fulladd IS
    PORT (x,y,cin : IN bit;
        sum, cout : OUT bit);
END fulladd;

ARCHITECTURE behavior OF fulladd IS
    SIGNAL sum1, cout1, cout2, cout3 : bit;
BEGIN
    sum <= sum1 XOR cin;
    sum1 <= x XOR y;
    cout1 <= x AND y;
    cout2 <= x AND cin;
    cout3 <= y AND cin;
    cout <= cout1 OR cout2 or cout3;
```

```
END behavior;
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY compar2 IS
    PORT (a1, b1, a2 : IN bit := '0';
        c1 : INOUT bit;
        c2 : OUT bit);
END compar2;

ARCHITECTURE behavior OF compar2 IS
BEGIN
    c1 <= a1 XOR b1;
    c2 <= c1 XOR a2;
END behavior;
```

Subtype

SUBTYPE déclare un type qui est la combinaison de deux autres types.
Voir aussi TYPE.

Exemple

```
TYPE colors IS (red, yellow, blue, green, black);
SUBTYPE primary IS colors RANGE red TO blue;

TYPE integer IS RANGE -2147483647 TO 2147483647;
SUBTYPE absolu IS integer 0 TO 2147483647;

TYPE area IS ARRAY (natural RANGE <>, natural RANGE <>)
    OF bit;
SUBTYPE small_area IS area (0 TO 10, 0 TO 10);
```

T

TRANSPORT, TYPE.

Transport

Voir AFTER.

Type

Les TYPES prédéfinis

Les types prédéfinis reconnaissent six objets scalaires :

- **Bit** : représente un élément binaire avec les valeurs '0' et '1' ;
- **Bit_vector** : représente un tableau d'éléments binaires ;
- **Boolean** : représente une donnée pouvant être TRUE ou FALSE ;
- **Real** : représente un nombre réel ;
- **Integer** : est un entier de 32 éléments binaires ;
- **Character** : représente un charactère ASCII.

Dans la bibliothèque IEEE STD_LOGIC_1164 TYPE les types usuels sont déjà déclarés.
Le type *std_ulogic* contient 9 éléments de base.

```
'U' - Unitialized (élément de plus haute priorité)
'X' - Forcing Unknown
'0' - Forcing 0
'1' - Forcing 1
'Z' - High impedance
'W' - Weak unknown
'L' - Weak 0
'H' - Weak 1
'-' - Don't care (élément de plus basse priorité)
```

Ces objets scalaires peuvent être placés dans un contexte différent suivant leur déclaration.

Le TYPE énuméré

TYPE permet de définir un type utilisateur auquel on peut assigner un objet.
Tous les objets doivent être assignés à un type. Chaque TYPE déclaré doit être unique.

Syntaxe

```
TYPE identifier IS enumeration_type_literals;
```

Exemple

```
TYPE op_type IS (opadd, opor, opand, opxor);
TYPE letters IS ('A', 'a', 'R', 'r');
TYPE std_ulogic IS ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

Le TYPE numérique

Cette définition de type doit être dans une échelle de valeur.
La base par défaut est la décimale.

Syntaxe

```
TYPE identifier IS RANGE implementation_defined;
```

Exemple

```
TYPE byte IS RANGE 0 to 255; -- byte varie de 0 à 255
TYPE index IS RANGE 7 DOWNTO 0; -- index varie de 7 à 0
SUBTYPE byte IS INTEGER RANGE 0 TO 255; -- byte est un entier de 0
à 255
```

Le TYPE physique

Cette définition de type représente une quantité physique.
La base des unités doit être spécifiée. Les valeurs sont calculées à partir de cette base.

Syntaxe

```
TYPE identifier IS RANGE implementation_defined;
    UNITS
        base_unit_declaration
        [secondary_unit_declaration]
    END UNITS;
```

Exemple

```
TYPE time IS RANGE implementation_defined;
    UNITS
        fs;
        ps = 1000fs;
        ns = 1000fs;
        us = 1000ns;
        ms = 1000us;
        sec = 1000ms;
        min = 1000sec;
        hr = 1000min;
    END UNITS;
```

Le TYPE tableau

Le type tableau est un groupement de types identiques.
Les tableaux peuvent être multidimentionels.
Il existe deux sortes de tableaux :
Les tableaux avec contrainte (Constrained Array)

Ces tableaux sont définis avec des dimensions figées (rangées colonnes).

Les tableaux sans contrainte (Unconstrained Array)

Ces tableaux sont définis sans préciser les dimensions. Le tableau est donc dynamique.

Syntaxe

```
TYPE identifier IS ARRAY
    [unconstrained_array_definition];
    [constrained_array_definition];
```

Exemple

```
TYPE data_bus IS ARRAY (0 TO 31) OF bit; -- tableau de 32 bit
TYPE bit4 IS ARRAY (3 DOWNTO 0) OF bit; -- tableau de 4 bit : bit4
(3) à bit4(0)

TYPE string IS ARRAY (positive RANGE <>) OF character;
TYPE bit_vector IS ARRAY (natural RANGE <>) OF bit;

TYPE dim2 IS ARRAY (0 TO 7, 0 TO 7) OF bit; -- tableau à deux
dimensions
```

U

USE.

Use

Voir LIBRARY.

V

VARIABLE.

Variable

Les variables sont utilisées dans un PROCESS, une PROCEDURE ou une FONCTION.
On peut leur donner une valeur initiale. Cette valeur initiale est attribuée à chaque appel de la PROCEDURE ou de la FONCTION. Elles servent à manipuler des variables temporaires/intermédiaires pour faciliter le développement d'un algorithme séquentiel.

Syntaxe

```
VARIABLE var_name {, var_name} : type [:= value];
```

Exemple

```
VARIABLE i : INTEGER RANGE 0 TO 3; -- valeur intiale 0
VARIABLE x : std_ulogic; -- valeur intiale 'U'
```

Exemple

```
FUNCTION parity (x : std_ulogic_vector) -- recherche la parité
d'un vecteur
            RETURN std_ulogic IS
    variable tmp : std_ulogic := '0';
begin
    FOR j IN x'RANGE LOOP
        tmp := tmp XOR x(j);
    END loop; -- fin de la boucle
    RETURN tmp;
END parity;
```

W

WAIT, WHEN, WHILE, WITH, WORK.

Wait

WAIT est utilisé dans une déclaration séquentielle de type PROCESS ou PROCEDURE.
WAIT remplace une liste de sensibilité pour contrôler l'exécution et la suspension d'un PROCESS.
WAIT peut être placé n'importe où dans le PROCESS.
Sans WAIT ou sans liste de sensibilité, un PROCESS se boucle indéfiniment.

Il existe trois syntaxes différentes autour de WAIT.

WAIT ON est équivalent à une liste de sensibilité. La liste des signaux suit.
Ainsi :

```
process (A,B)
begin
    -- sequential statements
end process;
```

est équivalent à :

```
process
begin
    -- sequential statements
```

```
    WAIT ON a,b;
end process;
```

La condition testée par WAIT UNTIL d'expression booléenne doit parvenir à TRUE pour continuer l'exécution.

```
process
begin
    wait until CLK'event and CLK='1';
    Q1 <= D1;
end process;
```

Le temps imparti par WAIT FOR spécifie la durée maximale pendant laquelle le PROCESS reste suspendu.

```
STIMULUS: process
begin
        EN_1 <= '0';
        EN_2 <= '1';
    wait for 10 ns; -- attend 10ns avant de passer à la ligne
    suivante
        EN_1 <= '1';
        EN_2 <= '0';
    wait for 10 ns;
        EN_1 <= '0';
    wait; -- attent indéfiniment
end process STIMULUS;
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY upcont IS
    PORT (i : INOUT integer:=1; -- déclare et initialise i à 1
    clk : IN std_logic:='0';
    count : OUT integer);
END upcont;

ARCHITECTURE cpt OF upcont IS
BEGIN
    sens1: PROCESS
    BEGIN
        i <= i+1; -- incrémente i. i vaut 2 après le premier
        passage
        WAIT ON clk; -- attend une transition de clk (front
        montant ou descendant)
    END PROCESS sens1;

    sens2: PROCESS
    BEGIN
        count <= i; -- met à jour la sortie count
        WAIT ON i; -- attend un changement de i
    END PROCESS sens2;
END cpt;
```

When

WHEN est utilisé dans une liste de sélection avec l'instruction CASE.
On le retrouve aussi lors de l'assignement conditionnel d'un signal. Dans cette utilisation, il remplace l'écriture du IF THEN avantageusement car il permet l'imbrication des tests.

Syntaxe

```
[label:] target <= [options] value_expression [AFTER
time_expression] WHEN condition1
            ELSE
                [options] value_expression [AFTER
                time_expression] WHEN condition2;
```

Exemple

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY mux IS
    PORT (a,b,sel : IN std_logic;
            y : OUT std_logic);
END mux;
ARCHITECTURE behavior OF mux IS
BEGIN
    muxex: y <= a WHEN sel = '0' ELSE -- y = a sir sel = 0 sinon
                b WHEN sel = '1' ELSE -- y = b si sel = 1 sinon
                'X'; -- y est indéterminé
END behavior;
```

While

WHILE est une instruction de bouclage. Elle s'utilise avec LOOP.
On place la condition de fin de sortie dans l'instruction même contrairement au LOOP seul où on est obligé d'avoir recourt à la commande EXIT. On peut toutefois utiliser EXIT ou NEXT.
Le bouclage se fait tant que la condition reste TRUE.
Remarque : si la condition est FALSE dès le départ, la boucle ne sera pas exécutée une seule fois.

Syntaxe

```
[loop_label:] WHILE condition LOOP
        sequence_of_statements
END LOOP [loop_label];
```

Exemple

```
USE ieee.std_logic_1164.all;
ENTITY clock IS
    PORT (clk1 : INOUT std_logic := '0');
END clock;
ARCHITECTURE behv OF clock IS
CONSTANT pulse_time : time := 25ns;
```

```
BEGIN
    clkit: PROCESS
    VARIABLE a : integer := 1;
    BEGIN
        loop1: WHILE (a <= 10) LOOP -- la variable de boucle est
        a inférieure à 10
            WAIT FOR pulse_time; -- attend la valeur de
            pulse_time
            clk1 <= NOT clk1; -- complémente clk1
            a := a + 1; -- incrémentation de a
        END LOOP loop1;
    END PROCESS clkit;
END behv;
```

With

Voir SELECT.

Work

Voir Library.

X

XOR.

Xor

Voir les opérateurs.

Y

Z

?

+, -, *, **, /, &, =, >, <, <=, >=

Voir les opérateurs.

Index