23 電子オルゴールを作る

『はじめての FPGA 設計』の読者のための総まとめとして，譜面データを読み込んで，自動演奏する電子オルゴールを設計してみましょう。譜面をプログラムと読み替えれば，電子オルゴールの動作はマイクロプロセッサ（俗にマイコンと呼ばれているもの）と似たような動作をしているのです。つまり，ここではプロセッサの基本的な動作をする回路記述について紹介しています。

23–1 電子オルゴールの仕様

これから設計を進めていこうとしている**電子オルゴール**の回路構成を，**図 23-1** に示します。楽譜の情報をデータとして与えると，それを順に取り出して解読し，指定されている音程の音を，指定されている長さだけ発生させるというものです。想定している主な仕様を**表 23-1** に示します。

音程の範囲が **3 オクターブ**とは，ドレミファソラシド（これが **1 オクターブ**）が低い，中くらい，高いの 3 段階の範囲ということです。**長さ（継続時間）の範囲**とは，音符と休符の両方に関するもので，4 分音符の長さを 1 とすると，1/4 から 4 倍までの範囲ということです。テンポは「歩くような早さで」という「Andante」

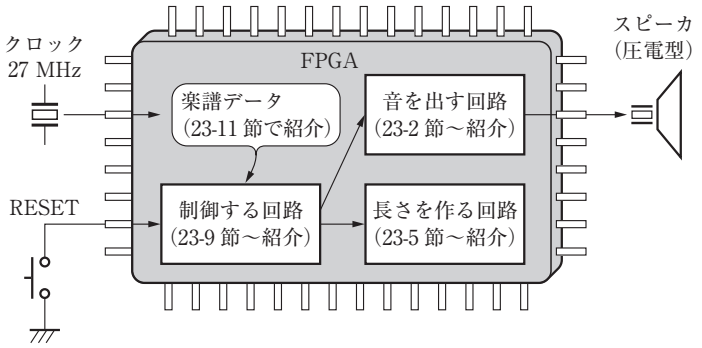


図 23-1 電子オルゴールの全体回路構成

表 23-1 電子オルゴールの主な仕様

項　目	説　明
音程の範囲	3 オクターブ
長さの範囲	16 分音符～全音符（休符も同じ）
テンポ	1 秒間に 4 分音符が 90 演奏される程度（Andante）
演奏できる曲の長さ	音符と休符の指定個数の合計が 256 以下
出力音の波形	デューティ 50％の方形波

としました。曲の長さは，譜面データを記憶しておくための記憶領域の大きさであり，今回は 256 以下としました（使用する FPGA の容量次第で増減できます）。出力音の波形が**方形波**とは，ディジタル回路で'1' と'0' を交互に出力することによって容易に発生できる波形です。**デューティ 50%**とは，出力音の波形の 1 サイクルを構成している'1' と'0' の長さが，同じ（50%ずつ）という意味です。以上，このように決めた深い根拠は特にありません。

23–2 音を出す回路の構成

音の波形として方形波を採用することにしましたが，'1' と'0' を高速に変化させた信号を，スピーカーに接続すると，音として聞こえます。その変化の高速さの度合いによって，音の高低（周波数）が変わります。つまり，ドレミファ…という音程になるのです。

「方形波には奇数次高調波が多く含まれているので，そう単純ではない！」と偉い先生方から怒られそうですが，逆に方形波をきっちりした音として出力することも至難の業なのです。結局は，今回の実験回路のような場合には，経験上から適当に高調波成分が減衰してしまって，程よい音程として聴こえるようになるのです。詳細については，コラム 14 に記述しました。

その音程の決め方として，現代では**平均率 12 音階**を用いるのが一般的です。ピアノなどのけん盤で，下のドから上のドまでで間（1 オクターブ）に，白と黒のけん盤が合わせて計 12 個（両方のドを入れると 13 個）になります。平均率 12 音階では，上のドが下のドの 2 倍の周波数になるように，つまり各音程間の周波数比を $2^{1/12}$（約 1.059）倍と決めた方式なのです。

オーケストラのチューニング（指揮者が入場してくる前の音合わせ）では，弦楽器の開放弦（指で押さえないで張ったままの弦）に共通している A 音（ハ長調のラ）

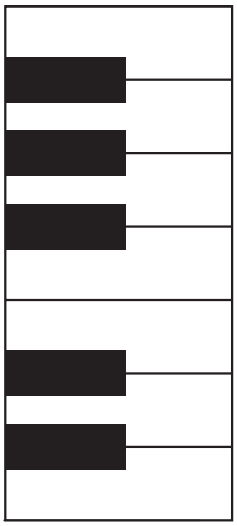


表 23-2 各音程の周波数とタイマー設定値

音　程	周波数〔Hz〕	タイマー設定値	周波数〔Hz〕	タイマー設定値	周波数〔Hz〕	タイマー設定値
シ	988	13 664	1 976	6 832	3 952	3 417
（黒鍵）	932	14 486	1 865	7 243	3 728	3 621
ラ	880	15 341	1 760	7 670	3 520	3 835
（黒鍵）	831	16 245	1 662	8 123	3 324	4 061
ソ	784	17 219	1 568	8 610	3 136	4 305
（黒鍵）	740	18 243	1 480	9 122	2 960	4 561
ファ	698	19 341	1 396	9 671	2 792	4 835
ミ	659	20 485	1 318	10 243	2 636	5 121
（黒鍵）	622	21 705	1 244	10 853	2 488	5 426
レ	587	22 998	1 174	11 499	2 348	5 750
（黒鍵）	554	24 369	1 108	12 185	2 216	6 092
ド	523	25 813	1 046	12 907	2 092	6 453

が用いられています。また，NHK ラジオの時報の最後の音，つまり，ポッ，ポッ，ポッ，ピーン！のピーンの音の周波数が，880Hz です。そして，この周波数こそが，A 音なのです。ちなみに，ポッの音は 440Hz（ピーンのラに対して 1 オクターブ下のラ）です。

さて，今回は下のラの周波数を 880Hz とし，このようにして求めた平均率 12 音階の各音程の周波数を**表 23-2** に示します。また，今回の音を出す回路は，**図 23-2** のような構成とし，具体的なクロック周波数 27MHz から必要とする各音程を作り出すために，タイマーに設定する値を，表 23-2 に対応させて示します。

この際，デューティを 50％とするのにトグル回路を使用しているため，ON と OFF の 2 回の動作によって 1 サイクル分の波形が発生することになります。よって，タイマーへ設定する値は，**発生させる周波数の 2 倍**になります。

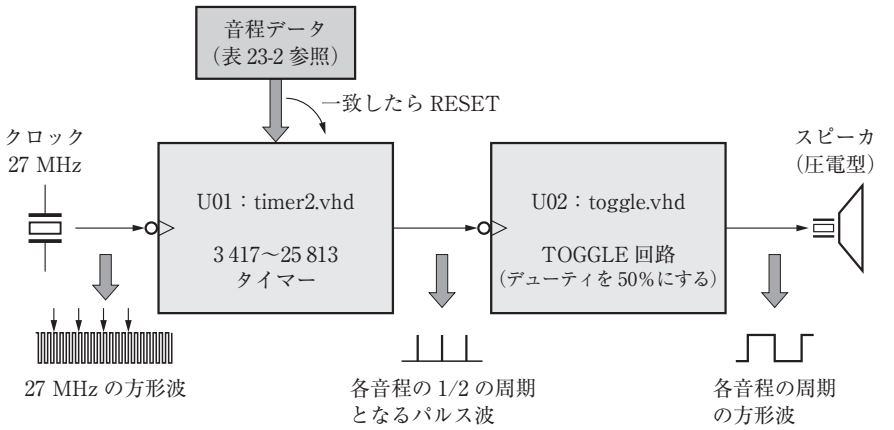


図 23-2　音を出す回路の模式図（波形はイメージ）

23–3　指定した音程の音を出す回路の VHDL 記述

音を出す回路の VHDL 記述例を**リスト 23-1** に示します。図 23-2 で紹介したように，タイマー回路とトグル回路を組み合わせただけです。ただし，今回のタイマー回路は，**リスト 23-2** に示すように，17-4 節のリスト 17-3 に紹介した汎用タイマーに，ロード機能を追加したものとなっています。さらに詳しくいえば，リスト 17-3 がアップカウンタ型であったのに対して，今回のはダウンカウンタ型となっています。このロード機能によって，音程データ（表 23-2 の設定値）をセットします。また，generic 文で設定しているタイマーの長さ CYCLE には，設定値の最大数まで収められるように，25 813 としました。

リスト 23-1 に示した音を出す回路をシミュレーションするためのテストベンチ記述例は，**リスト 23-3** のようになります。まず，RESET をかけた後，小さな設定値（3 と 5）を順にロードし，相当した方形波が出力されるかを確認するという内容です。実行結果を**図 23-3** に示します。

RESET = '0' で CNT が 0 となり，RESET = '1' でリセット解除となった後で，LD = '0' のタイミングで CNT に DATA がロードされています。これにより，CNT

リスト 23-1　音を出す回路の VHDL 記述例（sound.vhd）

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

entity SOUND is
port( CLK:      in  std_logic;
      INTERVAL: in  integer range 0 to 25813; -- ontei
      LD:       in  std_logic;
      RESET:    in  std_logic;
      SP:       out std_logic );
end SOUND;

architecture RTL of SOUND is
component TIMER2
    generic( CYCLE: integer :=13500000 ); -- 0.5s at 27MHz
    port( CLK:   in   std_logic;
          RESET: in   std_logic;
          POUT:  out  std_logic;
          LD:    in   std_logic;
          DATA:  in   integer range 0 to CYCLE-1 );
end component;

component TOGGLE
     port( SW0:   in   std_logic;
           LED0:  out  std_logic );
end component;

signal POUT: std_logic;

begin
U01: TIMER2 generic map(  CYCLE=>25813 )
            port map( CLK=>CLK, RESET=>RESET,
                      POUT=>POUT, LD=>LD, DATA=>INTERVAL );

U02: TOGGLE port map( SW0=>POUT, LED0=>SP );

end RTL;
```

リスト 23-2　今回使用するタイマー回路の VHDL 記述例（timer2.vhd）

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

entity TIMER2 is
    generic( CYCLE: integer :=13500000); -- 0.5s at 27MHz
    port( CLK:   in   std_logic;
          RESET: in   std_logic;
          POUT:  out  std_logic;
          LD:    in   std_logic;
          DATA:  in   integer range 0 to CYCLE-1 );
end TIMER2;

architecture RTL of TIMER2 is
signal CNT: integer range 0 to CYCLE-1:=0;

begin
    process( CLK, RESET, LD ) begin
        if( RESET = '0' ) then CNT <= 0;
        elsif( LD = '0' ) then CNT <= DATA;
        elsif( CLK'event and CLK='0') then
            if CNT=0 then CNT <= DATA;  POUT <= '1';
            else          CNT <= CNT-1; POUT <= '0';
            end if;
        end if;
    end process;
end RTL;
```

リスト23-3　リスト 23-1 をシミュレーションする VHDL 記述例（t_sound.vhd）

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

entity T_SOUND is
end T_SOUND;

architecture RTL of T_SOUND is
component SOUND
port( CLK:      in  std_logic;
      INTERVAL: in  integer range 0 to 25813; -- ontei
      LD:       in  std_logic;
      RESET:    in  std_logic;
      SP:       out std_logic );
end component;

signal CLK, LD, RESET, SP: std_logic;
signal INTERVAL:  integer  range 0 to 25813;
constant PERIOD:  TIME:= 37 ns; -- 1/27MHz

begin
U01:SOUND port map( CLK=>CLK, INTERVAL=>INTERVAL,
                    LD=>LD, RESET=>RESET, SP=>SP );

    process    begin
        CLK <= '0'; wait for PERIOD/2;
        CLK <= '1'; wait for PERIOD/2;
    end process;

    process  begin
        RESET <= '0'; LD <= '1';  wait for PERIOD/2;
        RESET <= '1';             wait for PERIOD;
        INTERVAL <= 3; LD <= '0'; wait for PERIOD;
        LD <= '1';                wait for PERIOD*20;
        INTERVAL <= 5; LD <= '0'; wait for PERIOD;
        LD <= '1';                wait for PERIOD*20;

        assert false;
        report "Simulation Complete !!"
        severity Failure ;
    end process;
end RTL;
```

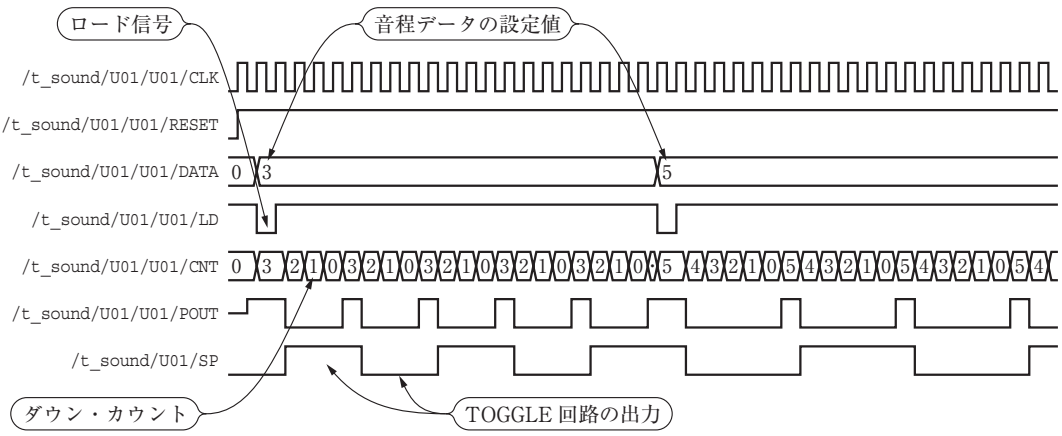


図 23-3　リスト 23-1 のシミュレーション結果

が 3 ～ 0 までカウントダウンするタイマーとなっていることがわかります。CNT が 0 となったとき，POUT が 1 クロック期間分だけ'1' となり，そのタイミングで TOGGLE 回路の出力である SP が反転しています。

23–4 実際に音を出してみよう！

以上のように，希望通りの動作が確認できましたので，実際に音を出してみることにしましょう。その前に，音を出すためのスピーカですが，**圧電素子**（**ピエゾ素子**）のものを採用しました。これは，電子機器などで操作音や警報音を出すのに多く用いられているもので，磁石とコイルを用いた通常のスピーカーに比べて，扱いが簡単です。単にディジタル回路の出力に直接接続するだけでよいのです。特別な駆動回路を必要としません。

音が出ることを確認するデモ用 VHDL 記述を，**リスト 23-4** に示します。表 23-2

リスト 23-4 音階の出ることを確認する VHDL 記述例（demo_sound.vhd）

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

entity DEMO_SOUND is
    port( CLK:   in  std_logic;
          RESET: in  std_logic;
          LED0:  out std_logic;
          LED1:  out std_logic;
          SP:    out std_logic  );
end DEMO_SOUND;

architecture RTL of DEMO_SOUND is
component SOUND
port( CLK:      in  std_logic;
      INTERVAL: in  integer range 0 to 25813;
      LD:       in  std_logic;
      RESET:    in  std_logic;
      SP:       out std_logic );
end component;

type ROMARRAY is array(0 to 7) of integer range 0 to 25813;
constant SCORE: ROMARRAY := ( 12907, 11499, 10243, 9671, 8610, 7670, 6832, 6453 );
signal POINT:    integer range 0 to 7;
signal CNT:      integer range 0 to 13500000; -- 0.5 sec
signal INTERVAL: integer range 0 to 25813;
signal LD: std_logic;

begin
U01: SOUND port map( CLK=>CLK, INTERVAL=>INTERVAL, LD=>LD, RESET=>RESET, SP=>SP );

INTERVAL <= SCORE( POINT );
    process( CLK, RESET ) begin
        if(RESET = '0' ) then POINT <= 0; CNT <= 0; LD <= '1';
        elsif( CLK'event and CLK='0') then
            if ( CNT = 13499999 ) then POINT <= POINT + 1; LD <= '0'; CNT <=0;
            else    CNT <= CNT+1; LD <= '1';
            end if;
        end if;
    end process;
end RTL;
```

の中くらいの 1 オクターブの範囲の音を，0.5 秒間隔で順に出し続けるという内容になっています。ここで，音のデータを記憶させるのに，**array**（**配列**）を用いています。そして，POINTER の値を 0 から順に＋1 し，配列の何番目のデータをセットするのかを決めています。

さて，どんな音が聞こえてくるのでしょうか？　是非とも，この感動を実機で体験していただきたいです。

23–5　長さ時間を作る回路の構成

ここでは，音の継続時間や休止している時間の長さを作り出す回路を設計しましょう。全体の回路構成は**図 23-4** のようになります。一見して，前述した図 23-2 の音を出す回路とよく似ています。

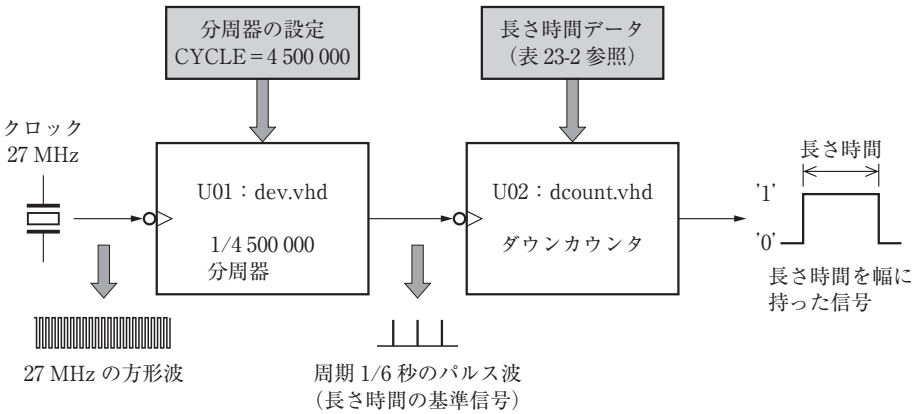


図 23-4　長さ時間を作る回路の模式図（波形はイメージ）

扱える**長さ時間の範囲**としては，音符でいえば，一番短い 16 分音符から，一番長い全音符までを可能としています（**表 23-3**）。また，絶対時間との関係は，音楽のルールに従って 4 分音符をテンポの基準とし，1 分間に 90 回鳴らすことのできる長さ時間（Andante という）を採用しました。ほかの長さ時間は，その 1/2 や 1/4，2 倍や 4 倍となります。

ここまできて，**長さ時間**と**速さ**で混乱していませんか？　両者は逆数の関係にあ

表 23-3　主な音符と休符の種類と長さ

音　符									
休　符									
長　さ	1/4	3/8	1/2	3/4	1	1.5	2	2.5	4
長さ時間データ	1	今回設定不可	2	3	4	6	8	12	16

ります。つまり，長さ時間が 1/2 なら，速さは 2 倍になるというわけです。速さというより，周波数といったほうがわかりやすいかもしれませんね。

まずは分周器で 27MHz のクロック信号から，Andante を実現する長さ時間の基準信号（ダウンカウンタ用のクロック）を分周器で作ります。そして次のダウンカウンタでは，この基準信号を何回数えたら必要となる長さの時間となるのかを指定します。以上によって，希望する長さ時間の幅を持つ出力信号が得られることになります。以下に具体的に説明していきます。

23–6 長さ時間データの基準は 16 分音符

具体的な長さ時間データの基準としては，ここで扱う範囲で最速の 16 分音符の長さ時間データとし，ほかの長さ時間データはこれの何倍（何回数える）かということで指定することにします。16 分音符は，4 分音符の 4 倍の速さ（長さ時間では 1/4 の短さ）なので，1 秒間の回数は 90×4/60 = 6 となります。つまり周期 1/6 秒の基準パルス信号を 27 MHz のクロックから作り出すことになります。ということは，27 000 000/6 = 4 500 000 となり，クロック信号を 4 500 000 回数えたら，1 クロック幅のパルスを 1 回出力する回路とすればよいのです。

このような機能の回路を**分周器**といいます。本書の，リスト 17-3（17-4 節参照）として紹介した汎用タイマー（timer.vhd）を用いて，リセット機能を追加した**リスト 23-5**（dev.vhd）を汎用分周器として作成しました。そして，dev.vhd を部品として使用する際には，CYCLE = 4 500 000 と設定すれば，今回の求める機能の回路となります。

リスト 23-5 汎用分周回路（dev.vhd）

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

entity DEV is
    generic( CYCLE: integer :=13500000);  -- 0.5S at 27MHz
    port( CLK:   in   std_logic;
          PULSE: out  std_logic;
          RESET: in   std_logic );
end DEV;

architecture RTL of DEV is
signal CNT: integer range 0 to CYCLE-1;

begin
    process( CLK, RESET ) begin
        if( RESET = '0' )  then CNT<=0;
        elsif( CLK'event and CLK='0') then
            if CNT=CYCLE-1 then CNT<=0; PULSE <= '1';
            else CNT <= CNT + 1;        PULSE<= '0';
            end if;
        end if;
    end process;
end RTL;
```

このように，分周器に generic を用いて汎用化したメリットとして，今回の電子オルゴールに関しては CYCLE に設定する値を変えるだけで，テンポを自由に変更することができます。大きな値にすれば遅くなり（たとえば Largo：ゆるやかに），小さくすれば速く（たとえば Allegro：快速に）なります。

しかし，この方法では，値を変更するたびに，コンパイルし直す必要があります。そこで，ロード機能を追加し，テンポを設定するための命令を加えれば，演奏の途中で任意にテンポを変えることも可能となります。挑戦してみてください。

このように，考えがどんどん広がって機能が追加されたり，仕様が変更となっても，FPGA なら容易に対応可能なところが楽しいですね！

23–7 長さ時間を '1' の幅で表現

さて次に，それぞれの音符の長さ時間（基準信号である 16 分音符の何倍か）を指定して，相当する幅の信号を出力する回路を設計しましょう。これにはリスト 23-2 として紹介した timer2.vhd をちょっと修正して，**リスト 23-6** のようなものとしました。ロード機能で指定された長さ時間データを取り込み，カウントダウンするカウンタです。

ここで，出力信号がパルスではなく，カウントダウン中は'1' を出力し続ける信号となっているのがミソです。この長さ時間信号が'1' の期間中，同じ音が出続けることになり，'0' になったタイミングで次の楽譜データの取り込みを行うことにします。このことについては，23-8 節で詳しく紹介します。

リスト 23-6 ダウンカウンタ回路（dcount.vhd）

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

entity DCOUNT is
port( CLK:   in  std_logic;
      PLS:   out std_logic;
      LD:    in  std_logic;
      DATA:  in  std_logic_vector(3 downto 0) );
end DCOUNT;

architecture RTL of DCOUNT is
signal CNT: std_logic_vector(3 downto 0);

begin
    process( CLK, LD )
    begin
        if( LD='0' ) then CNT<=DATA; PLS<='1';
        elsif( CLK'event and CLK='0' ) then
            if( CNT="0001" ) then    PLS<='0';
            else CNT<=CNT-1;
            end if;
        end if;
    end process;
end RTL;
```

この回路に加えるCLK信号としては，リスト23-5に示した汎用分周回路の出力信号PULSEを用いることはいうまでもありません。それでは，この回路の動きをシミュレータで確認してみましょう。**リスト23-7**がそのためのテストベンチ記述例で，長さ時間としては1，2，4を順に設定しています。

図23-5がシミュレーション結果です。DATAに長さ時間データを代入し，ロード信号LDをほんの一瞬だけ'0'にすると，ダウンカウンタCNTへDATAの値が移るとともに，長さ時間を幅に持った信号PLSが'1'となります。その後は，6 Hzの基準信号によりCNTの値がカウントダウンされ，0となった期間満了時点の基準信号の立下りエッジで，PLSが'0'となっているようすが確認できます。

リスト23-7　リスト23-6をシミュレーションするVHDL記述（t_dcount.vhd）

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

entity T_DCOUNT is
end T_DCOUNT;

architecture RTL of T_DCOUNT is

component DCOUNT
port( CLK:   in  std_logic;
      PLS:   out std_logic;
      LD:    in  std_logic;
      DATA:  in  std_logic_vector(3 downto 0) );
end component;

signal CLK, LD: std_logic;
signal PLS: std_logic:='0';
signal DATA:  std_logic_vector(3 downto 0);
constant PERIOD: TIME:= 167 ms; -- 1/6 sec
constant CLOCK:  TIME:=  37 ns; -- 1/27 MHz


begin
U01:DCOUNT port map( CLK=>CLK, PLS=>PLS, LD=>LD, DATA=>DATA );

   process begin
       CLK <= '0'; wait for PERIOD/2;
       CLK <= '1'; wait for PERIOD/2;
   end process;

   process begin
       DATA <= "0010"; LD <= '0'; wait for CLOCK;
       LD <= '1'; wait for PERIOD*10;

       DATA <= "0100"; LD <= '0'; wait for CLOCK;
       LD <= '1'; wait for PERIOD*10;

       DATA <= "0110"; LD <= '0'; wait for CLOCK;
       LD <= '1'; wait for PERIOD*10;

       DATA <= "1000"; LD <= '0'; wait for CLOCK;
       LD <= '1'; wait for PERIOD*10;

       assert false;
       report "Simulation Complete !!" severity Failure ;
   end process;
end RTL;
```

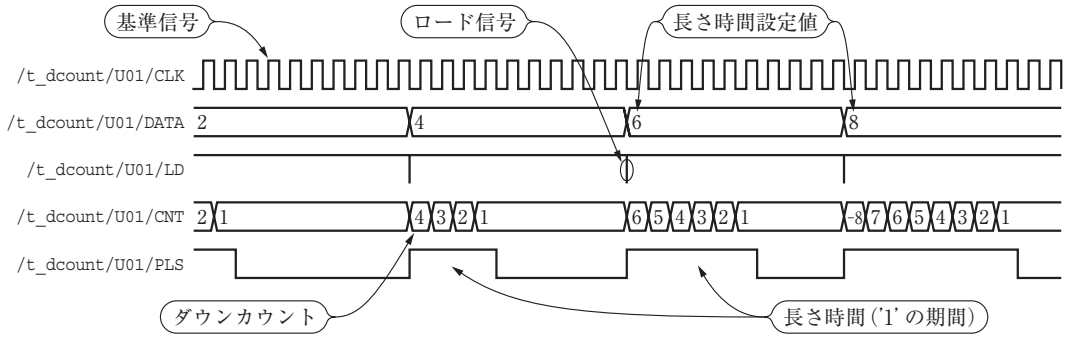


図 23-5　リスト 23-6 のシミュレーション結果

23–8　長さ時間回路の VHDL 記述

23-7 節で紹介した 2 つの回路（dev.vhd と dcount.vhd）を組み合わせることによって，図 23-4 に示した長さ時間を作る回路が構成できます。**リスト 23-8** に，VHDL 記述例を示しますが，単に分周器 U01 の出力信号 PULSE を，次段のダウンカウンタ U02 の入力信号 CLK へ接続しただけです。

リスト 23-8　長さ時間を作る回路（length.vhd）

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

entity LENGTH is
port( CLK:   in  std_logic;
      LD:    in  std_logic;
      DATA:  in  std_logic_vector(3 downto 0);
      RESET: in  std_logic;
      TEMPO: out std_logic );
end LENGTH;

architecture RTL of LENGTH is

component DEV
    generic( CYCLE: integer :=13500000);  -- 0.5S at 27MHz
    port( CLK:   in   std_logic;
          PULSE: out  std_logic;
          RESET: in   std_logic );
end component;

component DCOUNT
port( CLK:   in  std_logic;
      PLS:   out std_logic;
      LD:    in  std_logic;
      DATA:  in  std_logic_vector(3 downto 0) );
end component;

signal PULSE: std_logic;
begin
    U01: DEV generic map( CYCLE=>4500000 )
             port map( CLK=>CLK, PULSE=>PULSE, RESET=>RESET );

    U02: DCOUNT port map( CLK=>PULSE, PLS=>TEMPO,
                          LD=>LD, DATA=>DATA );
end RTL;
```

ここを接続する

この回路の動作をLEDの点灯によって確認するためのVHDL記述例を，**リスト23-9**に示します。この場合には，DATA = "0100"とし，4分音符を連続して指定しています。ここで，長さ時間を作る回路（length.vhd）の出力TEMPOの1音ごとの区切りは，たったの1/6秒のため，出力信号TEMPO（長さ時間を幅に持った信号）をそのままLED0に接続したのでは，各音ごとの境目を確認することが，極めて困難と思われます。そこで，toggle回路（リスト15-1（toggle.vhd）（15-2節参照））を用いて，LEDをON/OFF動作させることにしました。その結果，4分音符の長さ時間の2倍の周期でLEDが点滅を繰り返すことになります。つまり，1分間に45回点滅すればOKです。

リスト23-9　リスト23-8の動作確認をするVHDL記述（demo_length.vhd）

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

entity DEMO_LENGTH is
    port( CLK:   in  std_logic;
          LED0:  out std_logic );
end DEMO_LENGTH;

architecture RTL of DEMO_LENGTH is
component LENGTH
port( CLK:   in  std_logic;
      LD:    in  std_logic;
      DATA:  in  std_logic_vector(3 downto 0);
      RESET: in  std_logic;
      TEMPO: out std_logic );
end component;

component TOGGLE
    port( SW0:   in   std_logic;
          LED0:  out  std_logic );
end component;

signal DATA: std_logic_vector( 3 downto 0 );
signal H, LD, TEMPO: std_logic;

begin
H<='1'; DATA<="0100";          ← 4分音符の長さ時間を指定
U01: LENGTH port map( CLK=>CLK, LD=>LD, DATA=>DATA, RESET=> LD, TEMPO=>TEMPO );
U02: TOGGLE port map( SW0=>TEMPO, LED0=>LED0 );

   process( CLK, TEMPO )  begin
       if( TEMPO='0' )               then LD<='0';   ← TEMPOが'0'になったら次の長さ時間をロードする
       elsif( CLK'event and CLK='0') then LD<='1';
       end if;
   end process;
end RTL;
```

23–9　ステートマシンとしてまとめる

あらためて今回設計開発する電子オルゴールとは「楽譜の情報をデータとして与えると，順に取り出して解読し，指定されている通りに演奏する」というものです。実は，このような動作は，フォン・ノイマン氏が提案した**ストアド・プログラム方**

式というコンピュータの基本動作と同じなのです。

その動作とは，

①命令の取り出し（フェッチ）

②命令の解読（デコード）

③命令の実行（エグゼキュート）

という処理の繰り返しです。今回の場合，楽譜がプログラムで，音符や休符が命令に相当します。つまり，楽譜というプログラムにしたがって，メロディを演奏する専用コンピュータを設計することになります。

このように，全体の動作をいくつかの状態に分け，それらの状態間を何らかの条件によって移り変わる（遷移する）ことによって，目的とする機能を実現させるという考え方を**ステートマシン**と呼んでいます。一般に，これらを表現するのに，状態遷移図が用いられることについては，すでに第 20 章に紹介しました。ちなみに，電子オルゴール用の自動演奏プロセッサの動作を状態遷移図風に表現すると**図 23-6** のようになります。

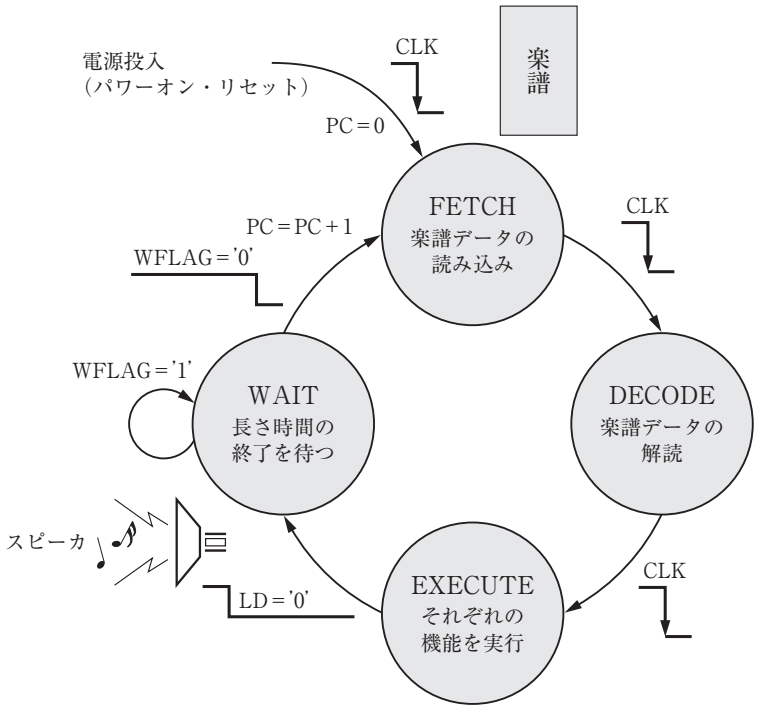


図 23-6　自動演奏プロセッサの状態遷移

ステートマシンを設計するときには，列挙型を採用するのが慣例となっているのです。今回の自動演奏プロセッサの動作は，図 23-6 に示したように 4 つのステートに分けることができることから，**図 23-7** のような type 宣言となります。MY_STATE が列挙型の型名で FETCH，DECODE，EXECUTE，WAITS の 4 つの値のみを取る型ということです。この型名によって定義された signal 信号 MODE を用いて，実際の記述を行うことになります。

図 23-7　自動演奏プロセッサでの列挙型の使用例

23-10　命令の種類と構成

それでは，プログラムを作るために必要となる命令のフォーマットを決めることにしましょう。一般に命令は，命令そのものの識別コード（INST）と，その命令を実行するときに必要となるデータ（オペランドという：OP）によって構成されています。今回は，**音符命令**と**休符命令**と**繰返し命令（最初へ戻る）**の 3 種を考えます。

音符命令の場合には，「音程」と「長さ時間」の 2 つのオペランドが必要となります。休符命令は「長さ時間」だけです。繰返し命令は，最初に戻るだけなのでオペランドはなしです。このように，命令にもいろいろなタイプがあります。よって，それらをすべて満足させるように，命令全体の長さ（ビット数）と，その中の分割の仕方を検討することになります。

今回の場合，命令数は少ない（たったの 3 つ）ので，INST は 16 通りも区別できればよいので 4 ビット（多すぎる！）を割り当てます。データとしては 2 種類が必要で，1 つ目の OP1 は長さ時間用で，**表 23-4** に示すようなコードを格納するために 4 ビットを割り当てました。この値は，直接カウンタに設定して使用することにしたため，今回は設定できない休符が出てしまいました。

表 23-4　長さ時間の指定方法（長さ時間データ）

音　符									
休　符									
長さ時間データ	1	今回設定不可	2	3	4	6	8	12	今回設定不可

2 つ目の OP2 は音符とし，**表 23-5** に示すようにオクターブの低中高の区別と音階の区別が必要なため，16 進表記した場合の 1 桁ずつとなるように，4 ビットずつの計 1 バイトとしました。以上をまとめて並べると，**図 23-8** に示すような 2 バイト長の構成となりました。

表 23-5　音程の指定方法（音程データ）

	音程データ		
音　程	低　音	中　音	高　音
シ	0B	1B	2B
（黒鍵）	0A	1A	2A
ラ	09	19	29
（黒鍵）	08	18	28
ソ	07	17	27
（黒鍵）	06	16	26
ファ	05	15	25
ミ	04	14	24
（黒鍵）	03	13	23
レ	02	12	22
（黒鍵）	01	11	21
ド	00	10	20

使用できる主な命令

命　令	コード	データ 1	データ 2
音符命令	0001	長さ	音程
休符命令	0010	長さ	—
繰返命令	1000	—	—
停止命令	1111	—	—

命令のフォーマット（2 進数表記）

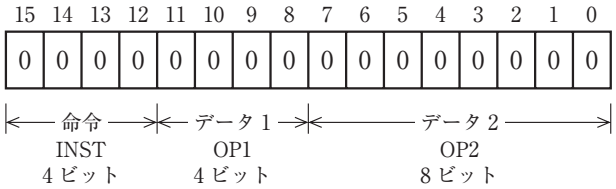


図 23-8　使用できる主な命令とそのフォーマット

23-11　楽譜データの作り方

コンピュータのプログラムに相当する楽譜データの作り方を，**図 23-9** に示します。具体的な楽譜から，命令に落とし込んだデータは，**リスト 23-10** のように VHDL ファイルとして記述します。楽譜データは，わかりやすいように 16 進表記（データの前に x を付ける）で記述しました。

ここで，楽譜データ記憶用として，1 要素が 2 バイト長の命令を 256 個扱えるような配列を宣言しています。したがって，音符と休符を 256 個以下で表せる曲を記憶させて演奏させることができます。

読み出す場所を指定する入力信号 ADDR は，std_logic_vector 型の 8 ビットデータのため，**配列型の romdata の引数（添え字）**としてはそのまま使用することができません。そのため，**conv_integer** によって **integer**（**整数**）に型変換を行っ

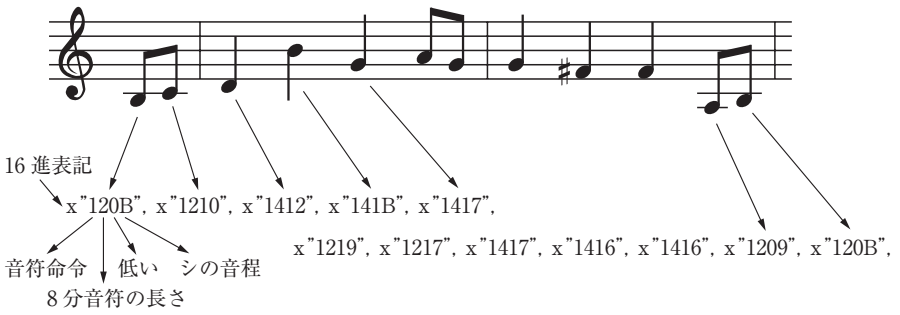


図 23-9　楽譜データの作り方（リスト 23-10 の初めの部分）

リスト 23-10　楽譜データの VHDL 記述例（score_rom.vhd）

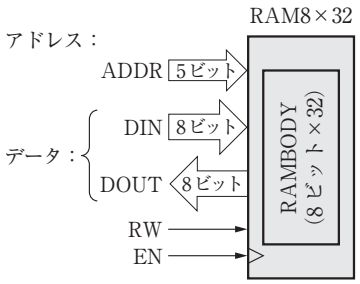
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;          ← Conv_integer を使用するために必要

entity SCORE_ROM is
  port( ADDR: in  std_logic_vector( 7 downto 0 );
        DATA: out std_logic_vector( 15 downto 0 ) );
end SCORE_ROM;

architecture RTL of SCORE_ROM is          ← 16 ビットデータが 256 個集まった配列
  type romarray is array(0 to 255) of std_logic_vector( 15 downto 0 );
  constant romdata: romarray := (          -- it's a small world!
        x"120B",x"1210",x"1412",x"141B",x"1417",x"1219",x"1217",x"1417",x"1416",x"1416",
        x"1209",x"120B",x"1410",x"1419",x"1416",x"1217",x"1216",x"1414",x"1412",x"1412",
        x"120B",x"1210",x"1412",x"1217",x"1219",x"141B",x"1219",x"1217",x"1414",
        x"1219",x"121B",x"1420",x"121B",x"1219",x"1412",x"1420",x"141B",x"1419",x"1817",x"1417",x"2500",
        x"1617",x"1217",x"141B",x"1417",x"1619",x"1219",x"1619",x"2200",
        x"1619",x"1219",x"1420",x"1419",x"161B",x"121B",x"161B",x"2200",
        x"161B",x"121B",x"1422",x"141B",x"1620",x"1220",x"1420",
        x"121B",x"1219",x"1412",x"1420",x"141B",x"1419",x"1817",x"1417",x"8000",
        others => x"0000" );                          ← 繰返し命令
begin
                                          ← std_logic_vector から integer への変換
    DATA <= romdata( conv_integer(ADDR));

end RTL;
```

ています。

余談になりますが，楽譜データはあらかじめ定数として記述しておくもので，実行中に書き換わることはありません。リスト 23-10 においても，constant として 16 進表記の命令を記述しています。これは，コンピュータの構成要素の ROM（読出し専用メモリ）に相当する機能です。

それに対して，読み書き可能なメモリとして RAM があり，その場合には signal の array として記憶場所を確保することになります。**リスト 23-11** に 8 ビット×32 個を記憶できる RAM の参考記述例を示します。ROM の場合と同様に，読み書きする場所を指定する入力信号 ADDR を用いて，配列 RAMBODY の引数（添え字）としています。RW で読み書きの区別（'1' で読出し '0' で書込み）をし，EN の立上りエッジで処理されます。書き込むデータは DIN に代入しておき，読み出したデータは DOUT にセットされてきます。

リスト 23-11　RAM の VHDL 記述例（ram8x32.vhd）




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

entity RAM8X32 is
    port( ADDR    : in  std_logic_vector(4 downto 0);
          DIN     : in  std_logic_vector(7 downto 0);
          DOUT    : out std_logic_vector(7 downto 0);
          RW      : in  std_logic;       --1:read, 0:write
          EN      : in  std_logic );
end RAM8X32;

architecture RTL of RAM8X32 is
type RAM_TYPE is array (0 to 31) of std_logic_vector(7 downto 0);
signal RAMBODY : RAM_TYPE;
signal ADR : integer range 0 to 31;

begin
    ADR <= conv_integer( ADDR );

    process( EN )
    begin
        if( EN'event and EN='1') then
                  if(RW='1') then DOUT <= RAMBODY( ADR );
            else            RAMBODY( ADR ) <= DIN;
            end if;
        end if;
    end process;
end;
```

23-12　自動演奏プロセッサの詳細動作

電子オルゴールの全体像を**図 23-10** に示します。そして，電子オルゴールを制御している自動演奏プロセッサの VHDL 記述例を**リスト 23-12** に示します。これまでに紹介したいろいろな VHDL 記述例が，部品として利用されていることがわかるでしょう。それでは，順を追って処理の流れを説明します。

ほとんどの電子回路では電源投入時に**パワーオン・リセット**という機能が働くようになっていて，必ず RESET 動作から始まります。その結果，STATE_MACHINE：process 内のリセット処理として，

①状態を表す MODE を FETCH にする

②楽譜から楽譜データを取り出す位置を指し示す PC を 0 にする

などの値が初期設定されることになります。

ここで，

PC <=（others => '0'）;

という記述中に登場する「**others**」は，これまでの「そのほかすべての値」といった意味合いではなく，「（何ビットで構成されているのかわかりませんが）PC という信号の全ビットを '0' で埋め尽くす」という役割です。generic 文などによって，要素数の変更される配列など，ビット数を特定できない場合に対する初期化などに用いると便利です。

パワーオン・リセットによって PC の値が '0' に設定されると，この直後に

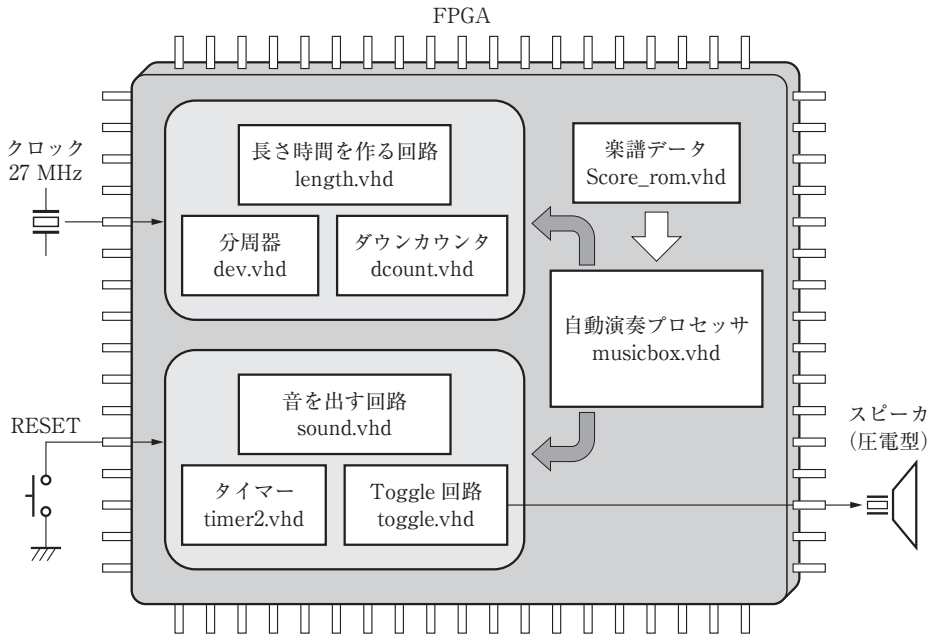


図 23-10　電子オルゴールの全体回路構成

リスト 23-12　自動演奏プロセッサーの VHDL 記述例（musicbox.vhd）

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

entity MUSICBOX is
port( CLK:   in   std_logic;
      RESET: in std_logic;
      SP:     out  std_logic );
end MUSICBOX;

architecture RTL of MUSICBOX is

component SOUND
port( CLK:      in  std_logic;
      INTERVAL: in  integer range 0 to 25813; -- ontei
      LD:       in  std_logic;
      RESET:    in  std_logic;
      SP:       out std_logic );
end component;

component LENGTH
port( CLK:   in  std_logic;
      LD:    in  std_logic;
      DATA:  in  std_logic_vector(3 downto 0);
      RESET: in  std_logic;
      TEMPO: out std_logic );
end component;

component SCORE_ROM
   port( ADDR: in  std_logic_vector( 7 downto 0 );
         DATA: out std_logic_vector( 15 downto 0 ) );
end component;

type    MY_STATE is   ( FETCH, DECODE, EXECUTE, WAITS );
signal  MODE:    MY_STATE;

type VALARRAY is array( 0 to 47 ) of integer range 0 to 25831;
constant INTVAL: VALARRAY := (
```

```
--  Do     --     Re     --     Mi     Fa     --     So     --     Ra     --     Si     Do
  25813, 24369, 22998, 21705, 20485, 19341, 18243, 17219, 16245, 15341, 14486, 13664, 12907, 0, 0, 0,
  12907, 12185, 11499, 10853, 10243,  9671,  9122,  8610,  8123,  7670,  7243,  6832,  6453, 0, 0, 0,
   6453,  6092,  5750,  5426,  5121,  4835,  4561,  4305,  4061,  3835,  3621,  3417,  3417, 0, 0, 0 );
```

表 23-2 の音程データ（タイマー設定値）

```
signal PC:    std_logic_vector(  7 downto 0 ); -- Program Counter
signal IR:    std_logic_vector( 15 downto 0 ); -- Instruction Register
signal ROMOUT: std_logic_vector( 15 downto 0 );
signal INST:  std_logic_vector(  3 downto 0 ); -- Instruction
signal OP1:   std_logic_vector(  3 downto 0 ); -- Operend 1
signal OP2:   std_logic_vector(  7 downto 0 ); -- Operand 2
signal VAL:   integer range 0 to 25813;        -- ontei
signal LEN:   std_logic_vector(  3 downto 0 ); -- nagasa
signal WFLAG: std_logic;                       -- nagasa seigyo
signal LD:    std_logic;                       -- OP latch timing

begin
U01: SOUND     port map(CLK=>CLK, INTERVAL=>VAL, LD=>LD, RESET=>RESET, SP=>SP);
U02: LENGTH    port map(CLK=>CLK, LD=>LD, DATA=>LEN, RESET=>LD, TEMPO=>WFLAG);
U03: SCORE_ROM port map(ADDR=>PC, DATA=>ROMOUT);

STATE_MACHINE: process( RESET, CLK ) is
begin
    if( RESET = '0' ) then MODE <= FETCH; PC <= (others=>'0');
    elsif( CLK'event and CLK='0' ) then
       case MODE is
           when FETCH   => MODE <= DECODE;
           when DECODE  => MODE <= EXECUTE;
           when EXECUTE => MODE <= WAITS;
           when WAITS  =>
               if ( INST = "1000" ) then PC <= (others=>'0'); MODE <= FETCH; --Loop
               elsif( WFLAG = '0' ) then PC <= PC + 1;        MODE <= FETCH;
               end if;
       end case;
    end if;
end process;

FETCH_state: process( RESET, CLK ) is
begin
    if( RESET = '0' ) then IR <=(others=>'0');
    elsif( CLK'event and CLK = '0' ) then
        if( MODE = FETCH )then IR <= ROMOUT;
        end if;
    end if;
end process;

DECODE_state: process( RESET, CLK ) is
begin
    if( RESET = '0' ) then INST <= "0000";
    elsif( CLK'event and CLK = '0' ) then
        if( MODE = DECODE )then
            INST <= IR( 15 downto 12 );
            OP1  <= IR( 11 downto  8 );
            OP2  <= IR(  7 downto  0 );
        end if;
    end if;
end process;

EXECUTE_state: process( RESET, CLK ) is
begin
    if( RESET = '0' ) then LD <= '1';
    elsif( CLK'event and CLK = '0') then LD <= '1';
        if( MODE = EXECUTE )then
            case INST is
                WHEN "0000" => null;                  --NOP
                WHEN "0001" =>                        --NOTE
                    LEN <= OP1; VAL <= INTVAL( conv_integer( OP2 )); LD <= '0';
                WHEN "0010" =>                        --REST
                    LEN <= OP1; VAL <= 0; LD <= '0';
                WHEN "1000" => null;                  --LOOP
                WHEN "1111" => null;                  --HALT
                WHEN others => null;
            end case;
        end if;
    end if;
end process;
end RTL;
```

U03：SCORE_ROM が PC の値が指し示す場所に記憶されている楽譜データを，ROMOUT に出力します。

少しして RESET 信号が'1'（リセット解除）になると，クロック信号の立下りエッジで，MODE が FETCH へ進みます。すると，次のクロック信号の立下りエッジで，FETCH_state：process が動き出し，U03：SCORE_ROM が出力した ROMOUT を命令レジスタ IR にセットします。

次のクロック信号の立下りエッジで，MODE が DECODE へ進み，今度は DECODE_state：process が IR を命令（INST）と 2 種のデータ（OP1，OP2）に分解します。そして次のクロック信号の立下りエッジで，MODE が EXECUTE へと進み，いよいよ命令ごとの処理が行われます。

DECODE_state：process では，解読できる命令として，音符命令（0001）と休符命令（0010）があります。null（ヌルまたはナルと読む：何もしないという意味）となっている命令がいくつかありますが，これらの機能は現段階では未実装ということです。ここでは LOOP 命令（楽譜の最初に戻って繰り返す）に関して null となっていますが，STATE_MACHINE：process の中で処理しています。

さて，INST が"0001"となっている音符命令では，まず OP1 に代入されている長さ時間データを，LEN に代入しています。また OP2 に代入されている std_logic_vector 型の音符データを integer 型に変換し，その値を使ってカウンタへの設定データを取り出し，VAL へ代入しています。この辺りのやり方は**テーブル変換**という定石です。

一方の INST が"0010"となっている休符命令では，OP1 に代入されている長さ時間データを LEN に代入するだけで，VAL には 0（音を出さない）を代入しておきます。

音符命令や休符命令では，それぞれの命令を実行するために必要となる値を決められた信号に代入した後で，ロード信号 LD をアクティブ（有効＝'0'）にしています。これによって，音を出す回路（sound.vhd）や長さを作る回路（length.vhd）が動作を開始します。ここまでで，やっと 1 音目が演奏されることになります。

その後，MODE が WAITS へと進み，STATE_MACHINE：process の中で WFLAG を監視して，長さ時間の終了を待つことになります。WFLAG が'0'になると，PC をインクリメント（＋1 すること）し，MODE を FETCH に戻しています。以上で一連の動作が終了しました。あとは，この繰り返しとなります。

さて，SCORE_ROM.VHD（リスト 23-11）に記述されている楽譜データでは，どんな曲を演奏したのでしょうか？ 実装した人のみが体感できる感動としておきます（リストのどこかに少しコメントしてありますので探してみてください）。

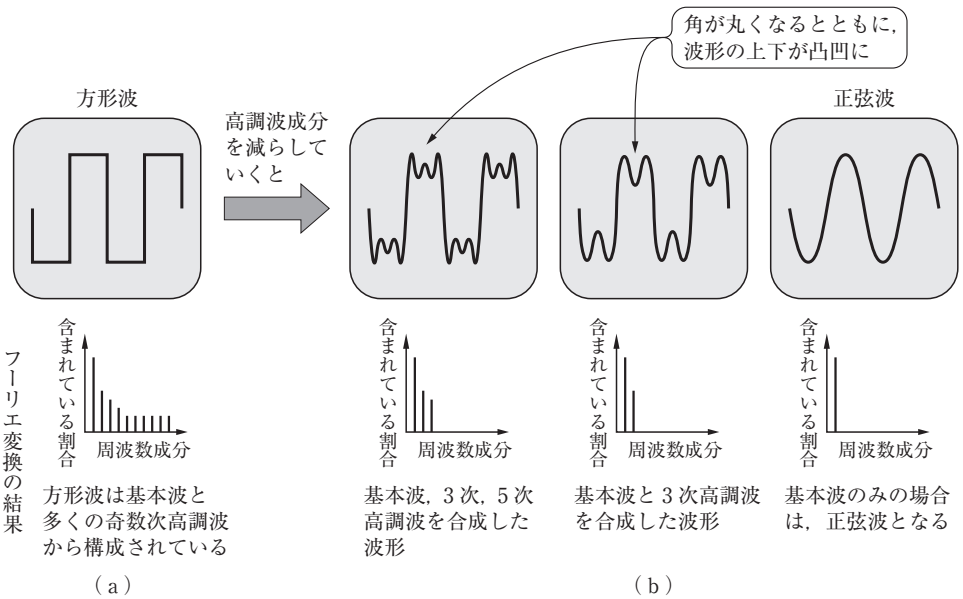
コラム 14　正弦波と方形波

フランスの数学・物理学者ジョゼフ・フーリエ（Jean Baptiste Joseph Fourier：1768～1830）は，「任意の周期関数は，正弦波（SIN 波）と余弦波（COS 波）の級数和で表すことができる」という，有名なフーリエ級数の原理を発表しました。一見して不規則に見える波形でも，周期性のある波形（繰返し波形）であれば，フーリエ級数の原理に基づいて，その周期の周波数である基本波周波数と，その整数倍の高調波周波数の正弦波（と余弦波）のそれぞれの成分の含まれている割合を求めることを**フーリエ変換**といいます。フーリエ変換をコンピュータで処理しやすいように考えられたアルゴリズムとして，**FFT（Fast Fourier Transform：高速フーリエ変換）**がよく知られています。

さて，方形波は周期性のある波形のため，フーリエ変換することができます。その結果，方形波には方形波と同じ周期の正弦波を基本波周波数として，その奇数次高調波（周波数が奇数倍の正弦波）によって構成されていることがわかります。奇数次の高調波のみというのが特徴です（**コラム _ 図 14-1** (a)）。

そこで，方形波を構成している高調波を高い周波数成分の側から順に減らしていくと，波形の角が丸くなるとともに，波形の上下の凸凹が大きくなってきます。そして，基本波のみになると，通常の正弦波になってしまいます（コラム _ 図 14-1 (b)）。

きちんと角ばった方形波を保つためには，相当高い奇数次高調波成分までが必須となります。このことは，ディジタル回路の主役的な存在である方形波信号の取り扱いに際して，高調波成分が減衰しないように十分注意する必要があるということです。



コラム _ 図 14-1　方形波は正弦波の集合体