**初めてでも使えるVHDL文法ガイド ―― 記述スタイル編**

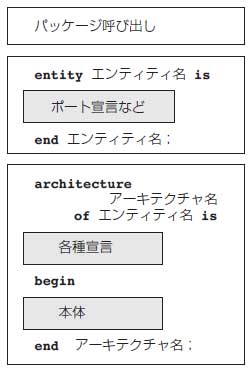
**VHDLは，もともと米国防総省がハードウェア仕様記述言語として用意したもので，Verilog HDLに比べて「重い文法」になっています．ただし，論理合成可能な回路記述や，シミュレーションのための記述に限れば，それほど差はありません．概念上よく似た部分も多数あります．ここではVHDLの記述スタイルについて解説します．（編集部）**

**1. 基本構造**

**● VHDL記述の基本構造はentityとarchitecture**  
　**図1**に示すように，VHDLの基本構造は，

1. パッケージ呼び出し
2. エンティティ（entity）
3. アーキテクチャ（architecture）

からなります．この構造で，回路記述やシミュレーション記述を行います．

**  
図1　VHDL記述のブロック構造**

　パッケージ呼び出しとは，各種演算子や関数などを定義した「パッケージ」を呼び出す部分です．実設計上，各ブロックの記述の先頭でかならず呼び出します．すべてのブロックに付き物と考えてください．記述上，とても重要な概念ですので，3節で詳しく説明します．

　エンティティ（entity）は，外部とのインターフェースを記述した部分です．回路図で言えば「シンボル」に相当します．

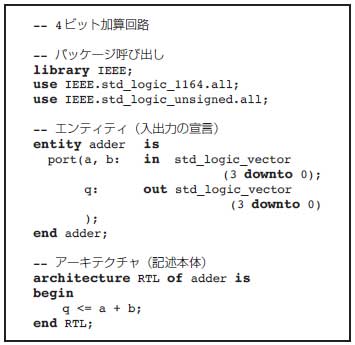
　一方アーキテクチャは，内部の動作を記述したもので，回路の本体です．回路図で言えば，「シンボル」に対応した内部の回路です．回路の構造や動作は，アーキテクチャ内で記述します．

　以上の基本構造で，回路の記述やネットリストの記述，シミュレーション用の記述が可能です．

**● エンティティは複数のアーキテクチャを持てる**　一つのエンティティは，複数のアーキテクチャを持つことができます．**図2**の例のように，一つのブロックでビヘイビア，RTL，ゲートの三つのアーキテクチャに対応することもできます．実行時にどれを選択するかは，コンフィグレーション宣言により対応づけられます．

|  |
| --- |
| http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f02a.gif |
| http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f02b.jpg  **図2　エンティティとアーキテクチャ** |

**● 基本構造の実例**　簡単な記述をもとに，基本構造の実際を説明します．**リスト1**は「4ビットの加算回路」です．二つの入力aとbの和をqに出力しています．

**リスト1　加算回路記述例**  


**（1） パッケージ呼び出し部**

　まずは「パッケージ呼び出し」です．ここでは，VHDL標準ライブラリIEEEの中の，std\_logic\_1164およびstd\_logic\_unsignedパッケージ内で定義してあるものすべてを読み込んでいます．型の宣言や，演算子を用いるために必要です．

**（2） エンティティ**

　エンティティ内では，ポート宣言を行っています．

**<ポート信号名>：<入出力の方向> <型>**

の形式で宣言します．inとoutは予約語です．

　<型>は，VHDLでは重要です．基本的に型が一致した信号どうしでのみ演算や接続が可能です．型が一致しないと文法エラーになります．ここでは，std\_logic\_vectorという型で，4ビットのポートを宣言しています．

　downtoは，降順で範囲を指定する予約語です．昇順の場合toを使います．実際の設計ではもっぱら降順を使います．

**（3） アーキテクチャ**

　回路の動作や構造はここで記述します．アーキテクチャ名は任意です．ブロックごとに変える必要はありません．筆者は，回路の記述ではRTL，シミュレーション記述ではSIMなどに固定しています．

　回路の動作は加算演算子+と，代入文（この場合，アーキテクチャ直下に記述したので「同時処理代入」と呼ぶ）で記述しています．

## 2. RTL記述

　一般に論理合成を前提とした回路記述をRTL記述と呼びます．RTLの回路記述には，次の四つの基本記述スタイルがあります．

1. 簡単な組み合わせ回路
2. 複雑な組み合わせ回路
3. 順序回路
4. 下位ブロック呼び出し

　どのような回路規模であっても，この四つの記述スタイルから構成されます．

### 2.1 簡単な組み合わせ回路

**図3**は，1行で記述できる組み合わせ回路です．アーキテクチャ内の同時処理代入文で記述します．それぞれの動作については，ブロック図（**図3（b）**）を見て類推してください．

|  |
| --- |
| http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f03a.jpg  （a） architecture内で記述する |
| http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f03b.gif  （b） 組み合わせ回路ブロック |

**図3　実例1 ――簡単な組み合わせ回路**

　信号が多ビットか1ビットかは，宣言で決まります．したがって，セレクタや加算回路などの記述はビット数に影響されません．

**図3（a）**の中の重要な項目を解説します．

**（1） 定数の表現**  
　定数は，'1'，"1001"のように文字や文字列として表現します．16進数や10進数の表現も可能ですが，制約が多く，実質的に2進数表記のみ用います．

**（2） 論理値**　1ビットの信号が取りうる値は，論理回路の世界では0と1ですが，VHDLの世界ではこれだけではありません．不定値Xや，ハイ・インピーダンスのZがあります．正確に言えば，これら論理値もパッケージの中で定義されています．

**（3） 演算子**  
　not，andなどは，論理演算子です．演算子には，文法ガイド編に示すものがあります．

**（4） 代入記号**  
　信号に対する代入には，<=を使います．ここでの代入は，アーキテクチャ直下に記述する「同時処理代入文」です．記述の順番に代入を行うのではなく，「同時」に代入されます．「接続している」と考えるとわかりやすいでしょう．

**（5） 条件分岐**  
　when ～ else を用いて，2方向分岐を記述できます．when直後の条件式が，

　　真：　when直前の式の値を，右辺値とする  
　　偽：　else直後の式の値を，右辺値とする

となります．

　when ～ elseの構造は，文法上の式ではないので，続けて演算することはできません．

　q <=(d1 **when** sel='1' **else** d0)+ d2; -- エラー

などとすると，文法エラーです．

**（6） コメント**  
　--で始まり，行末までがコメントです．いちおう漢字類も使えますが，文字コードによっては文法エラーを起こすことがあります．複数行コメントはありません．

**（7） 文字の大小の区別**  
　VHDLでは予約語も含めて文字の大小を区別しません．したがって，すべて大文字で書いても，混在しても同じものとして扱います．本稿では基本的に小文字で記述しました．

### 2.2 複雑な組み合わせ回路

　1行では表現できない複雑な組み合わせ回路は，process文を用いて記述します．**図4**に「2 to 4」のデコード回路を示します．完全なブロックの形式で記述してあるので，このまま論理合成可能です．

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f04a.jpg （a）　記述例 | http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f04b.gif （b）　ブロック図 |   **図4　実例2 ―― 複雑な組み合わせ回路** |
|  |

　processの次のカッコの中に信号リスト（これをセンシティビティ・リストという）を記述し，これらの信号に変化があったときにプロセス内を実行します．この場合，入力dの2ビットのいずれかに変化があれば，プロセス内のcase文を実行します．組み合わせ回路をプロセス文で記述するときは，入力すべてをセンシティビティ・リストに記述します．

　case文は多方向の分岐です．case文の入力dと，when直後に記述した値が一致すれば，=>以降を実行します．dは2ビットですので4方向に分岐し，出力に対する代入文を実行しています．

　othersは，どれにも一致しなかった場合を指定するものです．ここでは，4通りすべてを記述してあるようですが，論理値にはX（不定値）やZ（ハイ・インピーダンス）があります．case文は，入力が取りうるすべての分岐を記述しなければならないので，実質的にothersは必須だと考えてください．

### 2.3 順序回路

　順序回路とは，値を保持するフリップフロップやラッチを含む回路です．記述スタイルも，組み合わせ回路とは異なります．

**リスト2**に4ビットのバイナリ・カウンタを例に順序回路を記述してみました．**リスト2（a）**はクロック同期のリセット，リスト2（b）は非同期リセットです．

**リスト2　実例3 ―― 順序回路**

|  |
| --- |
| http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_l02.jpg  （a） 記述例（同期リセット）　　　　　　　　　　　　　　　　　　　　　　   　　（b）　記述例（非同期リセット） |
|  |

　順序回路もプロセス文を使って記述します．センシティビティ・リスト（processのカッコの中に記述する信号リスト）には，同期タイプはクロックのみ，非同期タイプはクロックと非同期制御信号（この場合，非同期リセットのres）を記述します．

**● if文でクロック判別**　if文は2方向の分岐です．if直後の条件式を評価し，

　　真：　then以降  
　　偽：　elseがあればelse以降

を実行します．

　予約語eventはアトリビュート（属性）と呼びます．信号や変数，定数などに付加されている一種のデータです．eventアトリビュートは，信号の変化の有無を意味します．つまり，

　ck'event **and** ck='1'

は，「ckに変化があり，かつ1のとき」，つまりckの立ち上がりを意味します．したがって，同期リセットの例（**リスト2（a）**）では，すべての動作はクロックの立ち上がりで行われます．

　一方，非同期リセットの例（**リスト2（b）**）では，クロックの有無をチェックする前に，リセット信号の判別があります．したがって，クロックに関係なく（クロックに非同期に）リセットが有効になります．

　ここで用いているels ifも予約語です．else ifと記述した場合と同じ意味です．

**● 内部参照のため信号宣言**　VHDLは，出力ポート信号を内部で参照できないという制約があります．したがって，

　q\_reg <= q\_reg + "1";

のように，内部で参照し，演算する記述がある場合，出力ポート信号を使うことができません．つまり，ここでは，

　q <= q + "1";

と記述すると，文法エラーです．

　そこで内部参照用に内部信号を用います．信号の宣言には，signalを使います．

　　signal：**<信号名> <型>；**

の形式で宣言します．

　　signal：**<信号名> <型>：＝<式>；**

とすれば，初期値も与えられます（論理合成では無視されるが...）．

**● もう一つの記述スタイル**  
　順序回路の記述には，もう一つの方法として，

**process begin**　**wait until**ck'event **and**ck='1';  
　**if**res='1' **then**  
　 q\_reg <= "0000";  
　 **else**　 q\_reg <= q\_reg + "1";  
　 **end if;**　**end process;**

のようなスタイルもあります．

　センシティビティ・リストのないprocess文は無限ループになります．

　wait untilにより，クロックの立ち上がりを待ち，if文以下の処理を行います．この方法も論理合成可能ですが，非同期リセットなどの記述ができないため，一般的に**リスト2**の記述スタイルを使います．

### 2.4 下位ブロック呼び出し

　下位のブロックを接続する記述スタイルを解説します．ここでは，4個のフル・アダーを並べた4ビットのリップル・キャリ加算回路を例にしました（**図5**）．

|  |  |  |  |
| --- | --- | --- | --- |
| http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f05a.jpg （a）　記述例 | |  | | --- | | http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f05b.jpg （b）　フル・アンダー記述例 | | http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f05c.gif  （c）　ブロック図 | |

**図5　下位ブロックの呼び出し**

　アーキテクチャ内で，呼び出すブロックのポート部分をあらかじめコンポーネント宣言により明示しておきます．アーキテクチャ内で呼び出しているブロックすべてです．ここでは1種類のフル・アダーを4回呼び出しているので，コンポーネント宣言は一つだけです．中身はエンティティと同じなのでコピー＆ペーストして作成します．

　下位ブロック接続時に使用する内部ネットは，すべて宣言する必要があります．アーキテクチャの宣言部分（isとbeginの間）で宣言します．coutは，フル・アダーのキャリ信号，すなわち各フル・アダー間の接続信号です．

　呼び出すブロックのポートと信号の接続には2形態あります．

1. 順番による接続
2. 名まえによる接続

です．名まえによる接続は，  
  
　　定義側ポート名 ＝＞ 接続信号名  
   
です．

　接続する信号部分は，式や定数を記述することができません．したがって，フル・アダーの初段のキャリ入力（0 固定）は，内部信号zeroを宣言し，アーキテクチャ内で 0に固定しています．

## 3. パッケージ

　ハードウェア・システムを設計するうえで，共有化できる部分をまとめたものがパッケージです．

　実際の設計では次のような目的でパッケージが使われます．

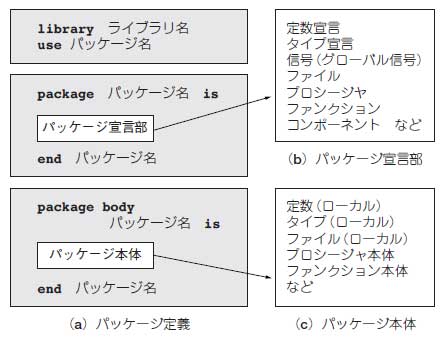
**（1） 標準パッケージ**　論理値，演算子，型などを定義して，多くの設計や各種ツール間で共通に使えるようにしたパッケージ．

**（2） 設計支援グループが供給するパッケージ**　論理合成に適した形で設計されたHDLライブラリ．

**（3） 設計者が構築するパッケージ**　設計対象に固有の定数や型を定義したり，シミュレーション記述のための各種ルーチン（プロシージャ）を定義するパッケージ．

**● パッケージの定義**　設計者みずからパッケージを構築する際には，パッケージの定義を行います（**図6（a）**）．まず，パッケージの定義で必要なほかのパッケージを呼び出します．パッケージは，宣言部と本体（body）で定義されます．パッケージ宣言部（**図6（b）**）では，定数やタイプ，信号などが宣言されます．またプロシージャやファンクションは入出力の宣言のみで，処理本体はパッケージ本体で定義します．

　パッケージ本体（**図6（c）**）では，プロシージャやファンクションの処理のほかに，これらで用いるローカルな定数や型を宣言できます．プロシージャやファンクションがないパッケージの場合，このパッケージ本体は不要です．

**  
図6　パッケージの定義**

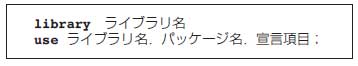
**● パッケージの所在と可視範囲**　パッケージは，ライブラリの中に存在します．例えば，std\_logic\_unsignedパッケージはIEEEライブラリの中に存在します．したがってパッケージを呼び出す際には，ライブラリを指定してから，use節で呼び出します（**図7**）．このとき，パッケージに含まれるすべての宣言を用いるなら，

**use** IEEE.std\_logic\_unsigned.all;

とall指定し，一つの宣言だけなら，

**use** IEEE.std\_logic\_unsigned.conv\_integer;

と指定します．

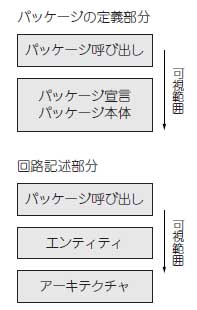
**  
図7　パッケージの呼び出し**

　設計者が記述したパッケージは，デフォルトでworkライブラリに含まれます．したがって，

**use** work.MYPAC.all;

のように呼び出します．workライブラリのlibrary宣言は不要です．

　回路記述やシミュレーションの記述には，パッケージ呼び出しが必須です．パッケージの可視範囲は，パッケージの定義部や，エンティティに付随するアーキテクチャおよびコンフィグレーションまでだったからです（**図8**）．一度呼び出せばずっと有効というわけではありません．

**  
図8　パッケージの可視範囲**

**● 代表的なパッケージ**  
　VHDLで設計するとき，最初に接するのがstd\_logic\_unsignedなどの「標準パッケージ」でしょう．これらは別途定義され，シミュレーション起動時に自動で読み込まれるので，呼び出すだけで使用できました．

　これらのパッケージの中では，論理値，演算子，型，単位，変換関数などが細部に至るまで定義されています．std\_logicなども，これらの中で定義されている一つの型にすぎません．しかし，回路記述のうえで標準的に使われています．

　代表的な標準パッケージには，次のものがあります．

* std\_logic\_1164 ：基本パッケージ
* std\_logic\_unsigned ：符号なし演算用（std\_logic\_1164の補足）
* std\_logic\_signed ：符号付き演算用（std\_logic\_1164の補足）
* std\_logic\_arith ：符号付き，符号なし混在演算用（std\_logic\_1164の補足）
* textio ：ファイル・アクセス用
* std\_logic\_textio ：textioの補足パッケージ

**● 型（データ・タイプ）**　VHDLには，数多くのデータ・タイプがあります．また設計者みずからデータ・タイプを作り出すことが可能です．タイプ宣言により，無限のデータ・タイプを定義できます．

　しかし，現実の回路設計の場合，論理合成のことを考えると，使うデータ・タイプは限られます．さまざまなデータ・タイプを用いて記述しても，論理合成後のネットリストでは，特定のデータ・タイプに固定されてしまいます．データ・タイプを駆使しても，ゲート・レベル検証を考えると無駄になってしまうことがあります．

　実際のところVHDLに標準で備えられているデータ・タイプは，回路記述ではあまり使いません（**表1**）．例えば標準のbitは，2値（0，1）しかなく，不定もハイ・インピーダンスもありません．これでは回路記述に使えないので，std\_logicとstd\_logic\_vectorをおもに使います．これらは9値で'X'や'Z'もあり，std\_logic\_1164パッケージで定義されています．

**表1　データ・タイプ**

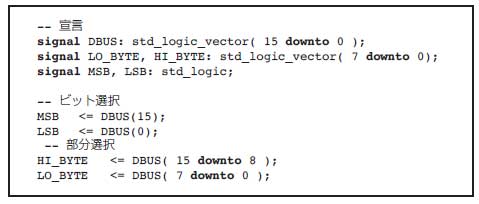
|  |  |
| --- | --- |
| http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_hyo01a.jpg （a） 回路記述でよく使う型 | http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_hyo01b.jpg （b） シミュレーション記述でよく使う型  http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_hyo01c.jpg （c ） ほどんど使わない型 |

**● 型が合わないとエラー**  
　代入や演算のときには型を合わせなければなりません．次のように型を合わせるためのくふうが必要な場合があります．**図9**は4 to 1のセレクタです．入力はそれぞれ1ビットなので，ビット選択（**リスト3**）を用いて1行で記述したいところです．しかし，

　q <= d(sel);

と記述すると文法エラーになります．selはstd\_logic\_vector型ですが，ビット選択で記述できるのはinteger型だからです．したがってstd\_logic\_vector型からinteger型への型変換が必要になります．パッケージstd\_logic\_unsignedで用意されているのが型変換関数conv\_integerです．

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f09a.jpg （a）　記述例 | http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f09b.gif （b）　ブロック図 |   **図9　型変換は必須** |
|  |

**リスト3　多ビット信号のビット選択と部分選択  
  
**

　これとは逆に，一見型が合わなくてもエラーにならない場合があります．例えば，

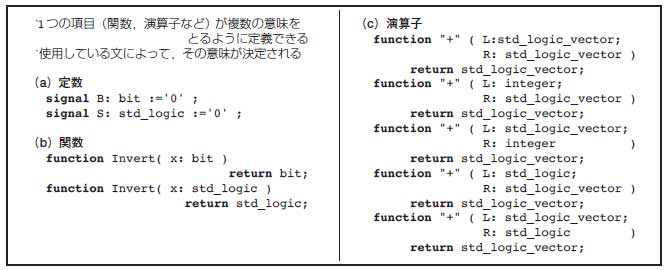
**signal**a, b: std\_logic\_vector(3 **downto** 0);

のとき，パッケージstd\_logic\_unsignedを用いれば，

　a <= b + 1;  
　a <= b + '1';  
　a <= b + "0001";

のいずれも，エラーではなく同じ動作（同じ回路）になります．

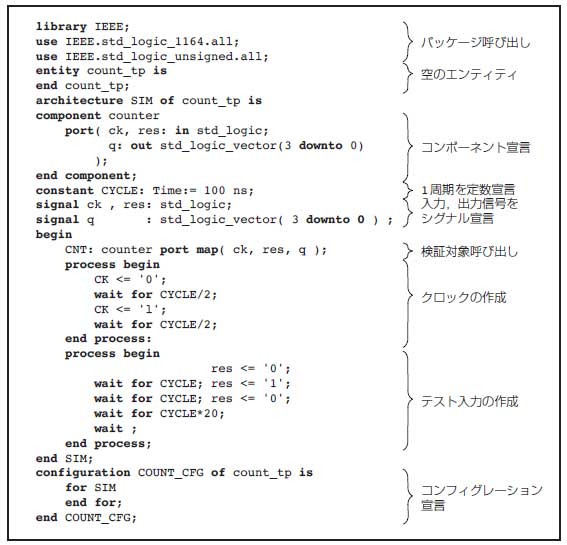
　これは，パッケージstd\_logic\_unsignedの中で，さまざまな型に対する加算演算を定義してあるからです．これをオーバロードと呼びます（**リスト4**）．

**リスト4 　オーバーロード  
  
**

## 4.シミュレーション記述

### 4.1 シミュレーション記述の構造と記述例

**● シミュレーション記述も回路記述と同じ構造**  
　シミュレーション記述の実際の例を**リスト5**に示します．シミュレーション対象は，**リスト2**の4ビット・カウンタです．

**リスト5 　シミュレーション記述  
  
**

　シミュレーション記述は，回路記述とほぼ同様に，

* パッケージ呼び出し
* エンティティ
* アーキテクチャ
* コンフィグレーション宣言

から構成されています．

　呼び出すパッケージは，回路記述に用いる標準的なもの以外に，シミュレーション記述に必要な，例えばファイル・アクセス用のパッケージを呼び出すこともあります．**リスト5**では，回路記述と同じものだけを呼び出しています．

　シミュレーションの最上位階層には，入力や出力はありません．したがってエンティティは，ポート宣言などのない空のエンティティとなります．

**● アーキテクチャ宣言部で，接続の準備**  
　アーキテクチャの宣言部（isとbeginの間）には，

* コンポーネント宣言（検証対象のエンティティ部分）
* 定数宣言（クロックの周期などを設定）
* 内部信号宣言

があります．

　コンポーネント宣言は下位ブロックを呼び出すために必要です．階層を持った回路記述でも同様でした．コンポーネント宣言の記述内容は，その階層のエンティティとほぼ同じです．

　定数は，コンスタント宣言を用いて設定します．この例では，1周期を示す定数CYCLEを100nsに設定しています．

　VHDLは，定数にも型があります．定数CYCLEは，時刻型の定数です．この型は以下のような単位を付けることができます．

　　fs, ps, ns, us, ms, sec, min, hr

　これらの単位表現と，数値の間にはスペースが必要です．100nsなどと付けて記述すると文法エラーです．

　呼び出す検証対象ブロックに接続される信号は，すべて信号（signal）宣言します．

**● アーキテクチャ本体で，シミュレーション入力を記述**  
　検証対象の呼び出しは，アーキテクチャ本体で行います．もし，シミュレーション・モデルなどがあれば，ここで検証対象に接続します．

　回路記述では，プロセス文のセンシティビティ・リストが必要でした．組み合わせ回路では入力のすべてが，順序回路ではクロックや非同期のリセット信号などがセンシティビティ・リストに記述されていました．

　クロックの記述のようにセンシティビティ・リストがなければ，「無限ループ」として動作します．**リスト5**では，半周期ごとにCK信号に0，1，0，1，...を印加しています．

　テスト入力の簡単な記述方法に，「遅延を与えて入力を変化させる」方法があります．ここでは，プロセス文の中で入力resに順次信号を与えています．プロセス文は「無限ループ」的な動作ですが，無条件のwait文を記述することで，処理を停止することができます．

　wait文には，

1. wait for 時間　　　　 ：　記述した時間待つ
2. wait until 条件　　　：　条件が真になるまで待つ
3. wait on 信号，信号，...：　信号に変化があるまで待つ

の三つの形式があります．

　さらに，これらを組み合わせて，

　wait on 信号 for 時間　　：信号に変化があるかどうか時間が来るまで待つ

　wait until 条件 for 時間：条件が真になるかどうか時間が来るまで待つ  
などとできます．

　また，  
  
　wait  
  
と，条件を何も記述しなければ，それ以降の記述は実行されません．processの中で記述されていれば，そのプロセス文の実行は停止します．

**● コンフィグレーション宣言はシミュレータに依存**  
　コンフィグレーション宣言は，本来エンティティとアーキテクチャを結びつけるものです．しかし実際の回路記述では，複数のアーキテクチャを持つことはほとんどありません．つまり，エンティティに対して一つのアーキテクチャしか持ちません．

**リスト5**のシミュレーション記述でも同様ですが，シミュレータが最上位階層のコンフィグレーション宣言を必要としているため，エンティティcount\_tpとアーキテクチャSIMを結びつけるコンフィグレーション宣言を記述しています．

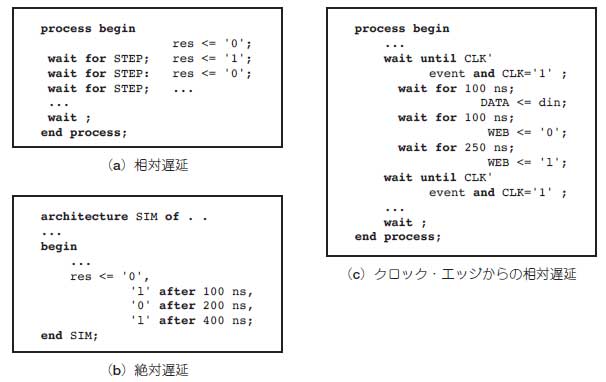
### 4.2 テスト入力の作成

　「シミュレーション記述を作成する」ことは，言い換えればテスト入力を作成することにほかなりません．内部信号を宣言したり，検証対象を呼び出したりすることは，シミュレーション記述作成の準備にすぎません．

　以下に3種類のテスト入力作成方法を解説します．

**● その1：遅延を与えて入力を変化させる**  
　簡易的なテスト入力作成方法です．ここでは遅延の与えかたに応じて，3種類の方法を示します．

**（1） 相対遅延を与える**  
　まず相対的な遅延を与える方法です（**リスト6（a）**）．wait forを用いて単純に「待ち」を作っています．一般的にシステム・クロックの1周期ごとに遅延させ，入力を変化させます．**リスト6（a）**のように1周期の時間を定数STEPにコンスタント宣言しておき，STEPごとに入力を変化させます．遅延を与えることがシミュレーションを進めることになります．

**リスト6　テスト入力作成法1 ―― 遅延を与えて入力を変化  
  
**

**（2） 絶対遅延を与える**  
　次の方法が絶対遅延です（**リスト6（b）**）．この例では，res信号に，初期値'0'，絶対時間100nsで'1', 200nsで'0'というように，シミュレーション開始を0nsとした絶対時間で信号が変化する時間を表現しています．信号resについては同時処理代入文で記述しています．つまり相対遅延と違って，プロセス文中ではなくアーキテクチャ内の直下で記述しています．

**（3） クロック・エッジからの相対遅延**  
　もう一つの方法がクロック・エッジからの相対遅延です（**リスト6（c）**）．システム・クロックCLKの立ち上がりを待って，ここから10ns，25nsなどの遅延を与えています．この，

**wait until**CLK'event **and** CLK='1';

という記述は，順序回路のもう一つの記述スタイルでも紹介したように，  
  
　「CLKの立ち上がりを待つ」  
  
という意味です．

**● その2：テスト入力をファイルから読み込む**　多量のテスト・パターンを一気に印加してシミュレーションを行うためには，パターン・ファイルを読み込んでテスト入力として印加する方法があります．

**図10**にファイルからの読み込み例を示します．ファイル・アクセス用の手続き類は専用のパッケージに格納されており，これらを呼び出す必要があります．TEXTIOと呼ばれるパッケージ類です．

|  |  |
| --- | --- |
| http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f10a.jpg （a）　入力ファイル | http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f10b.jpg （b）　記述例 |

**図10　テスト入力作成法2 ―― ファイルからの読み込み**

　ここでは，入力ファイルを1行ずつ読み込んで，WEB信号とDIN信号に印加しています．

　ファイル・アクセスのために，

* ファイル・アクセス用のパッケージ（std\_logic\_textioとtextio）の読み込み
* ファイル変数宣言（file VECTOR: TEXT is in "pat\_file. txt";）
* ライン・バッファ宣言（variable V\_LI: line;）

などの準備を行います．

　ファイルの読み込みは，

* readlineでライン・バッファへ読み込む
* readで1信号ずつ読み込む

の手順で行います．

**● その3：プロシージャを使ったテスト入力**  
　プロシージャを用いれば，シミュレーション記述のサブルーチン化が可能です．**図11（a）**のようなCPUのバスをイメージした入力を与える場合，1クロックごとに印加するのではなく，バスのシーケンスそのものを記述できます．

|  |
| --- |
| http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f11a.gif  　　　（a）　書き込みシーケンスの波形 |
| http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f11b.jpg 　　（b）　記述例 |

**図11　テスト入力の作成法3 ―― プロシージャ（タスク）を使う**

**図11（b）**では，アドレスとデータを引き数にした，write\_regというプロシージャを宣言しています．これは**図11（a）**の書き込みシーケンスをそのまま一つのプロシージャで表現したものです．これにより，

　write\_reg（アドレス，データ）;

の形式で呼び出すことができ，わかりやすく確実なテスト入力が得られます．

　プロシージャの定義は，

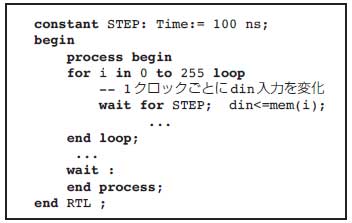
* パッケージ
* エンティティ
* アーキテクチャの宣言部
* プロセスの宣言部

などで可能です．一般的には，パッケージ内か，アーキテクチャの宣言部で定義します．

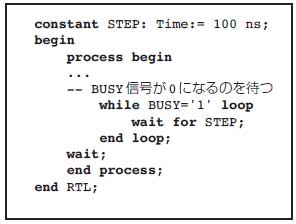
### 4.3 シミュレーション記述に役立つ各種構文

　シミュレーション記述の中で，ループ構造を使う場合が多々あります．ループ構造は，プロセス文中やプロシージャ内で記述できます．

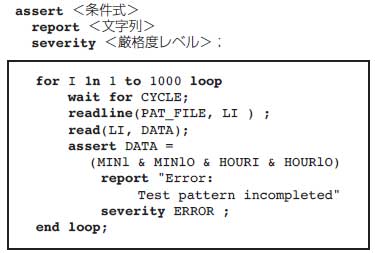
**● for文**　VHDLのfor文は，純粋にループを作る構文です（**図12**）．ループ変数は，ループを回るたびに1ずつ増減します．ダウン・カウントの場合にはtoの代わりにdowntoを用います．ループ変数は，暗黙的に宣言された整数型の変数です．したがって未宣言で使用できます．また文法上，ループ変数に対して代入することはできません．

**  
図12　for文**

**● while文**  
　条件判断が先にくるループです（**図13**）．whileの後の条件式が真であればloop以下を繰り返し実行します．ここでは，BUSY信号が0になるまで遅延を与え，シミュレーションを進めています．回路に対して起動信号を与え，処理の終了を待つ場合に使う手法です．処理が終了するクロック数が不明なときに有効な手段です．

**  
図13　while文**

**● シミュレーション結果をチェックするassert文**  
　シミュレーション結果に応じて，メッセージを出力したりシミュレーションを中止するための構文です．**図14**に示した例は，あらかじめファイルに保存しておいたシミュレーションのログ・ファイルと現在の結果を比較し，不一致ならメッセージを表示してシミュレーションを中止します．RTL検証とゲート・レベル検証の比較などに有効な手法です．

**  
図14　assert文**

　条件式が偽なら，report以降の文字列を出力します．厳格度レベルには，

　　NOTE, WARNING, FAILURE, ERROR

があります．FAILUREとERRORはシミュレーションを中止します．厳格度レベルを省略するとERRORとして扱います．

### 4.4 デルタ遅延

　VHDLには，仮想の遅延であるデルタ遅延があります．これは，コンピュータ上のシミュレーションで，ハードウェアの並列処理を実現するために考え出された無限小の遅延です．シミュレーション時刻上は遅延量0として扱われます．

**● デルタ遅延による信号の変化**  
　**図15**を用いてデルタ遅延を解説します．3個の直列接続されたインバータを想定します．同時処理代入文を使って，アーキテクチャ直下に記述しました（**図15（a）**）．

|  |  |
| --- | --- |
| http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f15a1.gif http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f15a2.jpg （a）　回路とVHDL記述 | http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f15b.gif （b）　各信号のようす |

**図15　デルタ遅延**

　入力Aが時刻Tで'1'→'0'に変化したとき，以下の順番で実行されます．

* 時刻T 代入処理 ：　A←'0'
* Bに代入する値の評価 ：　'1'
* 時刻T＋Δ 代入処理 ：　B←'1'
* Cに代入する値の評価 ：　'0'
* 時刻T＋2Δ 代入処理 ：　C←'0'
* Zに代入する値の評価 ：　'1'
* 時刻T＋3Δ 代入処理 ：　Z←'1'

　つまり，「評価」と「代入」を交互に行うため，代入する前に値が決まります．このため記述の順番やシミュレータに依存することなく，同一の結果が得られます．

**● 二つの信号代入文**  
　信号に対する代入文には，

* アーキテクチャ直下に記述する同時処理代入文
* プロセス内で記述する順次処理代入文

があります．

　いずれも記述の順番に関係なく正しく実行されますが，デルタ遅延の考えかたから，動作に若干の違いがあります．

**図16**では，いずれも同じ二つの代入文があります．順次処理代入文として記述されているプロセス文では，センシティビティ・リストがAだけとします．同時処理代入文は，すでに解説したように，**図16（a）**のように信号が変化します．

|  |  |
| --- | --- |
| http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f16a1.jpg http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f16a2.gif  （a）　同時処理代入 | http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f16b1.jpg http://www.kumikomi.net/archives/2009/07/ep38vhdl/ep38vhdl_f16b2.gif （b）　順次処理代入 |

**図16　同時信号代入と順次信号代入**

　一方，順次処理代入文では，次のように信号が変化します．初期値を，

　　A='1', B='1', Z='x'

とし，時刻TでAが'1'→'0'に変化したと想定します．

* 時刻T　 代入処理　 ：　A←'0'
* Bに代入する値の評価　 ：　'0'
* Zに代入する値の評価　 ：　'1'
* 時刻T＋Δ　 代入処理　 ：　B←'0'
* 代入処理　 ：　Z←'1'

　評価と代入が同時に行われないため，記述の順序に関係なく，BとZに正しく値が代入されます．