情報工学実験 **II**

第 4 回： 電子ピアノの実装

2019 年 1 月 10 日

柴田 裕一郎（shibata@cis.nagasaki-u.ac.jp）

1　音を出してみよう♪

今まで設計した回路では，その出力を LED に表示して動作を確認してきました．今回は少し趣向を変えて，音を出すハードウェアを実装してみましょう．

音を出すというと，今までのように LED を光らせるのとはずいぶん様子が異なると思うかもしれません．しかし，ディジタルシステムでは，音も光も所詮 0 と 1 で表されるデータとして扱いますから，基本的な考え方は同じなのです．実験ボードには，8 ビットのディジタルデータ（非負 10 進数で考えれば 0 から 255 までの値）をアナログの電圧値に変換する DAC（Digital Analog Converter）が付いています．FPGA を使って時間とともに周期的に変化する 8 ビットのデータを出力すると，これが DAC によって，時間とともに周期的に変化する電圧値の波へと変換されます．これを増幅してスピーカーを接続すれば空気の振動の波になります．すなわち音が聴こえるのです．

それでは早速やってみましょう．「とにかく音を鳴らす」ための回路の記述をソースコード 1 に示します．この回路には 8 ビットの入力信号 `sw` と 8 ビットの出力信号 `dout` があります．入力には実験ボード上の 8 つの押しボタンスイッチが接続されています．一番左のボタンが 0 ビット目（`sw[0]`），一番右のボタンが 7 ビット目（`sw[7]`）に繋がっていますが，とりあえずこの回路では一番左のボタンだけを使います．8 ビットの出力信号 dout は DAC に接続されており，ここに出力されたデータが最終的に音へと変換されます．

ソースコード 1　とにかく音を鳴らす sound モジュール（`sound.sv`）

```
1  ‘default_nettype none
2  module sound
3    (
4     input wire        clk,
5     input wire        rst,
6     input wire [7:0]  sw,
7     output wire [7:0] dout
8     );
9  
10    reg [11:0]         count;
11 
12    always_ff @(posedge clk) begin
13      if (rst)
14        count <= 12’d0;
15      else if (sw[0])
16        count <= count + 12’d1;
17    end
18 
19    assign dout = count[11]? 8’d255: 8’d0;
20 endmodule
21 ‘default_nettype wire
```

ソースコード 2　sound モジュール用のシミュレーションモジュール（sim_sound.sv）

```
‘default_nettype none
‘timescale 1ns/1ps
module sim_sound();
   localparam real    CLOCK_FREQ_HZ     = 1.5 * 10**6; // 1.5 MHz
   localparam real    CLOCK_PERIOD_NS   = 10**9 / CLOCK_FREQ_HZ;
   localparam integer SIMULATION_CYCLES = 10000;
   logic          clk, rst;
   logic [7:0]    sw;
   wire [7:0]     dout;

   sound sound_inst(.clk(clk), .rst(rst), .sw(sw), .dout(dout));

   initial begin
      clk <= 1'b0;
      repeat (SIMULATION_CYCLES) begin
       #(CLOCK_PERIOD_NS / 2.0)
         clk <= 1'b1;
       #(CLOCK_PERIOD_NS / 2.0)
         clk <= 1'b0;
         print();
      end
      $finish;
   end

   initial begin
      rst <= 1'b1;
    #(CLOCK_PERIOD_NS)
      rst <= 1'b0;
   end

   initial begin
      sw <= 8'b00000001;
   end

   task print();
      $write("time= %5d sw= %b count= %d dout= %d\n",
             $time, sw, sound_inst.count, dout);
   endtask
endmodule
‘default_nettype wire
```

さて，この回路の中心は 12 ビットのカウンタ（count）です．15 行目の if 文で一番左のボタン（sw[0]）が押されているときだけカウントアップされるように記述されています．押されていないときには（count）の値は変化しません．今回の実験ボードでは 1.5 MHz のクロックを利用します．したがって、このカウンタは，1 秒間に 1,500,000 回カウントアップされます．カウンタは 12 ビットですから，非負整数と考えれば最大値は 2 進数で 111111111111，10 進数では 4,095 です．この最大値にさらに 1 を足すとカウンタはオーバフローして 0 に戻ります．

一方，dout に出力される値は，19 行目からの assign 文で決まります．カウンタの最上位ビット（11 ビット目）が 1 のときには 10 進数で 255（2 進数では 11111111）が，そうでないときには 0 が出力されるように記述されています．つまり，カウンタの値が 2048 以上なら 255，そうでないなら 0 が出力されます．このよ

うにして周期的なデータが dout に出力されるのです．

どのような出力になるのか，ソースコード 2 に示したシミュレーションモジュールを使って確かめてみましょう．これは初めから 1 番左のボタンを押して 10,000 クロック進めるというものです．4 行目から 6 行目にかけての localparam は初めて登場する文法ですが，定数パラメータを定義するものです．ここでは real（実数型）や integer（整数型）のパラメータを定義しています．例えば，

```
localparam real CLOCK_FREQ_HZ = 1.5 * 10**6;
```

とすると，以降「CLOCK_FREQ_HZ」と記述すると，実数型で 1.5×10^6 と記述したのと同じになります．では，

```
% make sim
% vsim sim_sound.wlf &
```

として波形データを見てみます．カウンタ値 count と出力値 dout について，Radix を「Unsigned」に，Format を「Analog (automatic)」に設定すると，count は周期が 4,096 クロックの鋸歯状波[*1]，dout は同じ周期の矩形波[*2]になることが分かります．

それでは，いつものように論理合成し，FPGA に回路をコンフィギュレーションして音を聴いてみましょう．いかにも機械的な音ですが，ボタンを押すとスピーカーが鳴るはずです．

練習 1： この音の周波数を計算せよ．ちなみに，個人差もかなりあるが，人間の可聴周波数はだいたい 20 Hz から 20 kHz 程度である．当然ながら，求めた値はこの範囲に入っているはずである．

練習 2： ソースコード 1 の設計ではカウンタを 1 クロックに 1 ずつ増やしていた．SystemVerilog 記述の 16 行目を次のように変更し，1 クロックに 2 ずつ増やすように変更した．出力される音の周波数はいくらになるか?

```
count <= count + 12'd2;
```

2 音階の仕組み

音を出すこと自体は意外と簡単にできることが分かりました．せっかく実験ボードにはボタンスイッチが 8 個ありますので，それぞれをドレミファソラシドに対応させた電子ピアノを作ってみましょう．そのためには，まず，音の高さ（つまり周波数）と音階の関係を理解しておく必要があります．実はこれはこれでなかなか奥が深い話なのですが，今回は音楽理論の講義ではないのでここでは簡単な説明に留めます．

2.1 基準の音

オーケストラのコンサートに行くと，指揮者が登場して演奏を始める前に，演奏者が舞台上でチューニング（音合わせ）するのを見ることができます．楽器ごとに音の高さが微妙に異なると，一緒に演奏したときに音が濁ってしまうので，あらかじめ調整しておくのです．チューニングのやり方はどのオーケストラでもだいたい同じで，最初にオーボエ奏者が「ラ」の音を長く吹き，それに合わせてコンサートマスター（バイオリンの

[*1] 「きょしじょうは」と読みます．いわゆる「のこぎり波」のことです．

[*2] 「くけいは」と読みます．方形波ともいいます．

図 1　半音階と記譜

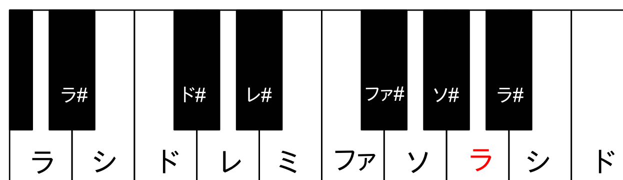

図 2　ピアノの鍵盤

首席奏者）が「ラ」の弦を調整し，その後，他の楽器が音を合わせていきます．ピアノを調律する時にも，一番初めにこの「ラ」の音を音叉に合わせて調整します．

西洋音楽では伝統的にこの「ラ」の音が基準の音として使われており，物理の分野ではその周波数は 440 Hz ということが国際標準で決められています．しかし，楽器の場合には 440 Hz ぴったりだとやや華やかさに欠けるため，実際には 440 Hz よりも数 Hz 高い周波数で「ラ」の音をチューニングすることが普通です[*3]．

2.2　オクターブと平均律

「ドレミファソラシド」と音階を歌うと 2 回「ド」が登場します．最初の「ド」は低い「ド」，最後の「ド」は高い「ド」です．この 2 つの「ド」の周波数はちょうど 2 倍の関係です．最初の「ド」の周波数が x ならば，高い「ド」の周波数は $2x$ です．このように周波数が 2 倍になる音の関係を 1 オクターヴと呼びます．基準の「ラ」の周波数を 440 Hz とすると，この音から 1 オクターヴ上の「ラ」の音の周波数は 880 Hz，1 オクターヴ下の「ラ」の音の周波数は 220 Hz になります．伝統的な西洋音楽で用いる音階では 1 オクターヴは 12 個の音から構成されています．これはピアノの白鍵と黒鍵の数を数えてみると分かります（図 1 および図 2）．

ここで周波数が 2 倍となる音の間（1 オクターヴ）を，隣り合う音の周波数の「比」が一定となるように 12 等分します．このように音階の各音の周波数を定めるのが「平均律」で，ピアノの調律には通常この平均律が使われます．周波数の「差」ではなく「比」が一定となるように 12 等分する点に注意して下さい．各音の周波数を順に並べると，等差数列ではなく等比数列になります．そして，その公比を 12 回掛けるとちょうど 2 となり，すなわち 1 オクターヴとなります．

練習 3： 基準の「ラ」音を 442 Hz とした平均律で調律されたピアノがある．白鍵の各音（ドレミファソラシド）の周波数を求めよ．

[*3] 一方，バロックや古典派の音楽が作曲された頃の基準音の周波数はもっと低かったとも言われており，時代考証に基づいて低めのチューニングで演奏する場合もあります．

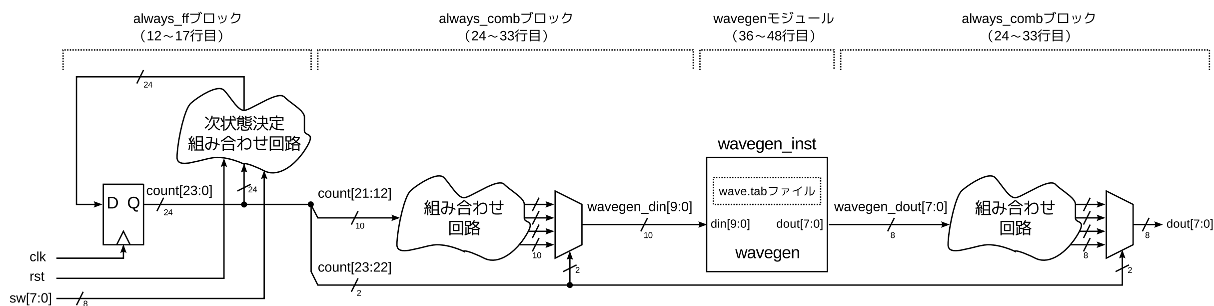

図 3　piano モジュールのハードウェア構成（行番号はソースコード 3 に対応）

ソースコード 3　電子ピアノの記述構成（piano.sv）

```
1  ‘default_nettype none
2  module piano
3    (
4     input wire         clk,
5     input wire         rst,
6     input wire [7:0]   sw,
7     output logic [7:0] dout
8     );
9
10    reg [23:0]         count;
11
12    always_ff @(posedge clk) begin
13      if (rst)
14        count <= 24’d0;
15      else
16        ...
17    end
18
19    logic [9:0] wavegen_din;
20    wire [7:0]  wavegen_dout;
21    wavegen wavegen_inst
22      (.din(wavegen_din), .dout(wavegen_dout));
23
24    always_comb begin
25      case (count[23:22])
26        2’b00: begin
27           wavegen_din <= count[21:12];
28           dout <= wavegen_dout;
29        end
30        2’b01:
31           ...
32      endcase
33    end
34 endmodule
35
36 module wavegen
37   (
38    input wire [9:0] din,
39    output reg [7:0] dout
40    );
41
42    always_comb begin
43       case (din)
44 ‘include "wave.tab"
45       endcase
46    end
47 endmodule
48 ‘default_nettype wire
```

3　電子ピアノの実装

3.1　仕様と設計方針

それでは電子ピアノの設計に取り掛かりましょう．仕様は以下のようにします．

- モジュール名は piano
- 入出力はソースコード 1 の sound モジュールと同じ
- クロックの周波数は 1.5 MHz
- 基準音（「ラ」）の周波数は 442 Hz
- 8 つの押しボタンスイッチを左から順にピアノの白鍵の各音に対応させる

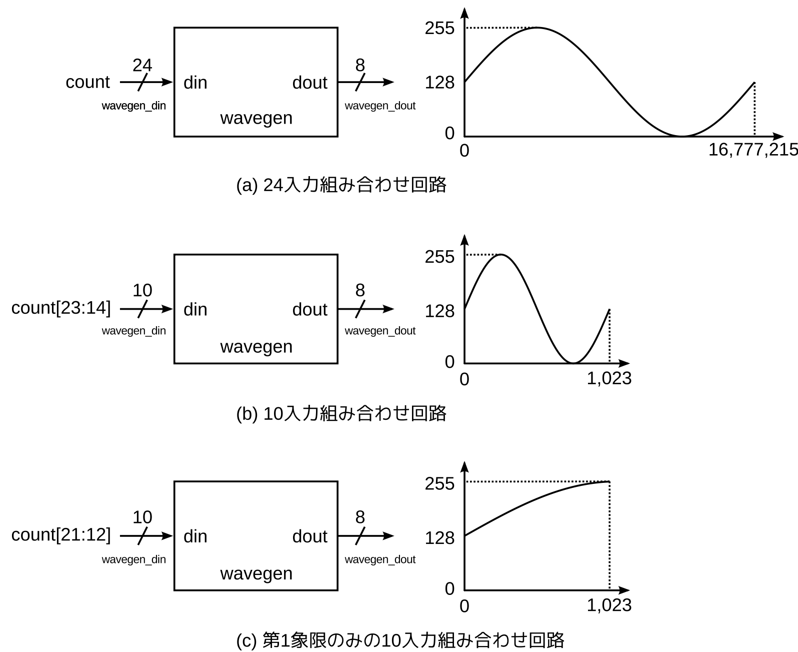

図 4　組み合わせ回路による関数テーブルの実現法

- 出力波形は 8 ビットの正弦波（10 進数で 0 から 255 までの値をとる）
- スイッチを押している間は音が鳴り続け，スイッチを離すと音は止む
- 同時に複数のスイッチが押された場合は，高い方の音のスイッチを優先する
- 24 ビットのカウンタを使って音の高さをコントロールする

sound モジュール（ソースコード 1）の出力波形は矩形波でしたが，今度は正弦波（sin 波）を出力させることにします．音色としてはフルートのような感じになるはずです．設計の大まかな枠組みを図 3 およびソースコード 3 に示します．sound モジュールに比べるとカウンタのビット幅が大きいので，より低い周波数の波を作り出すことができます．カウントアップする値を変化させれば出力音の高さが変わるので，押されたスイッチ（sw）に応じてこれを制御するように記述すれば電子ピアノの基本形ができあがります．このカウントアップ値の選択は，ソースコード 3 の 15 行目から始まる else 節に記述することになります．

練習 4：　クロック周波数を 1.5 MHz とする．24 ビットのカウンタを 1 クロックに 1 ずつカウントアップした場合，出力音の周波数はいくらか．

3.2　組み合わせ回路による正弦波の生成

残る課題は正弦波の生成です．三角関数の値を求めるには，Taylor 展開などを用いて四則演算で計算する方法や，関数値を事前に計算しておいてテーブル（表）にしておく方法などがあります．一般には関数テーブルの実現にはメモリを使うことが多いのですが，今回は簡単に，組み合わせ回路を使って実現する方法をとります．ソースコード 3 の構成では，サブモジュール wavegen がこの組み合わせ回路に対応します．

24 ビットのカウンタの値を入力とし，0 から 255 までの値をとる正弦波を出力すればよいので，もっとも安直に考えれば図 4（a）のように 24 入力 8 出力の組み合わせ回路を用いるのがよさそうです．あらかじめ 2^{24}

通り（0 から $16,777,215$）のカウンタの値に対して sin 関数の出力値を計算しておき，これを真理値表にしてしまおうという作戦です．しかし，残念ながらこの方法はうまくいきません．24 入力の組み合わせ回路はあまりに規模が膨大になってしまい FPGA に載せることができないのです．組み合わせ回路の複雑さは入力数に対して指数関数的に増加します．今回の実験の環境では，せいぜい 10 入力の組み合わせ回路までが限度です．

そこで考えられるのが，図 4（b）のようにカウンタの値の上位 10 ビット（count[23:14]）を 10 入力の組み合わせ回路に入力する方法です．カウンタの下位 14 ビットを切り捨てるので，図 4（a）の方法に比べると，時間軸上において「ぎざぎざ」の波形になります．精度をある程度犠牲にする代わりに回路を簡単化しようという方法です．

しかし正弦波の対称性に着目すると，10 入力の組み合わせ回路を使いながらもう少し精度を高めることができます．正弦波は第 1 象限（ラジアンで考えると 0 から π）の値を求めることができれば，残りの値は簡単に導くことができます．そこで図 4（c）のように，カウンタの 29 ビット目から 20 ビット目（count[21:12]）を第 1 象限の組み合わせ回路に入力し，その出力をカウンタの上位 2 ビット（count[23:22]）を使って変換してやればよいのです．このようにすることで，切り捨てるのはカウンタの下位 12 ビットだけでよくなります．すなわち図 4（b）の方法に比べて，4 倍「なめらか」な波形を出力することができるのです．

具体的には，ソースコード 3 の 24 行目から 32 行目の case 文において，count[21:12] と wavegen_din（wavegen モジュールへの入力）の関係と，wavegen_dout（wavegen モジュールからの出力）と dout（piano モジュールの出力）との関係を，カウンタの上位 2 ビット（count[23:22]）のよって変更する組み合わせ回路の記述を行います．

3.3 関数テーブルの記述

さて，組み合わせ回路で正弦波を生成する仕組みは分かりましたが，問題はその記述です．10 入力の組み合わせ回路の真理値表を作るには，$2^{10} = 1,024$ 通りの値を計算し記述しなければなりません．すべてを間違えないように手で入力するというのはまったく非現実的です．こういう機械的な作業は自動化したいものです．

そこでまず，関数テーブルの値を出力する C プログラムを記述します．そしてその出力をファイルに保存しておき，SystemVerilog 記述に取り込んでやるのです．SystemVerilog では「‘include」という記述を使って別ファイルの内容を読み込むことができます．C のプリプロセッサで言えば「#include」に相当します．ソースコード 3 では，wavegen モジュールの中の 44 行目において，

```
‘include "wave.tab"
```

と記述し，case 文の中身を wave.tab という別のファイルから読み込むようにしています．当然ながら，このファイルの中には

```
10’d0: dout <= 8’d128;
10’d1: dout <= 8’d128;
 ...
```

のように入力値（din）に対する出力値（dout）を，case 文の文法に合わせて 1,024 通り記述しておきます．つまり，このような記述を printf 関数で出力するような C プログラムを書いて，その出力をファイルに書き込んでおけばよいのです．

3.4 テーブル作成と検証

一連のシミュレーションによる検証と実機検証の手順を確認しておきましょう．

（1）関数テーブル作成

関数値出力プログラム（wave.c）を作成し，コンパイルしてテーブルファイル（wave.tab）に書き出します．

```
% gcc -o wave -Wall -O2 wave.c -lm
% ./wave > wave.tab
```

（2）シミュレーション実行

次にシミュレーションを行います．シミュレーションモジュールの例を例をソースコード 4 に示します．これはスイッチボタンを左から順番に 50,000 クロックずつ押していく動作を記述したものです．

```
% make sim
```

（3）出力波形確認

波形ビューアを起動した後，dout をアナログモードで表示し，綺麗な正弦波が出力されているか確認しましょう．

```
% vsim sim_piano.wlf &
```

（4）C プログラムによる検証

配布した C プログラムは，シミュレーションログファイル（sim_piano.log）を読み込み，各スイッチを押した際に dout が中央値（128）をクロスする時間の間隔から出力周波数を求めて期待値と比較します．また，dout 振幅が 0 から 255 まで振れているか，出力に不連続性（直前の出力と 1 以上の変化しているところ）がないかをチェックします．

```
% make check
```

（5）実機確認

論理合成・配置配線の後，FPGA にコンフィギュレーションして名曲を奏でてみましょう！また，複数のボタンを押したとき上の高い音が優先されているか確認しましょう．

```
% make
% make config
```

ソースコード 4　piano モジュール用シミュレーションモジュール

```
1  `default_nettype none
2  `timescale 1ns/1ps
3  module sim_piano();
4     localparam real    CLOCK_FREQ_HZ       = 1.5 * 10**6; // 1.5 MHz
5     localparam real    CLOCK_PERIOD_NS     = 10**9 / CLOCK_FREQ_HZ;
6     localparam integer BUTTON_PRESS_CYCLES = 50000;
7     localparam integer SIMULATION_CYCLES   = BUTTON_PRESS_CYCLES * 8 + 100;
8     logic           clk, rst;
9     logic [7:0]     sw;
10    wire [7:0]      dout;
11
12    piano piano_inst(.clk(clk), .rst(rst), .sw(sw), .dout(dout));
13
14    initial begin
15       clk <= 1'b0;
16       repeat (SIMULATION_CYCLES) begin
17        #(CLOCK_PERIOD_NS / 2.0)
18          clk <= 1'b1;
19        #(CLOCK_PERIOD_NS / 2.0)
20          clk <= 1'b0;
21          print();
22       end
23       $finish;
24    end
25
26    initial begin
27       rst <= 1'b1;
28     #(CLOCK_PERIOD_NS)
29       rst <= 1'b0;
30    end
31
32    initial begin
33       sw <= 8'b00000000;
34     #(CLOCK_PERIOD_NS * 2.0)
35       sw <= 8'b00000001;
36       repeat (8) begin
37        #(BUTTON_PRESS_CYCLES * CLOCK_PERIOD_NS)
38          sw <= (sw << 1);
39       end
40    end
41
42    task print();
43       $write("time= %5d sw= %b count= %d dout= %d\n",
44              $time, sw, piano_inst.count, dout);
45    endtask
46 endmodule
47 `default_nettype wire
```