VHDL 簡易マニュアル (v1.1)

1. 全体の書式の概観

```
--  ハイフン2つはコメント

--  使用するライブラリの宣言. 以下 2 行はお約束.
library ieee;
use ieee.std_logic_1164.all;

-- エンティティの宣言（名前と入出力信号線）
entity entity_name is
   port ( port_name_1: [in / out] [std_logic / std_logic_vector( portnum-1 downto 0)] ;
          -- ↑in は入力, out は出力.
          --  std_logic は 1 ビットの信号線. std_logic_vector(x downto 0)は x+1 ビット.
           ...
           port_name_n: [in / out] [std_logic / std_logic_vector( portnum-1 downto 0)]
          -- ↑信号線リストの最後にはセミコロンをつけないこと.
         );
end entity_name

-- アーキテクチャの宣言（内部動作）
architecture arch_name of entity_name is
-- 内部でのみ使用する信号線, コンポーネントはここに書く
   signal sig_name_1 : [std_logic / std_logic_vector( portnum-1 downto 0)] ) ;
     ...
   signal sig_name_n : [std_logic / std_logic_vector( portnum-1 downto 0)] ) ;
     -- ↑内部信号線には入出力の区別はない.
   component component_name
     port(仮信号線リスト);
   end component;
begin

--この部分に動作を記述

end arch_name;

--コンフィグレーション宣言（詳しくは後述）
configuration conf_name of entity_name is
  ...
end conf_name;
```

2. ライブラリ, パッケージ宣言部

以下の 2 行は必ず書く.

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

ベクトル型の信号線(std_logic_vector)同士を符号なし整数とみなして加減算を行う場合, 以下のパッケージを追加する.

```
use ieee.std_logic_unsigned.all;
```

3. エンティティ宣言部

VHDL では，コンポーネントのエンティティ宣言（名前とインタフェース）とアーキテクチャ宣言（内部動作の記述）を厳密に区別する．これにより，複数のアーキテクチャを宣言することが可能である（本実験では扱わない）．コンポーネントの名前を決定するのは entity 文であり，インタフェース（入出力信号線の名前，入力/出力の区別，ビット幅等）を宣言するのが entity 文の中にある port 文である．

例えば，d という名前の 2 ビットの入力，sel という名前の1ビットの入力，および x という名前の 2 ビットの出力を持つ，selector という名前のコンポーネントのエンティティ宣言は以下のようになる．

```
entity selector is
   port ( d: in std_logic_vector(1 downto 0);
          sel: in std_logic;
          x: out std_logic_vector(1 downto 0)
        );
end selector;
```

インタフェース宣言で入力として指定された信号線については，アーキテクチャ宣言部で値の参照は可能であるが，値の代入は不可能である．逆に，出力として指定された信号線については，代入は可能であるが参照はできない．

テストベンチ等，外部入出力信号線を持たないコンポーネントにおいては，port 文を省略可能である．例えば，上記のコンポーネント selector のためのテストベンチ test_selector を作成する場合，エンティティ宣言は以下のように記述する．

```
entity test_selector is
end test_selector;
```

4. アーキテクチャ宣言部

アーキテクチャは，entity 文で宣言されたエンティティの内部動作を定義するものである．前述のとおり，1つのエンティティに対して複数のアーキテクチャを定義可能であるため，各アーキテクチャに名前をつける必要がある．アーキテクチャの名前は，

architecture *arch_name* of *entity_name* is ...

のように，常に特定のエンティティと関連付けて定義される．従って，異なる2つのエンティティが同じ名前のアーキテクチャを持っていたとしても，これらは区別される．

アーキテクチャ宣言部は，内部で使用する信号線やコンポーネントの宣言部と，機能記述部に分けられる．まず，信号線やコンポーネントの宣言について列挙し，その後 begin キーワードに続けて機能を記述する．機能記述については次章で説明する．

信号線については，以下の書式で宣言する．

signal *sig_name* : [std_logic / std_logic_vector(*portnum*-1 downto 0)]) ;

entity 文の中の入出力信号線と異なり，内部信号線には in/out の区別がなく，いずれの信号も代入および参照が可能である．もちろん，内部信号線の値を出力信号線に代入することも可能である．また，内部のみで用いる信号が存在しない場合，内部信号線の宣言を行う必要はない．

VHDL では，別のエンティティとして宣言されたコンポーネントをアーキテクチャ内部で利用できる．これにより，システムの階層的な設計が可能となる．あるコンポーネントを利用する場合，まずそのコンポーネントの名前とインタフェースについて記述する．その後，機能記述部においてそのコンポーネントの実体を作成し，配線を行う．コンポーネントの実体の作成方法については次章で説明する．

例えば，3 章に述べたエンティティ selector をコンポーネントとして利用する場合，コンポーネント宣言は以下のようになる

```
component selector
   port ( d: in std_logic_vector(1 downto 0);
          sel: in std_logic;
```

```
        x: out std_logic(vector(1 downto 0)
      );
  end component;
```
基本的に，エンティティ宣言の entity キーワードと最後のエンティティ名を component キーワードに置き換え，is キーワードを取り除けばよい．ここで，コンポーネント宣言における信号線名(d, sel, x)は仮の名前であり，機能記述部でこれらの信号を利用できるわけではない．

5．機能記述部

エンティティ宣言部の begin キーワードに続けて，内部動作の記述を行う．内部動作記述の主なものは，コンポーネントの実体の生成，同時信号代入文，process 文などである．

5．1 コンポーネントの実体の生成

コンポーネントは，実体を作成することではじめて利用可能となる．例えば，4 章に述べたコンポーネント selector の実体を selector1 という名前で作成する場合，以下のように記述する．

```
  selector1: selector port map (x -> sig_X, d -> sig_D, sel -> sig_SEL);
```

ここで，sig_D, sig_SEL, sig_X は実際の信号線の名前であり，入出力信号線（エンティティ宣言部）あるいは内部信号線（アーキテクチャ宣言部）としてあらかじめ宣言されている必要がある．また，仮の信号線と実際の信号線のビット幅は等しくなければならない．

信号線割当てリスト(port map)において仮の信号線名が省略された場合，コンポーネント宣言で定義された信号線リスト(component 文中の port)と同じ順序で信号線が割当てられる．例えば，上の実体宣言は

```
  selector1: selector port map (sig_D, sig_SEL, sig_X);
```

と記述することも可能である．

同一のコンポーネントから，複数の異なる実体を作成することも可能である．たとえば，コンポーネント selector の実体 selector1 と selector2 を生成する場合，

```
  selector1: selector port map (sig_D1, sig_SEL1, sig_X1);
  selector2: selector port map (sig_D2, sig_SEL2, sig_X2);
```

と記述すればよい．

5．2 値と演算子

信号代入文について説明する前に，信号線のとりうる値と演算子について簡単に説明する．std_logic 型の信号は，0, 1, L, H, Z, X の7値のいずれかをとる．L, H はそれぞれ弱い 0, 1 である．Z はハイインピーダンス，X は不定値を意味する．std_logic 型の信号に対する論理演算としては，and, or, xor, not, nand, nor が定義されている．

std_logic_vector 型は，std_logic 型の信号線からなるベクトルであり，各ビットがとりうる値は std_logic 型と同一である．std_logic_vector 型の信号線は，任意の1ビット，連続するビット，全てのビットに対して代入や参照が可能である．例えば，

```
  signal a: std_logic_vector(3 downto 0);
```

として宣言された4ビットの内部信号 a を考える．このとき，以下のような代入文は全て有効である．

```
  a <= “0001”;
  a(2) <= ‘0’;
  a(3 downto 2) <= “11”;
```

上記のように，1 ビットの定数はシングルクォーテーションで囲い，ベクトル型の定数はダブルクォーテーションで囲う．

std_logic_vector 型においても，and, or などの論理演算が可能である．ただし，演算子の左右の項，および代入文における右辺値と左辺値は全て等しいビット幅でなければならない．この場合に有用なのが，ビット連結演算子&である．例えば，4 ビットの信号 a と2ビットの信号 b がある場合，以下のような記述が可能である．

```
  a <= “00” & b;
  a(2 downto 0) <= b(1) & (b or “11”);
```

パッケージ ieee.std_logic_unsigned や ieee.std_logic_arith を用いることで，std_logic_vector 型同士の加減算(算術演算子+, -)が可能になる．

５．３ 同時信号代入文

アーキテクチャ宣言部の begin 以降に直接記入された信号代入文は，同時信号代入文となる．例えば，

```
architecture arch of entity_test is
begin
  a <= b;
  c <= d or e;
end arch
```

というソースリストにおいて，a <= b; と c <= d or e;がそれぞれ1つの同時信号代入文である．<=は代入記号である．VHDL はハードウェアを記述する言語であるため，多くのソフトウェア記述言語と異なり，信号代入文の実行順序はソースリスト中の位置には依存しない．同時信号代入文は，右辺に存在するいずれかの信号が変化した場合に呼び出され，演算の結果が左辺値に代入される．従って，上のリストにおいて a <= b; と c <= d or e;の順序を入れ換えた場合でも，生成されるハードウェアは全く同じように動作する．

同時信号代入文では，when キーワードを用いることによって，代入される値の条件付けを行うことができる．when キーワードの使用例を以下に示す．

```
a    <=    “0000” when sel = “00”
       else  “1111” when sel = “11”
       else  “1010”;
```

改行と空白は見やすいように入れたものであり，なくてもよい．この記述では，a は sel が 00 の場合に 0000，sel が 11 の場合に 1111，それ以外の場合には 1010 となる．ここで，最後の else 以降が存在しない場合，sel が 00, 11 以外の値になったときには a は変化しない．このような記述は文法的な誤りではないが，順序回路を生成してしまう．when キーワードを用いて 00, 01, 10, 11 の 4 通りについて条件を設定した場合についても，std_logic 型は L, H などの値をとりうるため，最後の else がなければ全ての条件を記述したことにはならない．従ってやはり順序回路を生成してしまう．不必要な順序回路は回路誤動作の原因となるため，注意が必要である．

when キーワード以下の条件式では，以下の関係演算子を利用可能である．

= (等しい)，　/= (等しくない)，　< (小さい)，<= (小さいか等しい)，> (大きい)，>=(大きいか等しい)

５．４ process 文と順序信号代入文

process 文によって，プロセス（回路を構成するオブジェクト）を記述することができる．process 文は，あらかじめ指定された信号線の値が変化すると呼び出され，内部に存在する代入文を上から順番に実行する．従って，process 文はある信号線に対するドライバとなる．1 つの process 文の中に複数の代入文を記述することが可能である．従って process 文は複数の信号線のドライバとなることも可能であるが，このような使用法は誤りの原因となりやすいため注意が必要である．

process 文自体は同時信号代入文と同様に，他の同時信号代入文や process 文と並列に動作する．ソースリスト中における位置には依存しない．

process 文の例を以下に示す．

```
process(b, c) begin
  a <= b xor c;
  d <= not c;
end process;
```

(b, c)の部分はセンシティビティリストと呼ばれ，ここに記述された信号線の値が変化した場合にこの process 文が実行される．a <= b xor c; および d <= not c;はそれぞれ信号代入文である．（この例では a と d という 2 つの異なる信号をドライブしていることになる．あまり良い例ではない．）

注意しなければならないのは，process 文の中に記述された代入文は同時信号代入文ではなく，上から順番に実行されるということである．ただし，信号線へ値を代入する場合，微少な遅延時間(Δ遅延)が発生する．この遅延時間は process 文全体が実行される時間より長いと定義されている．以下のような process 文を考える．

```
process(b, c) begin
  a <= b xor c;
  d <= not a;
end process;
```

このとき，a の値が b xor c になるのは process 文が終了した後であり，d の代入文が実行される時点では a はまだ以前の値を保持している．従って，この例では d は not (b xor c)とはならない．もしセンシティビティリストに a も含まれて

いたならば，1 度 process 文が終了した後，a の変化によってΔ時間後に再度 process 文が呼び出され，そこで d が変化する．

任意の同時信号代入文は，process 文を用いて書き直すことが可能である．しかし，上記のように，process 文は明示的に指定された信号線（センシティビティリストに含まれるもの）が変化しないと呼び出されない．このため，容易に順序回路を生成しうる．process 文を用いて組合せ回路を作成する場合，センシティビティリストや後述する条件文の記述において注意が必要である．

process 文の中では，if などを用いて複雑な条件判断を行うことが可能である．有用と思われるいくつかの構文について簡単に説明する．

if 文

記法

```
if (  条件式) then
    順序信号代入文
[ elsif (条件式) then
    順序信号代入文  ]
[ elsif ...]
[ else
    順序信号代入文  ]
end if;
```

備考

elsif や else は省略可能である．条件を満たさない場合には信号線が変化しないため，順序回路となる．従って，else を省略した場合にはかならず順序回路を生成する．

case 文

記法

```
case  信号線  is
   when  値  =>
      順序信号代入文
   [when  値  => ... ]
   [ when others =>
      順序信号代入文  ]
 end case;
```

備考

同時信号代入文の when キーワードとは使用法が全く異なる．c 言語などと異なり，条件に合致する when キーワードの部分のみが実行される．2 個目以降の when キーワードは省略可能である．when others => は，それ以前の条件に全て合致しない場合に実行される．

wait 文

記法

```
wait for  時間;
wait until  条件式;
wait;
```

備考

指定された時間，あるいは条件が真になるまで，あるいは無条件に process 文の実行をそこで停止する．時間は，例えば 100 ns (100 ナノ秒), 10 us (10マイクロ秒)などのように指定する．process 文に wait 文が存在する場合，センシティビティリストを省略してもよい．wait 文は，テストベンチの記述の際に，適当な信号を代入した後指定時間待つ場合などに非常に有用である．

上の説明で順序信号代入文と書かれた場所には，複数の代入文を記述することが可能である．また，if 文や case 文などをさらに記述し，入れ子構造にすることも可能である．

6． コンフィグレーション宣言

VHDL では複数のアーキテクチャを宣言できるため，シミュレーションや論理合成時にどのアーキテクチャを用いるかを指定する必要がある．使用するアーキテクチャを宣言するのがコンフィグレーション宣言である．階層化設計で

は，少なくとも最上位階層のエンティティには必ずコンフィグレーション宣言が必要である．

コンフィグレーション宣言の記法を以下に示す．
```
configuration コンフィグレーション名 of エンティティ名 is
  for アーキテクチャ名
    for コンポーネントの実体名: コンポーネント名 use entity work.コンポーネント名(アーキテクチャ名);
    end for;
    [for ... ]
  end for;
end コンフィグレーション名;
```
アーキテクチャ名およびコンポーネントの実体名には，all キーワードを用いることも可能である．アーキテクチャ名に all を用いた場合，そのエンティティではアーキテクチャによらず同じ設定が使われる．また，コンポーネントの実体名に all を用いた場合，そのコンポーネントに対しては常に同じアーキテクチャが使われる．本実験では常に all を用いてよい．

7. 諸注意

7．1 クロックエッジに同期した順序回路

本実験では，単相クロックを用い，フリップフロップなどの順序回路は全てクロック信号のポジティブエッジ（立ち上がり）またはネガティブエッジ（立ち下り）に同期して値を変更するとする．信号線の event 属性を用いた条件式を記述することで，このようなエッジの検出が可能である．

クロック信号 clk の立ち上がりに同期して信号線 d の値を q に取り込むようなフリップフロップの記述例を以下に示す．
```
process (clk) begin
  if (clk'event and clk = '1') then
    q <= d;
  end if;
end process;
```
clk'event は，信号線 clk の値が変化した瞬間だけ真となる．従って，clk'event and clk = '1'によって，clk のポジティブエッジを検出できる．

上記のフリップフロップに対し，リセット信号 r による同期リセット機能を付け加えた場合の記述例を以下に示す．
```
process (clk) begin
  if (clk'event and clk = '1') then
    if (r = '1') then
      q <= '0'
    else
      q <= d;
    end if;
  end if;
end process;
```
一方，リセット信号 r によるリセットが非同期リセットの場合，以下のようになる．
```
process (clk, r) begin
  if (r = '1') then
    q <= '0'
  elsif (clk'event and clk = '1') then
    q <= d;
  end if;
end process;
```
同期リセットの場合，信号の変化は常にクロック信号のエッジに同期するため，センシティビティリストには r を書かなくてもよい（r があっても間違いではなく，正しく動作する）．一方，非同期リセットでは，リセットはクロック信号とは独立に行われるため,センシティビティリストに r を記述する必要がある．

クロックエッジを検出しない（event 属性を使用しない）場合，順序回路を生成するとその順序回路はラッチ動作となる．すなわち，クロック信号が 1 である間に他の入力信号が変化した場合，出力が変化する可能性がある．このような記述は誤動作の原因となる．

7．3 識別子

信号線やコンポーネント名，アーキテクチャ名などの識別子としては，英数字とアンダースコア(_)からなる 1 文字以上の文字列を使用できる．ただし，名前の最初は英字でなければならない．また，アンダースコアで終わってはならない．さらに，アンダースコアは 2 個以上連続してはならない．

VHDL にはいくつかの予約語が存在する．これらの予約語に合致する文字列は，上記のルールを満たしていても識別子としては使用できない．

正式な規格では，英字の大文字と小文字は区別されない．ただし，処理系によっては区別するものも存在するので注意が必要である．例えば，ALU というコンポーネントを作成した場合，別のファイルや場所で alu や Alu などと書かない方が良いかもしれない．

7．4 信号線の初期値

内部信号線には，初期値を設定することが可能である．信号線の初期値を設定する記述の例は以下のとおりである．

```
signal a: std_logic_vector(3 downto 0) := “0000”;
```

設定された初期値は，シミュレーションにおいては有効である．従って，所望の動作をするか否かを簡便に判断することが可能となる．しかし，初期値は論理合成を行ってハードウェアを生成する場合には無視される．従って，設定した初期値が存在することを期待して回路を作成してはならない．

プログラムカウンタなど，起動時に特定の値になることが要求される信号線やフリップフロップが存在する場合，外部から適切なリセット信号などを入力することが要求される．

信号線の初期値を設定しなかった場合，シミュレーションでは，特定の値が入力されるまで不定値 X として扱われる．

7．5 文法誤りについて

VHDL は，Pascal や C 言語などのプログラミング言語と記法が大きく異なる．特にこれらのプログラミング言語に慣れた者は，記法の違いからくる文法誤りに注意するように．

よくある間違いとしては，
・entity 文中の port 文において，最後の信号線にもセミコロンをつけてしまった
・end process や end アーキテクチャ名 といったような，end の後に記述するものを間違えた
などが挙げられる．

8． サンプルコード

以下に，7 セグメント LED 用のデコーダ回路と，そのテストベンチの記述例を示す．

7 セグメント LED デコーダ回路 (sevenseg.vhd)

```
-- 7 セグメントデコーダ
--  入力 : a 4bit (ボタンを押すと 0)
--  出力 : b 7bit (0..消灯  1...点灯), c 4bit (各 LED を制御，0 : 点灯，点灯は 1 箇所だけ)
--            -6-
--        1|      |5
--            -0-
--       2 |      |4
--            -3-

library ieee;
use ieee.std_logic_1164.all;

entity sevenseg is
   port ( a: in std_logic_vector(3 downto 0);
```

```
            b: out std_logic_vector(6 downto 0);
            c: out std_logic_vector(3 downto 0) );
end sevenseg;

architecture arch of sevenseg is
   signal d: std_logic(3 downto 0);
begin
   d <= not a;
   b <=       “1111110” when d = “0000”
         else “0110000” when d = “0001”
         else “1101101” when d = “0010”
        -- 途中省略
         else “1111011” when d = “1001”
         else “0000000”;
end arch;
```

7 セグメント LED デコーダ回路のためのテストベンチ (test_sevenseg.vhd)

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

entity test_sevenseg is
end test_sevenseg;

architecture arch of test_sevenseg is
   component sevenseg
      port ( a: in std_logic_vector(3 downto 0);
             b: out std_logic_vector(6 downto 0) ;
             c: out std_logic_vector(3 downto 0));
   end component;
   signal sig_a: std_logic_vector(3 downto 0);
   signal sig_b: std_logic_vector(6 downto 0);
   signal sig_c: std_logic_vector(3 downto 0);
begin
   sevenseg1: sevenseg port map(sig_a, sig_b,sig_c);

   process begin
     sig_a <= “0000”;
     wait for 100 ns;
     sig_a <= “0001”;
     wait for 100 ns;
     -- 途中省略
     sig_a <= “1010”;
     wait;
   end process;

end arch;

configuration conf_sevenseg of test_sevenseg is
   for arch
      for all: sevenseg use entity work.sevenseg(arch);
      end for;
   end for;
end conf_sevenseg;
```