Design Wave BOOKS

VHDLで学ぶ ディジタル回路設計

ディジタル回路の理論と
VHDL設計の基礎を同時に学ぶ

吉田 たけお
尾知 博 共著

第0章

ディジタル回路設計の世界

ディジタル回路は，コンピュータの心臓部としての役割だけでなく，テレビ，ビデオ，オーディオ，携帯電話，テレビゲームなどの一般家電の制御中枢としての役割を担っており，その応用分野は，半導体技術の進歩とともに広がり続けている．このようなディジタル回路は，一体どのように設計されるのであろうか？ ここでは，ディジタル回路設計の世界を概観しよう．

0.1(*) ディジタル回路の設計過程

0.1.1 ディジタル回路設計における設計段階

ディジタル回路は，その設計を開始してから実際の LSI 回路として**実現 (implementation)** されるまでに，**図 0.1** に示すような設計過程を経る．

(1) **方式設計 (system design)**
- ディジタル回路の動作やそれを実現する手順，アーキテクチャなどの**仕様**を決定する．

(2) **機能設計 (functional design)**
- 必要となる構成要素や構成要素間のデータの流れを決定する．

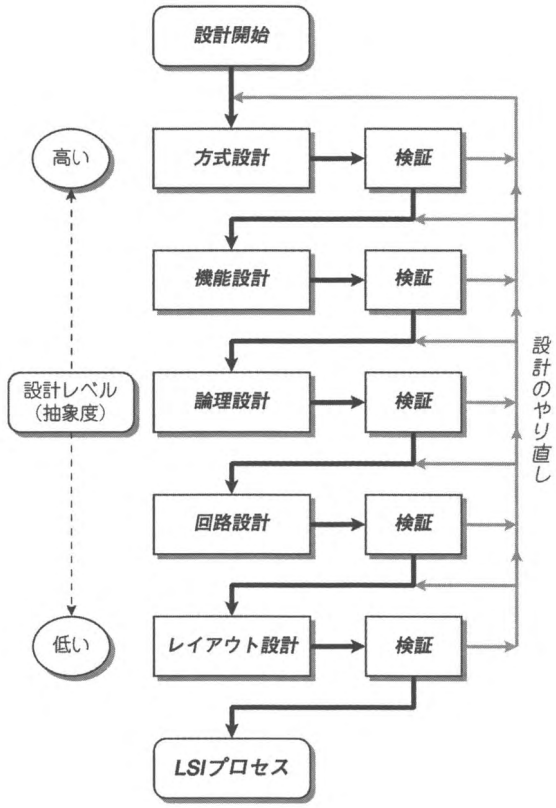
(3) **論理設計 (logic design)**
- ゲート回路などのディジタル素子を用いた回路図を作成する．

(4) **回路設計 (circuit design)**
- トランジスタなどのアナログ素子を用いた回路図を作成する．

(5) **レイアウト設計 (layout design)**
- LSI プロセスにおけるフォトマスク原画となるマスクパターンを作成する．

ディジタル回路の設計は，上記 (1)〜(5) の順に行われ，各段階における設計作業が終了すると，**図 0.1** に示すように，その設計作業が正しく行われたかどうかを確認する．この確認作業を**設計検証 (design verification)** または単に**検証 (verification)** という．検証の結果，設計作業が正しく行われていると判断された場合は，次の設計段階に進む．一方，設計作業が正しく行われていないと判断された場合は，その設計段階または以前

図 0.1　ディジタル回路の設計過程



の設計段階に戻って[注1]，設計作業をやり直す．この設計作業のやり直しは，検証結果が良くなるまで繰り返される．すなわち，ディジタル回路は，設計と検証の繰り返し作業によって実現されていく．

設計の初期の段階では，実際のディジタル回路に対する物理的なイメージが少なく非常に抽象的である．設計が進んでいくと，具体的な回路イメージに近づいていく．ディジタル回路の設計においては，**抽象度**の高い設計段階を**高レベル** (high level) といい，抽象度の低い設計段階を**低レベル** (low level) という．上記の設計過程は，高いレベルから低いレベルへと進んでいくため，**トップダウン設計** (top-down design) と呼ばれる．一方，これとは逆に，低いレベルから高いレベルへと進めていく設計手法もあり，これを**ボトムアップ設計** (bottom-up design) と呼ぶ．

0.1.2　トップダウン設計とボトムアップ設計

トップダウン設計手法は，まずディジタル回路全体の動作や機能を決定し，除々に具体化 (回路化) していく方法である．**トップダウン設計**手法は，ハードウェアの基礎知識が無くても比較的容易に設計作業が行え

注 1：実際の設計現場では，各設計段階が独立しているため，以前の設計に戻ってやり直すことができない．

第1章

2進数とゲート回路

コンピュータやそれを構成するディジタル回路では，各桁が ‘0’ または ‘1’ で表現される 2 進数を用いて数値計算を行っている．理由は，‘0’ と ‘1’ の 2 値を表現できるエレクトロニクス素子が安価で簡単に製作実現できるからである．本章では，こうしたディジタル回路の基本である 2 進数とゲート回路について学び，ディジタル回路の世界の入口を覗いてみることにする．

1.1　10進数と2進数

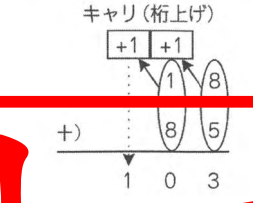
1.1.1　導入

導入演習 1.1 (10 進数の加算と 2 進数の加算)

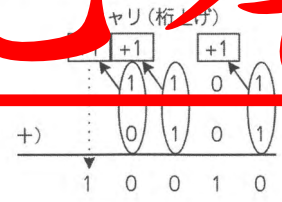
以下に示した **10 進数 (decimal number)** の加算 $(18)_{10} + (85)_{10}$ の計算手順を参考にして，**2 進数 (binary number)** の加算 $(1101)_2 + (0101)_2$ の計算手順を示しなさい．ただし，$(x)_n$ は，x が n 進数であることを表している．

[解]

$$(18)_{10} + (85)_{10} = (103)_{10}$$



$$(1101)_2 + (0101)_2 = (10010)_2$$



□

このように，**2進数**の加算手順は，**10進数**の加算と同様に，最下位桁から順に**キャリ (carry：桁上げ)** を考慮しながら計算していく．ここで，1桁の2進数の加算を**図1.1**の回路ブロックで表現しておくと，導入演習1.1で示したような4桁の2進数の加算を実行する回路は，**図1.2**のブロック図となることが理解できる．

図1.1　2進加算器 (1桁分)

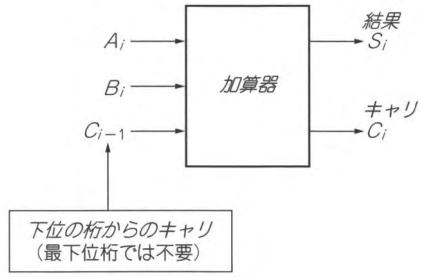
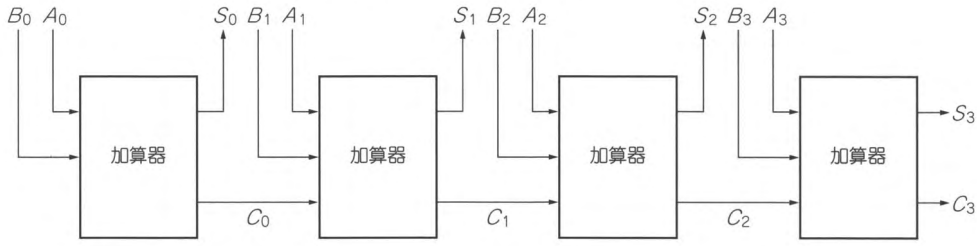


図1.2　2進加算器 (4桁分)



$$(A_3A_2A_1A_0)_2+(B_3B_2B_1B_0)_2=(C_3S_3S_2S_1S_0)_2$$

この演習で考えた2進加算器は，コンピュータで数値計算を行っているCPUに必ず内蔵されている，基本的でかつ重要なディジタル回路である．さてそれでは，こうした2進加算器は，実際にはどのような素子を用いて実現されているのであろうか？ 本書の目的の一つは，こうしたディジタル回路の設計を行うことである．本書を読み進んでいくにつれて，加算器などの実用上重要なディジタル回路の設計や解析が行えるようになる．

1.1.2　2進数の表現

まず始めにディジタル回路で扱われる2進数の表現について検討してみよう．

◆ n進数

m 桁の n 進数 $I=(a_{m-1}a_{m-2}\cdots a_0)_n$ の10進数における値 $V(I)$ は，

$$V(I)=a_{m-1}\times n^{m-1}+a_{m-2}\times n^{m-2}+\cdots+a_0\times n^0 \tag{1.1}$$

$$=\sum_{i=0}^{m-1}a_i n^i \tag{1.2}$$

と表される．なお，n 進数の各桁 a_i，$i = 0, 1, \cdots, m-1$ は，$\{0, 1, \cdots, m-1\}$ のいずれかの値をとる．

◆ 小数の表現

1 未満の小数を **n 進数**で表す場合は，10 進数の場合と同様に小数点を用いて，

$$F = (.a_{-1}a_{-2}\cdots a_{-k})_n \tag{1.3}$$

と表す．各桁 a_{-i} は n^{-i} の重みをもつので，F の 10 進数としての値 $V(F)$ は，

$$V(F) = a_{-1} \times n^{-1} + a_{-2} \times n^{-2} + \cdots + a_{-k} \times n^{-k} \tag{1.4}$$

$$= \sum_{i=1}^{k} a_{-i} n^{-i} \tag{1.5}$$

となる．

◆ 2 進数とその 10 進数としての値

以上より，一般に **2 進数** B は，

$$B = (a_{m-1}a_{m-2}\cdots a_0.a_{-1}a_{-2}\cdots a_{-k})_2 \tag{1.6}$$

で表現され，その 10 進数としての値 $V(B)$ は，式 (1.2)，式 (1.5) より，

$$V(B) = \sum_{i=0}^{m-1} a_i 2^i + \sum_{i=1}^{k} a_{-i} 2^{-i} \tag{1.7}$$

となる．

◆ ビットとバイト

2 進数において各桁を**ビット (bit)** という．また 8 桁の 2 進数，すなわち 8 ビットのかたまりを 1 **バイト (byte)** と呼ぶ．

バイトにはキロ (kilo : K)，メガ (mega : M)，ギガ (giga : G) などの補助単位が付くことが多い．通常，キロは 1,000，メガは 1,000 K，ギガは 1,000 M を表す．しかし，2 進数を扱うコンピュータやディジタル回路の世界では，$2^{10} = 1{,}024$ バイトを 1 キロバイト (Kbyte : KB)，2^{20} バイト = 1,024 キロバイトを 1 メガバイト (Mbyte : MB)，2^{30} バイト = 1,024 メガバイトを 1 ギガバイト (Gbyte : GB) と表すので注意が必要である．

◆ 2 進化 10 進 (BCD) 符号

ディジタル回路では 2 進数を扱うため，我々がよく使用する 10 進数を 2 進数に変換する必要がある．この変換には 10 進数を各桁ごとに 2 進数で表す **2 進化 10 進 (binary coded decimal : BCD) 符号**が，比較的に簡単に変換を行えるために，よく用いられる．1 桁の 10 進数に対する BCD 符号の対応表を**表 1.1** に示す．

例えば，2 桁の 10 進数 $(84)_{10}$ は，(10 進数 84 は) 7 桁の 2 進数で表される．この 10 進数 $(84)_{10}$ を BCD 符号で表す場合，各桁毎に**表 1.1** の対応に従って 2 進化する．すなわち $(84)_{10}$ は，BCD 符号では $(10000100)_{\mathrm{BCD}}$ と 8 桁の 2 進符号として表される．

第7章
VHDLによるディジタル回路設計

前章までで，ディジタル回路に関する内容を一通り終えたので，以下では，VHDL 設計におけるヒントやディジタル回路の実装技術などについて述べておく．また，VHDL を使い始めて半年程度の初心者の記述例を紹介するので，こちらも参考にしていただきたい．

7.1　ディジタル回路の設計方針

ディジタル回路 (論理回路) は，**組み合わせ回路**と**順序回路**とに大別される．ここでは，これらの回路の違いや HDL を用いた回路設計の流れについて述べる．

7.1.1　データパスと制御回路

一般にディジタル回路といった場合，**同期式順序回路**を指すことが多い．この同期式順序回路は，**図 7.1** (次頁) に示すように，**データパス (data path)** と**制御回路 (controller)** から構成される．データパスは，算術演算，論理演算などのデータ処理を中心に行う回路であり，データパスの入出力を，それぞれ**データ入力 (data input)**，**データ出力 (data output)** と呼ぶ．また制御回路は，データパスを制御するための回路であり，制御回路の入出力を，それぞれ**制御入力 (control input)**，**制御出力 (control output)** と呼ぶ．

データパスの設計では，まず，必要な**フリップフロップ (FF)** や**レジスタ**を配置し，次に FF やレジスタ間でのデータ処理を行う組み合わせ回路を設計する，という手順が踏まれる．

制御回路は，データパス内の FF やレジスタへの制御信号 (リセット信号やイネーブル信号など) を生成する．この制御回路は，通常，**ステートマシン**[注1]として設計される．

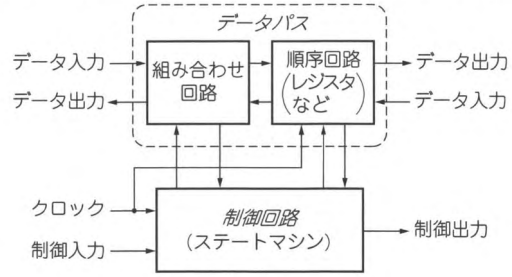
すなわち，ディジタル回路の設計には，データパスとしての同期式順序回路と組み合わせ回路の設計および制御回路としてのステートマシンの設計が含まれていることになる．

7.1.2　組み合わせ回路と順序回路の違い

ディジタル回路の多くは，組み合わせ回路としても順序回路としても設計可能である．実際に本書では，次章で RSA 暗号器と呼ばれる回路を両方の場合について設計する．ここでは，ディジタル回路を組み合わせ回路として設計する場合と順序回路として設計する場合との違いについて述べる．

注 1：ステートマシンは，学術的には順序回路と同義である．しかし，実際の設計現場では，データパスを制御するための回路をステートマシンと呼ぶことが多いので，本書でも順序回路とステートマシンを使い分けることにする．

図 7.1　同期式順序回路のモデル (データパスと制御回路)



◆ 回路規模の違い

たとえば，乗算器を考えてみよう．乗算器を順序回路として実現する場合，通常，被乗数をシフトレジスタに格納しておき，シフトしながら加算を行うという方法をとる．この場合，32 ビット乗算器を順序回路として実現すると，4,000 ゲート程度に収まる．一方，組み合わせ回路として実現した場合，10,000 ゲート程度の回路規模になる．

なお，上記のゲート数は正確な値ではない．設計の仕方によって，これらの数値は大幅に異なるので，あくまでも参考値にとどめていただきたい．また，もともと規模の小さい回路の場合，上記のようにはならない．たとえば，8 ビット乗算器を組み合わせ回路として実現した場合，500 ゲート程度になる．これに対して，順序回路として実現すると，700 ゲート程度になってしまう．

◆ 処理時間の違い

上記のような 32 ビット乗算器を順序回路として実現した場合，計算が終了するまでに，32 クロック (被乗数の桁数) 分の処理時間を必要とする．一方，組み合わせ回路として実現した場合，1 クロック (組み合わせ回路の遅延時間) 分の処理時間で計算が終了する[注2]．

以上のように，ディジタル回路を組み合わせ回路として実現すると，一般に回路規模は大きくなるが，出力が得られるまでの処理時間は格段に短くなる．もともと規模の小さい回路や演算器などの高速処理を行わせたい回路などは，できる限り組み合わせ回路として設計するのが望ましい．

ところで，次章で設計する RSA 暗号器では，暗号の安全性を考慮した場合，後述するように 1024 ビットから 2048 ビット程度の乗算器が必要となる．このような大きな乗算器を組み合わせ回路として設計すると，数百万ゲートを越える規模になってしまい，非現実的である．このような場合は，乗算器を順序回路として設計する必要がある．

7.1.3　ステートマシンを設計する目的

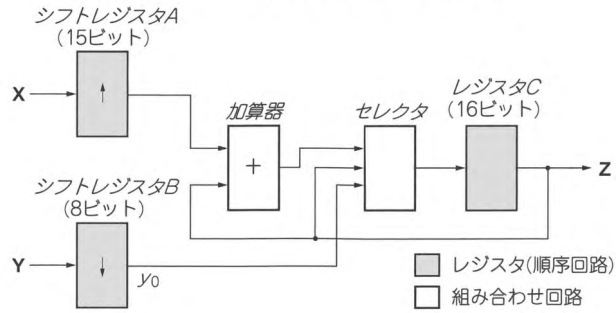
前述のように，**ステートマシン**は，データパスの制御回路として用いられる．ここでは，8 ビット乗算器を例にして，ステートマシンを設計する目的を説明する．

◆ 8 ビット乗算器のデータパス

いま，$X = (x_7, x_6, \cdots, x_0), Y = (y_7, y_6, \cdots, y_0)$ とし，$Z = X \times Y = (z_{15}, z_{14}, \cdots, z_0)$ を計算する 8 ビット乗算器を順序回路として実現することを考える．8 ビット乗算器を順序回路として実現するには，被乗数 X を

注 2：クロックの周期や組み合わせ回路の遅延時間を考慮していないので，順序回路の方が 32 倍の時間を必要とするとは言えない．

図 7.2 8 ビット乗算器のデータパスの例



MSB 側にシフトしながら加えていけばよい．具体的には，**図 7.2** のような回路構成にすればよい．

図 7.2 では，被乗数 X を MSB 側にシフトさせるシフトレジスタ A，乗数 Y を LSB 側にシフトさせるシフトレジスタ B および，計算結果を保持するレジスタ C の計 3 個のレジスタを用いている．加算器およびセレクタは組み合わせ回路として実現する．

セレクタは，$y_0 = 1$ のときに加算器の出力を選択し，$y_0 = 0$ のときはレジスタ C の出力を選択する．すなわち，シフトレジスタ A により，被乗数 X の 2 倍の値を次々と生成し，シフトレジスタ B の LSB が 1 のときに，その値を加算していく．この過程を乗数の桁数回繰り返すことにより，乗算の機能を実現する．

◆ 8 ビット乗算器の制御回路

しかし，**図 7.2** に示す回路は，乗算器として不完全である．なぜならば，加算を行う回数を制御する機構がこの回路には無いためである．この制御機構を実現するために，ステートマシンが用いられる．

ステートマシンの設計においては，「**状態**」の概念が必要となる．そこで，**図 7.2** のデータパスに行わせたい動作を，順をおって見てみよう．まず計算を始める前に，各レジスタを初期化 (リセット) する必要がある．次に，被乗数および乗数を読み込み，実際の計算を開始する．乗算の計算は，乗数の桁数回の加算を行うことによって終了する．以上の一連の動作は，**状態遷移図**を用いると明確に表現できる．これを**図 7.3** (p.160) に示す．

このように状態遷移図は，データパスを制御するための一連の動作を表すのに適している．また既に説明したように，状態遷移図をもとにディジタル回路を設計する方法も確立されている．以上のような理由によって，データパスの制御回路としてステートマシンが用いられている．

なお参考のために，8 ビット乗算器の順序回路としての VHDL 記述をリスト 7.1 (次頁) に示す．また，**リスト 7.1** の各 process の役割を表 7.1 (p.160) に示す．**リスト 7.1**，**図 7.2**，**図 7.3** をよく見て，**リスト 7.1** が乗算器になっていることを確認してほしい．

7.1.4 HDL によるディジタル回路設計の流れ

ここでは，**HDL** を用いたディジタル回路設計の流れについて説明する．

HDL を用いたディジタル回路の設計には，

- 設計期間を短縮できる
- 過去の設計資産を再利用しやすくなる

リスト 7.1　乗算器の順序回路としての VHDL 記述

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity MULTIPLIER is
    generic( L : integer := 8 );
    port ( CLK, RESET, START : in  std_logic;
           X : in  std_logic_vector(L-1 downto 0);
           Y : in  std_logic_vector(L-1 downto 0);
           Z : out std_logic_vector(2*L-1 downto 0));
end MULTIPLIER;

architecture SYNC_SEQ of MULTIPLIER is

type STATE is (INIT, OP_MUL);
signal CRST, NTST : STATE;
signal SET_MUL : std_logic;
signal S_X : std_logic_vector(2*L-2 downto 0);
signal S_Y : std_logic_vector(L-1 downto 0);
signal S_ADD, S_SEL, S_MUL : std_logic_vector(2*L-1 downto 0);
signal C_MUL : integer range 0 to L+1;
constant ZV_X : std_logic_vector(L-2 downto 0) := (others => '0');

begin
    P_CONTROL_REG: process ( CLK, RESET ) begin
        if ( RESET = '1' ) then
            CRST <= INIT;
        elsif ( CLK'event and CLK = '1' ) then
            CRST <= NTST;
        end if;
    end process;

    P_CONTROL_CNT: process ( CLK ) begin
        if ( CLK'event and CLK = '1' ) then
            if (CRST = INIT) then
                C_MUL <= 0;
            elsif ( CRST = OP_MUL ) then
                C_MUL <= C_MUL + 1;
            end if;
        end if;
    end process;

    P_CONTROL_STATE: process ( CRST, C_MUL, START ) begin
        case CRST is
            when INIT => if ( START = '1' ) then
                             SET_MUL <= '1';
                             NTST <= OP_MUL;
                         else
                             SET_MUL <= '0';
                             NTST <= INIT;
                         end if;
```

Appendix A
VHDLの文法概要

本書では，初心者向けの文法書としても使用できるように，VHDL の主要な構文を解説する．VHDL には，本書で解説する構文以外にも多くの構文がある．それらの構文の詳細は他の文献に譲るが，本書で解説する構文を理解しておけば，VHDL の知識としてはとりあえず十分である．

A.1 VHDLの記述方法

まず VHDL の記述方法について述べておく．

VHDL 記述は，**図 A.1 (a)** (次頁) に示すように，`library` 宣言，`entity` 宣言，`architecture` 宣言，`configuration` 宣言から構成される．このうち，`entity` 宣言，`architecture` 宣言[注1]は，全階層において記述する必要があり，`library` 宣言，`configuration` 宣言は，必要に応じて記述すればよい．なお本文で述べたように，`configuration` 宣言は，最上位階層では必ず記述する．

なお，パッケージを記述する場合，その VHDL 記述は，**図 A.1 (b)** に示すように，`library` 宣言，`package` 宣言，`package_body` 文から構成される．

A.2 VHDLの構文解説

本書の構文解説は，以下のような構成になっている．

(1) 構文

VHDL の構文を示す．なお構文の表記は，以下の規則に従っている．

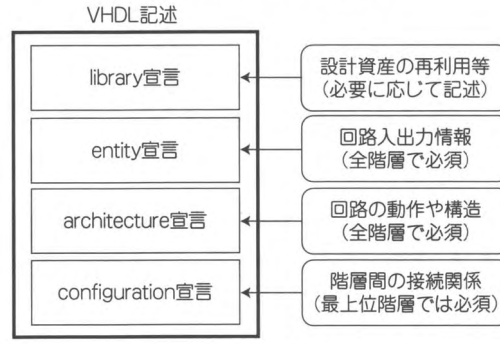
(a) **予約語 (キーワード)** と識別子について

予約語: VHDL において，予め用途の定められている文字列を予約語 (キーワード) という．VHDL では大文字と小文字を区別しないが，予約語と**識別子**を区別するために，本書では小文字を用いて予約語を記述する．

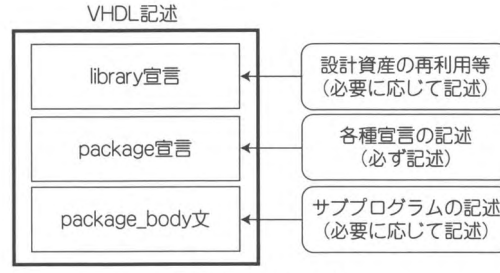
識別子: 回路名，信号線名など，設計者が定める必要のある文字列を識別子という．識別子には，英数字とアンダースコア '_' を用いることができる．ただし，識別子の最初の文字に数字を用いることや，識別子の最後の文字にアンダースコアを用いることはできない．また，アンダースコアを連続して用いることもできない．なお，予約語と識別子を区別するために，本書では大文字を用いて識別子を記述する．

注1：ライブラリに格納しておくこともできる．

図 A.1　VHDL 記述の構造



(**a**) 回路の記述



(**b**) パッケージの記述

(b) 省略記号について

[ ]: 省略可能な部分を“[”と“]”で括る．ただし，“[”と“]”で括られた部分を記述する場合は，たかだか一つだけ記述可能である．

{ }: 省略可能な部分を“{”と“}”で括る．ただし，“{”と“}”でられた部分を記述する場合は，複数 (回) 記述できる．

(2) 機能解説

構文の機能に関する簡単な説明文を示す．

(3) 使用例

構文の使用例とその解説を示す．

● 構文解説 1（architecture 宣言）

[構文]

```
architecture アーキテクチャ名 of エンティティ名 is
  { 宣言文 }
begin
  { 同時処理文 }
end [ アーキテクチャ名 ];
```

[機能解説]

- architecture 宣言は，エンティティに対する一つのアーキテクチャを宣言するための構文である．VHDL では，一つのエンティティに対して，複数のアーキテクチャを持たせることができる．**アーキテクチャ名**は，それらのアーキテクチャを区別するための識別子である．
- アーキテクチャ名の付け方に対する制限は特にないが，どの設計段階に対応する記述なのか，また，構造の記述なのか，動作の記述なのかがわかるような識別子を用いることが望ましい．
- architecture 宣言の宣言文を記述する箇所を **architecture 宣言部**，同時処理文を記述する箇所を **architecture 本体**という．architecture 宣言部には，signal 宣言，constant 宣言，type 宣言，subprogram 宣言，component 宣言などを記述する．

[使用例]

```
architecture STRUCTURE of FULL_ADDER is
begin
  { 同時処理文 }
end STRUCTURE;
```

この使用例では，“FULL_ADDER” という名前 (エンティティ名) の回路を “STRUCTURE” というアーキテクチャ名で設計することを宣言している． □

● 構文解説 2（assert 文）

[構文]

```
assert 条件 [ report "出力メッセージ" ] [ severity エラー・レベル ];
```

[機能解説]

- assert 文は，シミュレーション時に，メッセージを表示させるための構文であり，論理合成後の回路構造には影響を与えない．
- assert 文は，条件が成り立たなかった場合に，report 文で指定した**出力メッセージ**および **severity 文**で指定したエラー・レベルを表示する．**エラー・レベル**を指定する場合，あらかじめ列挙タイプとして定義されている “**NOTE, WARNING, ERROR, FAILURE**” のいずれかを指定する必要がある．
- assert 文は，architecture 本体，entity 宣言，process 文，block 文，subprogram 本体などに記述することができる．

[使用例]

```
assert A > B report "Not valid signals" severity WARNING;
```

この使用例では，シミュレーション時に assert 文の箇所で，その条件 A > B が成り立たなかった

ISBN978-4-7898-5295-1
C3055 ￥3400E
CQ出版社
9784789852951

VHDLで学ぶ ディジタル回路設計