情報工学実験 II

第 6 回： ステートマシン

2019 年 1 月 24 日

柴田 裕一郎（shibata@cis.nagasaki-u.ac.jp）

1　ステートマシン

これまでカウンタ（数を数える回路）を中心とした順序回路を設計してきましたが，今回はより一般的な動作を行うステートマシン（state machine: 状態遷移機械）について考えます．以下に簡単な例題を用いて，状態遷移図から具体的なハードウェアを設計する手順を説明します．

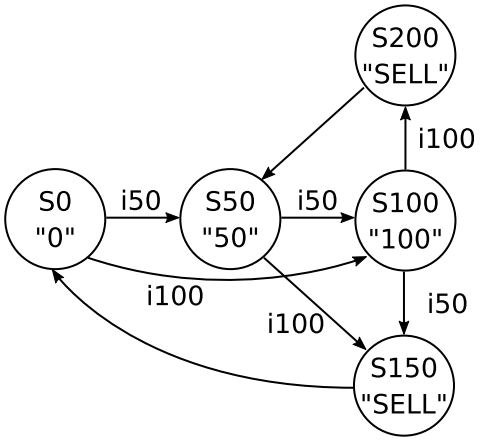

図 1　自動販売機の状態遷移図

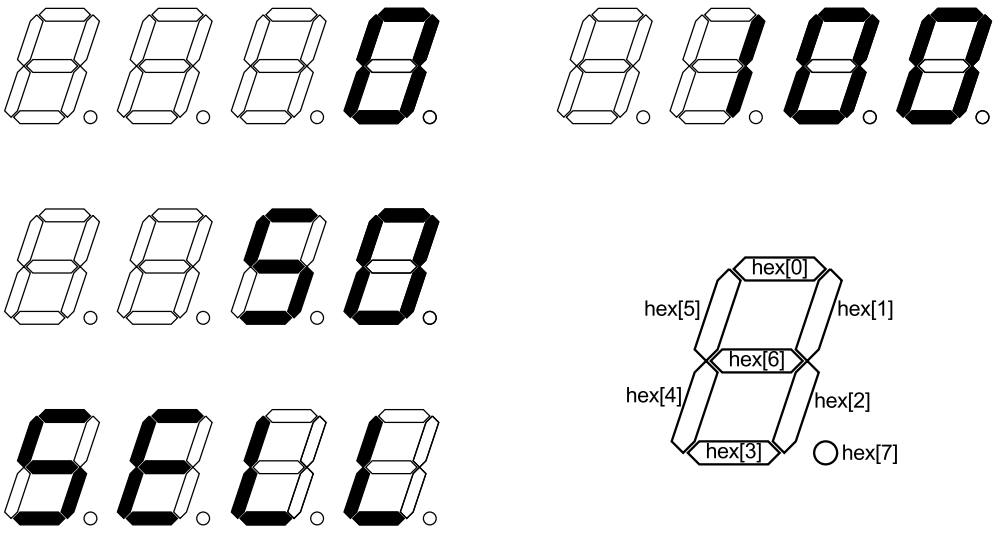

図 2　自動販売機の LED 出力パターン（hex5 および hex4 は常に消灯）

1.1 自動販売機の設計

150 円の商品を販売する自動販売機を考えます．入力としては，100 円玉と 50 円玉を受け付けることとします．硬貨が投入されると，硬貨の種類に対応した信号（「`i100`」または「`i50`」）が 1 クロックサイクル ‘1’ になります．この際，これらの 2 つの信号が同時に 1 になることは考えなくてよいことにします．出力は 7 セグメント LED とし，投入された硬貨の総額を表示します．投入された金額が 150 円以上になったら商品を販売することを示すために 1 クロックサイクルの間「SELL」と表示します．お釣りがある場合には返却せずに，投入済みの金額として引き継ぐことにします．

ステートマシンの設計では，まず，状態を決めて状態遷移図を書きます．今回は投入された金額に応じて状態を割り当てれば良いでしょう．図 1 に状態遷移図を示します．D-FF を使った同期式順序回路ではクロック信号が立ち上がるたびに状態が移り変わっていきます．

次に，図 1 に示した各状態に対して，それぞれ固有の 2 進数を割り付ける必要があります．この作業はステートコーディング（state coding: 状態符号化）と呼ばれます．2 進符号化やワンホット符号化，Gray コードによる符号化など，様々な状態符号化の手法がありますが，今回は設計時に符号化は行わず論理合成ツールに任せることにします．こうすることで，状態符号化のことを気にせず，状態名だけを考えて抽象度の高い設計をすることができます．

この後は比較的機械的な作業になります．ステートマシンの枠組みを示すと図 3 のようになります．

- 現在の状態と入力から**次の状態**を作る次状態決定回路
  現在の状態を記憶しておくための D-FF と，次状態を作る組み合わせ回路から構成されます，組み合わせ回路の出力は D-FF の各 D 入力へとフィードバックされます．各 D-FF はクロックの立ち上がりのたびにこのフィードバックされた値を取り込み，状態の遷移が起こります．
- 現在の状態と入力から**出力**を作る出力決定回路
  組み合わせ回路で実現されます．現在の状態から出力が一意に決まるのが Moore 型，現在の状態と入力から出力が決まるのが Mealy 型です．今回の自動販売機は Moore 型の例です．この例では出力決定回路の出力が外部の 7 セグメント LED へと接続されます．

1.2 同期微分回路による立ち上がりエッジ検出

実験ボードでは，硬貨投入を検出する信号（「`i100`」および「`i50`」）を外部から与えるのに押しボタンスイッチを使いたいのですが，ボタンスイッチが長押しされても複数回の状態遷移が起きないように，1 クロックサイクルだけ ‘1’ が入力されるようなメカニズムが必要です[*1]．つまり，ボタンが押されている間ずっと ‘1’ が入力されるのではなく，ボタンが押されて入力が ‘0’ から ‘1’ に変化した立ち上がりエッジを検出し，1 クロックサイクルだけ ‘1’ を生成する必要があるのです．

このような回路は同期微分回路と呼ばれます．その構成と動作を図 4 に示します．入力 `din` の立ち上がりエッジが検出され，1 クロックサイクル分のパルスが `dout` に出力されることが分かります．

[*1] 本来，押しボタンスイッチを使う場合には，非同期入力によるメタステーブルを対処するためのシンクロナイザやチャタリングの問題に対処するためのフィルタなども必要ですが，これらの回路はトップモジュール側に入っていますので，今回の実験では気にしなくて結構です．

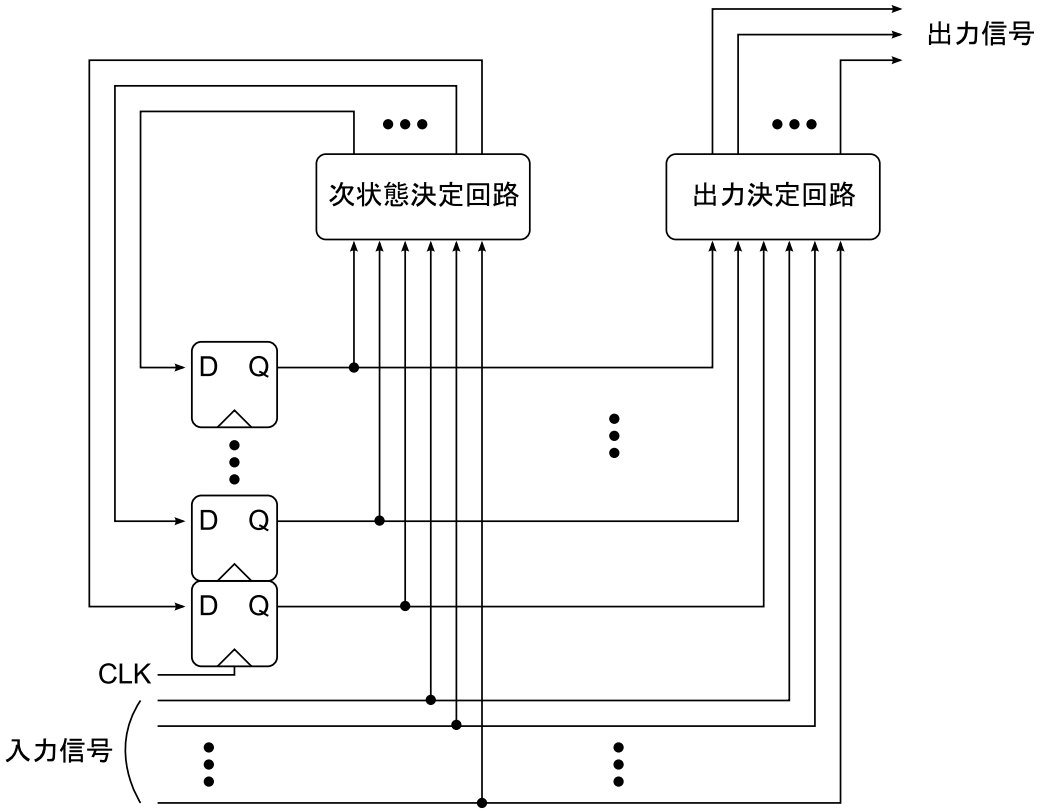

図 3　ステートマシンの枠組み

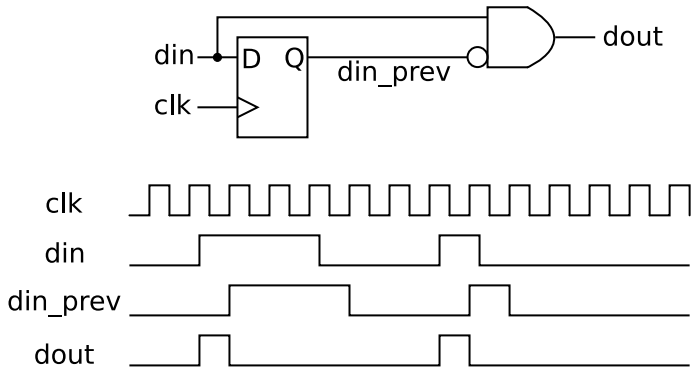

図 4　同期微分回路の構成と動作（FF 出力の初期値を 0 とした場合）

2　ステートマシンの SystemVerilog 記述

2.1　自動販売機の記述例

ここまでの設計を SystemVerilog で記述した例をソースコード 1 に示します.

ソースコード 1　自動販売機の記述例（fsm.sv）

```
1  `default_nettype none
2  typedef enum {S0, S50, S100, S150, S200} state_t;
3
4  module fsm
5    (
6     input wire         clk,
7     input wire         rst,
8     input wire [7:0]   sw,
9     output logic [7:0] hex5, hex4, hex3, hex2, hex1, hex0
10    );
11
12    wire i50, i100;
13
14    reg [7:0]         sw_prev;
15    always_ff @(posedge clk) begin
16       if (rst)
17         sw_prev <= 8'b0;
18       else
19         sw_prev <= sw;
20    end
21    assign i50  = sw[0] & (!sw_prev[0]);
22    assign i100 = sw[1] & (!sw_prev[1]);
23
24    state_t state;
25    always_ff @(posedge clk) begin
26       if (rst)
27         state <= S0;
28       else begin
29          case (state)
30            S0:
31              if (i50)
32                state <= S50;
33              else if (i100)
34                state <= S100;
35
36            S50:
37               if (i50)
38                 state <= S100;
39               else if (i100)
40                 state <= S150;
41
42            S100:
43              if (i50)
44                state <= S150;
45              else if (i100)
46                state <= S200;
47
48            S150:
49              state <= S0;
50
51            S200:
52              state <= S50;
53          endcase
54       end
55    end
56
57    always_comb begin
58       case (state)
59         S0: begin
60            hex5 <= 8'b00000000;
61            hex4 <= 8'b00000000;
62            hex3 <= 8'b00000000;
63            hex2 <= 8'b00000000;
64            hex1 <= 8'b00000000;
65            hex0 <= 8'b00111111;
66         end
67
68         S50: begin
69            hex5 <= 8'b00000000;
70            hex4 <= 8'b00000000;
71            hex3 <= 8'b00000000;
72            hex2 <= 8'b00000000;
73            hex1 <= 8'b01101101;
74            hex0 <= 8'b00111111;
75         end
76
77         S100: begin
78            hex5 <= 8'b00000000;
79            hex4 <= 8'b00000000;
80            hex3 <= 8'b00000000;
81            hex2 <= 8'b00000110;
82            hex1 <= 8'b00111111;
83            hex0 <= 8'b00111111;
84         end
85
86         S150: begin
87            hex5 <= 8'b00000000;
88            hex4 <= 8'b00000000;
89            hex3 <= 8'b01101101;
90            hex2 <= 8'b01111001;
91            hex1 <= 8'b00111000;
92            hex0 <= 8'b00111000;
93         end
94
95         S200: begin
96            hex5 <= 8'b00000000;
97            hex4 <= 8'b00000000;
98            hex3 <= 8'b01101101;
99            hex2 <= 8'b01111001;
100           hex1 <= 8'b00111000;
101           hex0 <= 8'b00111000;
102        end
103      endcase
104   end
105 endmodule
106 `default_nettype wire
```

■状態記述用の列挙型の定義（2 行目）　状態を格納するための新しい列挙型を定義しています．typedef による新しい型の定義や enum による列挙型は C 言語と同様の文法を持ちます．ここでは，S0，S50，S100，S150，S200 という 5 つの状態名を格納できる列挙型を新たに定義し，その型の名前を state_t 型と名付けています．

■同期微分による立ち上がり検出（14 行目〜22 行目）　sw_prev は同期微分のためのレジスタで，1 クロックサイクル前のスイッチ入力 sw の値を保持します．同期微分により，sw[0] が押されたときは i50 に，sw[1] が押されたときは i100 に 1 クロックサイクルの '1' が出力されます．

■状態保持用の変数（24 行目）　2 行目で定義した state_t 型を使って変数 state を定義しています．この変数がステートマシンの現在の状態を保持するレジスタに対応します．

■状態の制御（25 行目〜55 行目）　クロックが立ち上がる度に，状態がどのように遷移するかを記述しています．29 行目からの case 文で，各状態における遷移条件を表していますが，S0 や S50 などの状態名を使って記述できるのが列挙型を用いた効果です．各状態がどのような 2 進数で表されるのかは，論理合成の際にツールによって決定されます．なお，state に代入が起こなわれない場合には，現在の状態に留まります．

■出力の記述（57 行目）　各状態において，どのような出力をするのか組み合わせ回路として記述しています．組み合わせ回路を手続的に記述するため（case 文を使うため）always_comb ブロックを用いています．また，手続的代入を行うため，出力である hex5，hex4，hex3，hex2，hex1，hex0 は logic 型としています（9 行目）．

2.2　シミュレーションモジュール記述例

自動販売機の例題用のシミュレーション記述例をソースコード 2 に示します．ほとんどこれまでの例と同様の記述ですが，3 点だけ注意点を補足します．

■無限ループ（36 行目〜39 行目）　forever 文は無限ループ（終了条件のないループ）を記述するための文法で，begin から end までの文がシミュレーションが終了するまで永遠に繰り返されます．この例では，シミュレーションの終了は別の initial ブロック内で指示されています（24 行目）．

■乱数によるスイッチ入力（38 行目）　このシミュレーションでは，毎クロックサイクル乱数によってスイッチ入力を与えています．使うスイッチは sw[0] と sw[1] と 2 つであり，また，これらが同時に押された際の動作は考えなくてよいことにしているので[*2]，$urandom_range を使って '0'，'1'，'2' の 3 値の乱数を振っています．この乱数にしたがって 1 ビットの '1' にシフト演算を行うことで，sw[0] だけ '1'，sw[1] だけ '1'，sw[0] も sw[1] も '0' の 3 種の入力から 1 つを選んでいます．なお，ここでは左シフト演算に負数を指定すると右シフト演算になることを利用しています．

■状態名の表示（46 行目）　SystemVerilog では変数名に「.name」を付加し「%s」を用いることで，列挙型の変数の中身を文字列として表示することができます．符号化する前に状態名の段階で動作の確認ができるのが，ハードウェア記述言語による設計の 1 つのメリットです．

[*2] 今回の実装では，ソースコード 1 を見るとわかるように，両方同時に押された場合には 50 円玉が投入されたと認識されるようになっています．

ソースコード 2　fsm モジュール用シミュレーションモジュール（sim_fsm.sv）

```
1  `default_nettype none
2  `timescale 1ns/1ps
3  module sim_fsm();
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 SIMULATION_CYCLES   = 100;
7     logic           clk, rst;
8     logic [7:0]     sw;
9     wire [7:0]      led, hex0, hex1, hex2, hex3, hex4, hex5;
10
11    fsm fsm_inst(.clk(clk), .rst(rst), .sw(sw),
12                 .hex5(hex5), .hex4(hex4), .hex3(hex3),
13                 .hex2(hex2), .hex1(hex1), .hex0(hex0));
14
15    initial begin
16       clk <= 1'b0;
17       repeat (SIMULATION_CYCLES) begin
18        #(CLOCK_PERIOD_NS / 2.0)
19          clk <= 1'b1;
20        #(CLOCK_PERIOD_NS / 2.0)
21          clk <= 1'b0;
22          print();
23       end
24       $finish;
25    end
26
27    initial begin
28       rst <= 1'b1;
29     #(CLOCK_PERIOD_NS)
30       rst <= 1'b0;
31    end
32
33    initial begin
34       sw <= 8'b00000000;
35     #(CLOCK_PERIOD_NS * 2.0)
36       forever begin
37        #(CLOCK_PERIOD_NS)
38          sw <= 1'b1 << ($urandom_range(0, 2) - 1);
39       end
40    end
41
42    task print();
43       $write("time= %5d sw= %b i100= %b i50= %b hex= %b %b %b %b %b %b state= %s\n",
44              $time, sw, fsm_inst.i100, fsm_inst.i50,
45              hex5, hex4, hex3, hex2, hex1, hex0,
46              fsm_inst.state.name);
47    endtask
48 endmodule
49 `default_nettype wire
```