情報工学実験 II

第 3 回： D フリップフロップとカウンタ

2018 年 12 月 20 日
柴田 裕一郎（shibata@cis.nagasaki-u.ac.jp）

1 基本事項の復習

これまでは組み合わせ回路を作ってきましたが，今回は順序回路を SystemVerilog で記述して実装します．論理回路は組み合わせ回路と順序回路に分類されるので，この 2 種類の作り方さえマスターすれば基本的にはどんなハードウェアでも作れるようになります．それでは，具体的な記述方法を説明する前に，順序回路について簡単に復習しておきましょう．

1.1 組み合わせ回路と順序回路

そもそも「組み合わせ回路」や「順序回路」とは何のことだったでしょうか? ここでもう一度その定義をはっきりとさせておきましょう．

- 組み合わせ回路
  出力が入力だけによって決まる回路．つまり，入力の値が決まれば出力の値も一意に決まる．もっとわかりやすく言えば，同じ値が入力されたときには，いつでも同じ値が出力される．
- 順序回路
  出力が入力と回路の状態によって決まる回路．同じ入力値が与えられても，そのときの回路の状態によって出力される値が異なることがある．

たとえば，前回作った 16 ビット加減算器は，2 つの 16 ビットの入力（`ain` と `bin`）と，加算を行うのか減算を行うのかを指定する 1 ビットの入力（`sub`）が決まれば，出力は一意に決まります．したがって組み合わせ回路です．

順序回路の例でよく登場するのは自動販売機です．80 円のコーヒーを売っている自動販売機を考えてみます．「お金が全く入っていない状態」でボタンを押しても何も出てきませんが，「100 円が入っている状態」で同じボタンを押すとコーヒーやお釣りといった「出力」が出てきます．

順序回路を作る場合，「回路の状態」を保持するために何らかの記憶素子が必要になります．たとえば自動販売機の例では，現在お金がいくら投入されているかという「状態」を保持（記憶）する必要があります．皆さんも，すでに SR ラッチや D フリップフロップ，JK フリップフロップなど，何種類かの記憶素子の仕組みを勉強したとことと思います．SystemVerilog を用いた設計では，エッジトリガ型 D フリップフロップがもっとも良く使われる記憶素子です．単にフリップフロップ（FF と略すことが多い）といえば D フリップフロップ（以下，D-FF と略します）のことを指すのが普通です．

1.2 D フリップフロップ

D-FF は図 1 に示すように，入力 D と出力 Q を持ち，さらにクロック入力 CLK を持っています．また，これらに加えて反転出力 $\overline{\mathrm{Q}}$ を持つものが一般的ですが，これは単に出力 Q を反転したものなので，ここでは省

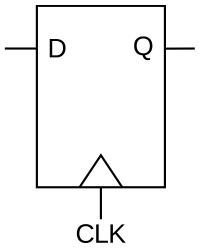


図 1　D-FF の記号

表 1　D-FF の特性表

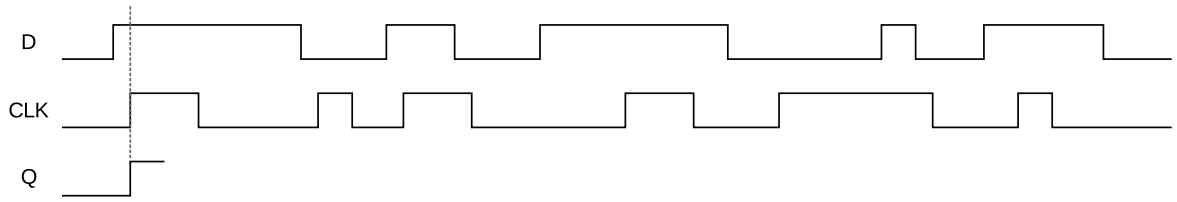
D	CLK	Q
0	↑	0
1	↑	1
0	その他	直前の Q の値
1	その他	直前の Q の値

略して説明をすすめます．

D-FF の動作を理解する上で大切なのは，クロック入力におけるエッジトリガの考え方を理解することです．普通のディジタル回路の信号線は，それが ‘0’ であったり，あるいは ‘1’ であったりすることに意味がありますが，D-FF のクロック入力はそのレベル（‘0’ なのか ‘1’ なのか）よりも変化（エッジ）に意味があるのです．D-FF の動作は以下のとおりです．

- クロック入力の ‘0’ から ‘1’ への変化時（立ち上がりエッジ）
  入力 D の値を取り込む．出力 Q はそのときの D の値になる．
- その他の時
  入力 D の値を記憶しつづける．出力 Q の値もそのまま変わらない．

これを特性表にまとめると表 1 のようになります．ここで，↑の記号は立ち上りエッジを示しています[*1]．それでは練習として次のタイミングチャートを完成させてみましょう．



1.3　クロックとリセット

多くのディジタルシステムでは，水晶振動子を用いて 0 と 1 を周期的に繰り返す信号を作り，これをクロック信号として利用します．この信号の周波数（周期の逆数）がパソコン等でもお馴染みのクロック周波数です．本実験で使用する FPGA ボードには 24 MHz[*2]のクロック信号を生成する発振子が搭載されています．

ところで，表 1 の特性表にあるように，D-FF はクロック信号の立ち上がり以外のときは直前の Q の値を保持します．それでは電源投入直後，まだクロック信号が一度も立ち上がっていないときには，出力 Q の値はどうなるのでしょうか? 実はこれは 0 になるか 1 になるか誰にも分かりません．このように 0 なのか 1 なのか分からない値のことをハードウェア設計の分野では不定値と呼んでいます．SystemVerilog では不定値を「x」で表すことになっています．

[*1] クロックの立ち下がりでデータを取り込む D-FF もありますが，動作の基本は同じです．
[*2] 1 秒間に 24,000,000 回振動します．周波数 1 MHz は 10^6 Hz であり，2^{20} Hz ではありません．

しかし，すべてのフリップフロップの最初の値が不定値だとシステム全体の挙動が制御不能になってしまいます．そこで図 1 に示した D 入力，Q 出力，クロック入力の 3 つの他に，初期化用の「リセット入力」を備えたものがよく使われます．リセットがかかると D-FF の出力 Q は 0 になります．また，出力を 1 に初期化する「セット入力」を持つ D-FF もあります．通常は，システムの起動時に自動的にリセットがかかり D-FF が初期化される仕組みを設計するのが普通です．

2 例題 1：3 ビットカウンタ

それでは D-FF を使った順序回路の例として，次のような仕様をもつ 3 ビットカウンタ（数を数える回路）の設計について考えてみましょう．

- 入出力ポートのインタフェースは以下のとおり
  - `clk`: クロック入力
  - `rst`: アクティブ High の同期リセット入力[*3]
  - `led`: 8 ビットの LED 出力端子
- クロックの立ち上がりのたびに，$000_{(2)}$，$001_{(2)}$，$010_{(2)}$，... という具合にカウントアップしながら 3 ビットの 2 進数をそのまま 3 個の LED に出力する
- LED 出力は 8 ビットあるが，上位 5 ビットは常に 0 とする
- カウンタの値が $111_{(2)}$ まで行ったら次は $000_{(2)}$ に戻す
- `rst` に '1' が入力されたらカウンタの値を $000_{(2)}$ に戻す

2.1 SystemVerilog による記述例

この 3 ビットカウンタの SystemVerilog による記述例をソースコード 1 に示します．モジュール名は `binary_counter` としています．順序回路を記述するために登場した新しい文法事項を中心に以下に説明します．図 2 のような構造を頭に浮かべながら記述の意味を把握して下さい．

■レジスタ変数（10 行目） 今回は 3 ビットのカウンタを作りたいので，3 個の D-FF が必要です．SystemVerilog では D-FF は reg 型の変数として定義できます．ここでは，配線のときと同じように `[2:0]` という記述を使って 3 ビットの D-FF を定義しています．変数名は `count` としています．reg はレジスタ（register）の略です．教科書的には「多ビットのフリップフロップのことをレジスタという」ということになっていますが，実際のところはフリップフロップとレジスタは同じ意味で使われることが多いです．

■always_ff ブロック（12 行目〜17 行目） レジスタ（フリップフロップ）の動作を記述するためのブロックです．always_ff 文は通常「`@(posedge クロック信号)`」という記述を伴い，「クロック信号」の立ち上がりエッジのたびに起動され，begin から end までの文が順番に実行されます．always_ff の「ff」は，もちろん flip-flop（フリップフロップ）の略です．「posedge」は positive edge（立ち上がりエッジ）の略です．always_ff ブロックの中では，reg 型の変数に手続的代入ができます．なお，always_ff ブロックの中で代入が行われる変

[*3] 「アクティブ High のリセット入力」とは High（1）が入力されたときにリセットが有効になるという意味です．反対に，Low（0）が入力されるとリセットがかかる場合には「アクティブ Low のリセット入力」といいます．「同期リセット入力」とは，クロックの立ち上がり時にリセット入力がアクティブだとリセットが有効になるという意味です．クロックとは関係なく，リセット入力がアクティブになるとすぐにリセットがかかるものは「非同期リセット」といいます．

ソースコード 1　3 ビットカウンタ（binary_counter.sv）

```
1  `default_nettype none
2  // 3-bit binary counter
3  module binary_counter
4    (
5     input wire        clk,
6     input wire        rst,
7     output wire [7:0] led
8     );
9
10    reg [2:0] count;
11
12    always_ff @(posedge clk) begin
13      if (rst)
14        count <= 3'd0;
15      else
16        count <= count + 1'b1;
17    end
18
19    assign led = {5'b00000, count};
20 endmodule
21 `default_nettype wire
```

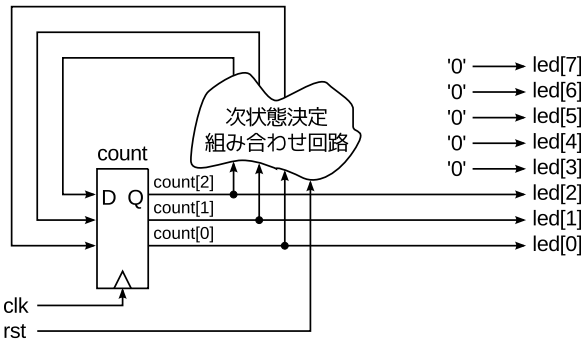


図 2　3 ビットカウンタの構造

数には，そのブロックの外では一切代入することができないので注意が必要です．

■リセット記述（13 行目・14 行目）　if 文を用いてリセット入力 rst が 1 ならばレジスタ変数 count を 0 にクリアしています．手続的代入なので代入記号「<=」を使っています．このように，always_ff ブロック内では，まず初めにリセット動作の if 文を記述するのが一般的です．なぜなら，リセットは他の動作よりも優先されるべき動作だからです．

■カウンタ動作（15 行目・16 行目）　15 行目からの else 節はリセットがかかっていないときの動作です．16 行目でレジスタ変数 count に 1 を加えています．前述のように，この always_ff 文はクロックの立ち上がりのたびに起動されるので，クロックの立ち上がりのたびに count が 1 ずつ増えることになります．逆に言うと，クロックの立ち上がりと次のクロックの立ち上がりとの間には，例え入力値が変化しても count の値は変わりません．これが記憶の動作に対応しています．なお，count は 3 ビットで定義されているので，$111_{(2)}$ の次はオーバフローして自動的に $000_{(2)}$ になります．

■連接演算子（19 行目）　SystemVerilog では「{」と「}」は連接演算子と呼ばれます．複数の信号線をカンマ（「,」）で区切って「{」と「}」で囲むと，それらを並べて 1 つのバス信号線を作ることを意味します．左に書いた信号線が上位になるように並べられます．19 行目の記述では，5 ビットの 0（5'b00000）を上位に，3 ビットのレジスタ count を下位に並べて合計 8 ビットのバス信号線を作り，これをそのまま 8 ビットの出力 led に接続しています．

2.2　シミュレーションモジュールの記述例

次に 3 ビットカウンタ用のシミュレーションモジュールの記述例をソースコード 2 に示します．ポイントとなるのは，検査対象モジュールである 3 ビットカウンタに，どのようにクロック信号とリセット信号を与える

ソースコード 2　sim_binary_counter.sv

```
1  `default_nettype none
2  `timescale 1ns/1ps
3  // simulation module for binary_counter
4  module sim_binary_counter();
5     logic clk, rst;
6     wire [7:0] led;
7
8     binary_counter binary_counter_inst
9       (.clk(clk), .rst(rst), .led(led));
10
11    initial begin
12       clk <= 1'b0;
13       repeat (25) begin
14        #20
15          clk <= 1'b1;
16        #20
17          clk <= 1'b0;
18          print();
19       end
20       $finish;
21    end
22
23    initial begin
24       rst <= 1'b1;
25     #30
26       rst <= 1'b0;
27    end
28
29    task print();
30       $write("time= %5d led= %b count= %d\n",
31              $time, led, binary_counter_inst.count);
32    endtask
33 endmodule
34 `default_nettype wire
```

かです．この点に注目しながら，ソースコード 2 を見てみましょう．

■2 つの initial ブロック（11 行目〜21 行目・23 行〜27 行目）　まず，このモジュールには 2 つの initial ブロックがあります．11 行目から始まる 1 つ目の initial ブロックは，クロック入力（clk）に値を与える記述です．23 行目から始まる 2 つ目の initial ブロックはリセット入力（rst）に値を与える記述です．重要なことは，これらの 2 つの initial ブロックはそれぞれ並列に動作することです．シミュレーション開始時に同時に 2 つの initial 文が起動され，それぞれのブロック内の文が実行されていきます．もちろん，それぞれのブロック内部では begin から end までの文が上から順番に逐次的に実行されます．

■repeat 文による繰り返し（13 行目）　前回，すでに for 文を使った繰り返しの記述は出てきましたが，単に指定した回数の繰り返しをしたい場合には repeat 文が使えます．「repeat（ループ回数）」とすることで，begin から end までの処理を指定した回数だけ繰り返すことができます．ここでは，ループ回数に 25 が指定されているので，「20 ナノ秒待ってから clk を 1 にし，さらに 20 ナノ秒待ってから clk を 0 に戻して表示用のタスク print() を呼び出す」という処理を 25 回繰り返すことになります．したがって，clk には 20 ナノ秒間ごとに 1 と 0 が交互に与えられるため，このシミュレーションにおけるクロック周期は 40 ナノ秒，クロック周波数は 25 MHz となります．ちなみに，周波数は周期の逆数です．

■シミュレーションの表示（30 行目・31 行目）　順序回路の動作検証においては時間の概念が重要です．SystemVerilog ではシステムタスク $time によって現在のシミュレーション時間を取得できます（31 行目）．単位はタイムスケール（この例では 1 ナノ秒）です．このシミュレーションモジュールでは，取得した時間をフォーマット指定子「%5d」を用いて 5 桁の整数として画面に表示しています（30 行目）．

■階層指定によるモジュール内部信号の表示（31 行目）　検証対象モジュールである 3 ビットカウンタ（binary_counter）の出力は 3 ビットの led だけですが，ときには出力端子だけではなく，モジュールの内部信号も画面に表示させて動作確認したい場合もあります．やや強引な方法ですが，SystemVerilog では「インスタンス名.信号名」とすることで，モジュール内部の信号にアクセスすることができます．31 行目では，binary_counter_inst.count とすることで，検証対象モジュール内部のレジスタ count の値を直接表示し

ています．ちなみに，これはソースコード 1 の 10 行目で定義されているレジスタです．

2.3 検証用 C プログラムの記述例

それではシミュレーションを実行してみます．次のような表示が確認できるはずです．

```
# time=    40 led= 00000000 count= 0
# time=    80 led= 00000001 count= 1
# time=   120 led= 00000010 count= 2
# time=   160 led= 00000011 count= 3
# time=   200 led= 00000100 count= 4
# time=   240 led= 00000101 count= 5
# time=   280 led= 00000110 count= 6
# time=   320 led= 00000111 count= 7
# time=   360 led= 00000000 count= 0
# time=   400 led= 00000001 count= 1
# time=   440 led= 00000010 count= 2
...
```

この程度のカウンタならば動作検証は目視によるチェックでも構わないでしょう．しかし，後々複雑な設計をすることも考えて，自動検証用の C プログラムを作ってみましょう．記述例をソースコード 3 に示します．

前回の加減器用の検証プログラムと概ね同じ流れになっています．前回と異なるのは，シミュレーションログファイルの検証が 1 行済むごとに，40 行目で次の行のカウンタの期待値を計算している点です．ログファイルには検証対象モジュールの出力端子である led と内部レジスタの count の値が記録されています．前者は 2 進表記，後者は 10 進表記されていますが，今回のカウンタでは両者は本質的には同じ内容であることに注意して下さい．33 行目と 34 行目でそれぞれの値を読み込んでいますが，2 進表記の文字列を整数にするために strtol 関数を使っています．37 行目では，両者がそれぞれカウンタの期待値と一致するかをチェックしています．

ソースコード 3　検証用プログラム例（check_sim_binary_counter.c）

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #define BUF_SIZE 256
5
6  int main(int argc, char *argv[])
7  {
8    FILE *fp;
9    char buf[BUF_SIZE];
10   int  lineNumber = 1;      /* 行数カウンタ */
11   int  checkedLines = 0;    /* 検査済みテストベクタ数 */
12   int  expectedCount = 0; /* カウンタの期待値 */
13
14   /* ログファイルが指定されていなければ終了 */
15   if (argc != 2) {
16     fprintf(stderr, "error: ログファイルが未指定\n");
17     exit(1);
18   }
19
20   /* ログファイルがオープンできなれば終了 */
21   if ((fp = fopen(argv[1], "r")) == NULL) {
22     fprintf(stderr, "error: ファイルオープン失敗: %s\n", argv[1]);
```

```
23      exit(1);
24    }
25
26    /* ログファイルから 1行ずつ読み期待値と比較 */
27    while (fgets(buf, BUF_SIZE, fp) != NULL) {
28      /*「# time=」で始まる行だけ処理 */
29      if (strncmp(buf, "# time=", 7) == 0)  {
30        int led, count;
31
32        /* 出力値の読み込み */
33        led   = strtol(buf + 19, NULL, 2); /* 2進数として読み込み */
34        count = atoi(buf + 34);
35
36        /* 出力と期待値を比較し異なれば終了 */
37        if (led == expectedCount && count == expectedCount) {
38          checkedLines++;
39          /* 次のカウンタ期待値の計算 */
40          expectedCount = (expectedCount + 1) % 8;
41        }
42        else {
43          fprintf(stderr, "error: %d 行目: ", lineNumber);
44          fprintf(stderr, "count=%d count 期待値=%d ", count, expectedCount);
45          fprintf(stderr, "led=0x%02x led 期待値=0x%02x\n", led, expectedCount);
46          exit(1);
47        }
48      }
49      lineNumber++;
50    }
51
52    /* 検査したテストベクタ数が 0なら終了 */
53    if (checkedLines == 0) {
54      fprintf(stderr, "error: テストベクタが見つかりませんでした\n");
55      exit(1);
56    }
57
58    /* 検査したテストベクタ数を表示して正常終了 */
59    printf("success: 検査テストベクタ数: %d\n", checkedLines);
60    return (0);
61  }
```

それでは検証してみましょう．検査対象の行がすべて問題なければ，

```
success: 検査テストベクタ数: 25
```

のように表示されるはずです．検証が済んだら実機での動作確認もしてみましょう．

3　例題 2：7 セグメントデコーダ付き 6 進カウンタ

次に，先ほどの 3 ビットカウンタを少し拡張する例題に取り組んでみましょう．変更点は以下の 2 点です．

- カウンタ値の 2 進数をそのまま出力するのではなく，7 セグメント LED を使って数字を表示する
- 0 から 5 まで数えたら 6 には行かずに 0 に戻す（6 進カウンタにする）

7 セグメント LED は 図 3 に示すように，個別に点灯・消灯を制御できる 7 つのセグメントをもつ LED 表示

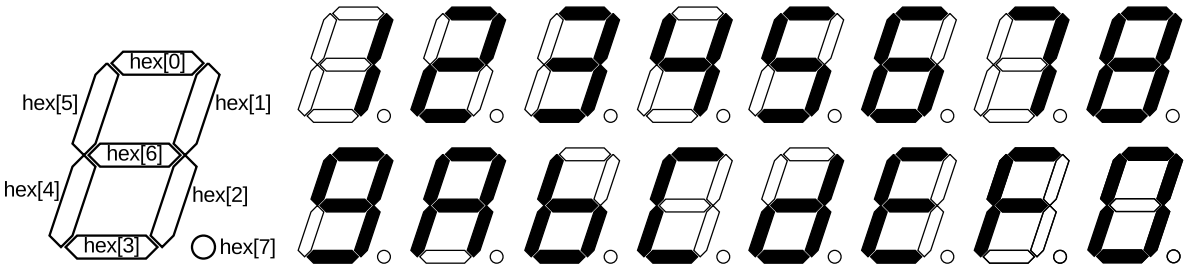


図 3　7 セグメント LED の構成と表示例

ソースコード 4　6 進カウンタ（digit_counter.sv）

```
1  `default_nettype none
2  module digit_counter
3    (
4     input wire        clk,
5     input wire        rst,
6     output wire [7:0] hex
7     );
8
9     reg [2:0]         count;
10
11    always_ff @(posedge clk) begin
12      if (rst)
13        count <= 3'd0;
14      else begin
15         if (count == 3'd5)
16           count <= 3'd0;
17         else
18           count <= count + 1'b1;
19      end
20    end
21
22    decode_7seg decode_7seg_inst(.din(count), .dout(hex));
23 endmodule
24
25
26 module decode_7seg
27   (
28    input wire [2:0]   din,
29    output logic [7:0] dout
30    );
31
32    always_comb begin
33       case (din)
34         3'd0:    dout <= 8'b00111111;
35         3'd1:    dout <= 8'b00000110;
36         3'd2:    dout <= 8'b01011011;
37         3'd3:    dout <= 8'b01001111;
38         3'd4:    dout <= 8'b01100110;
39         3'd5:    dout <= 8'b01101101;
40         default: dout <= 8'b00000000;
41       endcase
42    end
43 endmodule
44 `default_nettype wire
```

装置で，16 進数の数字 1 桁を表すことができます．実験ボードに使われているものは小数点の部分にも LED が配置されており，1 桁あたり合計 8 ビットの入力をもちます（この場合でも 8 セグメント LED とはいわず，習慣的に 7 セグメント LED と呼んでいます）．どのビットがどのセグメントに対応しているかは 図 3 の左側にあるとおりで，ここへ 1 を出力すると対応するセグメントの LED が点灯します．

3.1　階層記述を使った設計例

それでは早速ソースコード 4 に示した SystemVerilog による記述例を見てみましょう．モジュール名は digit_counter としています．入力ポートは 3 ビットカウンタのときと同様 clk（クロック）と rst （リセット）の 2 つです．出力ポートは 7 セグメント LED に接続された 8 ビット幅の hex です．

カウンタの値（2 進数）を LED の表示パターンに変換するためには組み合せ回路が必要になります．このような回路のことをデコーダと呼びます．ソースコード 4 ではこのデコーダのモジュールを「部品」として扱い，モジュールの中にモジュールを作る階層構造で表現しています．C 言語でいえば main 関数が他の関数を

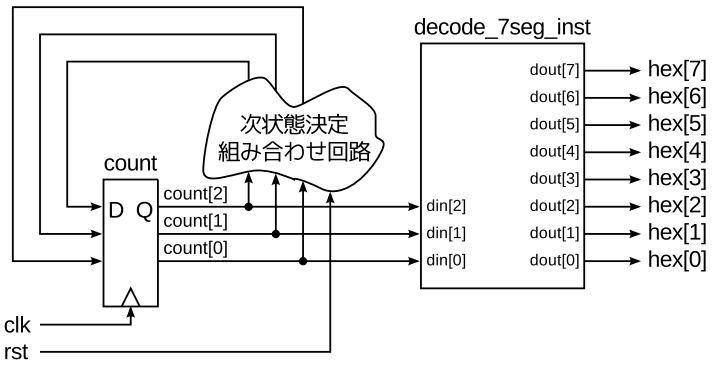


図 4　7 セグメントデコーダ付き 6 進カウンタの構成

呼び出すようなイメージです．大規模なハードウェアを設計する上で，モジュールの階層記述は重要な役割を果たします．この例では 2 行目から 23 行目までの digit_counter がカウンタのメインモジュール，26 行目から 43 行目までの decode_7seg が 7 セグメント LED 用のデコーダモジュールです．

図 4 にこのカウンタの構成を示します．先ほどの 3 ビットカウンタの例ではレジスタ cout の値がそのまま外部に出力されていましたが，この設計ではレジスタの値が組み合わせ回路のモジュール decode_7seg_inst を通過してから出力 hex に接続されています．この構成をイメージしながらソースコード 4 の記述を具体的に検討していきましょう．

■6 進カウンタの制御（11 行目～20 行目）　11 行目からの always_ff ブロックで，レジスタ count のクロックごとの動作を記述しています．12 行目からの if 節では，まずリセットがアクティブならカウンタ値を 0 にしています．14 行目からの else 節はリセットがかかっていないときの動作です．15 行目の if 文が 6 進カウンタのために新しく追加された論理で，現在のカウンタの値が 5 なら，次のクロックでは 0 に戻すようにしています（16 行目の代入文）．一方，それ以外のとき（カウンタの値が 5 ではないとき）には，カウンタの値を 1 増やします（18 行目の代入文）．

■サブモジュールのインスタンシエーション（22 行目）　先ほどの 3 ビットカウンタの例では，レジスタ count の値を assign 文によって出力に接続していました（ソースコード 1 の 19 行目）．一方，今回の例では 22 行目でデコーダのモジュールを 1 個インスタンシエーション（実体化）して，そこへ count を接続しています．この文法自体は，シミュレーションモジュールで検証対象モジュールをインスタンシエーションする記述とまったく同じです．decode_7seg がモジュール名，decode_7seg_inst が実体化したインスタンスに付けた名前です．28 行目と 29 行目を見ると分かるように，din と dout はそれぞれデコーダモジュールの入力と出力のポート名です．22 行目の記述では，レジスタ count をデコーダの入力ポート din に接続し，デコーダの出力ポート dout をそのままメインモジュールの出力ポート hex に接続しています．

■デコーダモジュールのインタフェース（26 行目～30 行目）　7 セグメント用のデコーダである decode_7seg モジュールは，3 ビット幅の入力 din と 8 ビット幅の出力 dout を持ちます．入力された 3 ビットの数を 7 セグメント LED で表示するために，どのセグメントを点灯すべきか 8 ビットで出力します．これは組み合わせ回路ですが，今回は出力 dout の論理を手続的代入を使って記述したかったので，dout は logic 型としています．

■case 文による条件記述（33 行目〜41 行目） デコーダの論理の記述は 32 行目から始まります．手続的代入を使って組み合わせ回路を書きたい場合には，always_comb ブロックを使うのでした．前回の加減算の例では always_comb ブロックの中で if 文を使いましたが，今回は case 文を使った例を示しています．SystemVerilog の case 文は C 言語の switch 文に似た形式で条件分岐を記述することができ，次のような構造をもちます．

```
case (式 0)
  式 1: 文 1;
  式 2: 文 2;
  式 3: 文 3;
  ...
  default: 文 n;
endcase
```

まず式 0 が式 1 と比較され，一致すれば文 1 が実行されます．一致しなければ次に式 0 と式 2 が比較され，一致すれば文 2 が実行されます．これも一致しなければ，さらに式 0 と式 3 が比較され，一致すれば文 3 が実行されます．このような処理が繰り返され，どれも一致しないと default に対応した文 *n* が実行されます．なお，default は省略することも可能です．

したがって，33 行目からの記述は，入力 din の値が 0 なら dout に $00111111_{(2)}$ が出力され，din が 1 なら dout に $00000110_{(2)}$ が出力され…という振る舞いに対応します．また，default を使って din が 0 から 5 までのどれにも該当しない場合には，dout に $00000000_{(2)}$ を出力して全セグメントを消灯するようにしています．

3.2 シミュレーションモジュール

シミュレーションモジュールの記述例をソースコード 5 に示します．3 ビットカウンタ用のシミュレーションモジュール（ソースコード 2）とほとんど同じですので，ほとんど説明の必要はないと思います．シミュレーションを実行すると，

```
# time=    40 count=  0 hex= 00111111
# time=    80 count=  1 hex= 00000110
# time=   120 count=  2 hex= 01011011
# time=   160 count=  3 hex= 01001111
# time=   200 count=  4 hex= 01100110
# time=   240 count=  5 hex= 01101101
# time=   280 count=  0 hex= 00111111
# time=   320 count=  1 hex= 00000110
...
```

のように，シミュレーション時間，カウンタの値（10 進表記），それに対応する 7 セグメント LED の点灯パターン（2 進表記）が各行に表示されます．また，シミュレーション時間 280 のところ見ると，カウンタ値が 5 の次に正しく 0 に戻っていることが確認できます．

3.3 C プログラムによるシミュレーションログの検証

今回のカウンタも動作自体は複雑ではありませんが，それでも 7 セグメント LED のパターンを目視で確認するのは厄介ですし，将来，複数桁の数字を表示するようになったらもっと大変です．そこで，7 セグメント LED 用の表示パターンの検証も含めて自動化できる C プログラムを作成してみましょう．

ソースコード 5　sim_digit_counter.sv

```
1  `default_nettype none
2  `timescale 1ns/1ps
3  // simulation module for digit_counter
4  module sim_digit_counter();
5     logic clk, rst;
6     wire [7:0] hex;
7
8     digit_counter digit_counter_inst
9       (.clk(clk), .rst(rst), .hex(hex));
10
11    initial begin
12       clk <= 1'b0;
13       repeat (25) begin
14        #20
15          clk <= 1'b1;
16        #20
17          clk <= 1'b0;
18          print();
19       end
20       $finish;
21    end
22
23    initial begin
24       rst <= 1'b1;
25     #30
26       rst <= 1'b0;
27    end
28
29    task print();
30       $write("time= %5d count= %2d hex= %b\n",
31          $time, digit_counter_inst.count, hex);
32    endtask
33 endmodule
34 `default_nettype wire
```

ソースコード 6 にプログラム例を示します．まず，7 行目から 11 行目で，各数字の LED の表示パターン（図 3 に示した情報）をアスキーアート的に表現して文字列配列 SegmentPattern に格納しています．これを 37 行目から 46 行目の for 文で 2 進数のビットパターンに自動変換し配列 hexPattern に格納しています．例えば数字の「3」に対応する LED の表示パターンは hexPattern[3] に格納されます．このような仕組みによって，LED 表示パターンを直接 2 進数で記述するのに比べてミスを減らすことができますし，表示パターンに変更があった際も容易に対応できます．64 行目では，この hexPattern を用いてログファイルから読み込んだ LED 表示パターンとその期待値とを比較しています．

他にも 3 ビットカウンタ用の検証プログラム（ソースコード 3）に比べて，柔軟性や保守性を改善するための変更をいくつか行っています．まず，51 行目です．ログファイルから読み込んだ行が検査対象の行かどうかを先頭の文字列を見て判断しますが，今までは strncmp 関数に比較する文字列パターンと文字数を直接与えていました（ソースコード 6 の 29 行目）．しかし，これではシミュレーションログの出力形式が変わると，文字列パターンとその文字数の両方を書き換えないといけません．そこで，文字数は strlen 関数で自動計算することにし，文字列パターンはマクロとして定義するようにしました．これによって，検査対象の行の先頭文字列が変更になっても，5 行目の TEST_VECTOR_PREFIX の定義を書き換えるだけで対応できるようになりました．

もうひとつの変更は，シミュレーションログから値を取り込む部分です．今までは各々のデータがログファイルの何列目から始まるかを直接記述していました．例えば，ソースコード 3 の 33 行目と 34 行目では，led の値は 19 列目，count の値は 34 列目といった具合でした．しかし，これではシミュレーションログの出力形式が変わるたびに，何の値が何列目に表示されるのかを数えて書き換えなければならず，1 列でも数え間違えると，正しい値を取り込めなくなってしまいます．そこで，今回は 56 行目の for 文で strtok 関数を使って入力行を空白文字で区切られたフィールドに分割し，各フィールドの先頭文字のアドレスをポインタ配列 sp に格納するようにしました．今回の検査対象行は，たとえば

```
# time=    40 count=  0 hex= 00111111
```

のようになっていますが，この場合は sp[0] が "#"，sp[1] が "time =", sp[2] が "40", sp[3] が "count=",

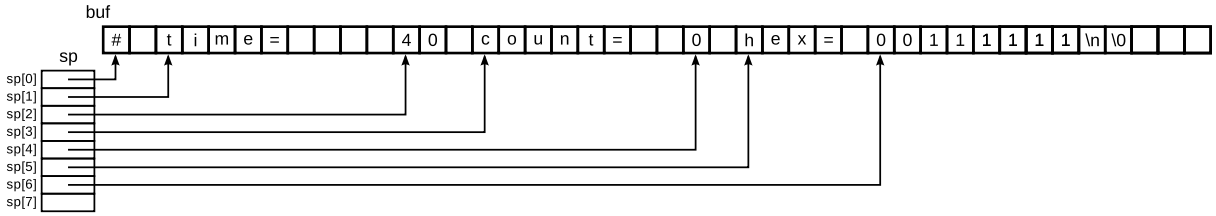


図 5　検査対象行（配列 buf）とポインタ配列 sp の関係

sp[4] が "0"，sp[5] が "hex="，sp[6] が "00111111"をそれぞれ指すポインタとなります（図 5）．60 行目と 61 行目では，この仕組みを使って，列数を指定せずに何番目のフィールドかだけを指定して値を取り込んでいます．

67 行目では次のクロックのカウンタの期待値を計算していますが，剰余演算のオペランドが 8 から 6 に変更されています．これはもちろん，今回の検証対象モジュールの仕様が 6 進カウンタに変更されたためです．

ソースコード 6　digit_counter 用の検証プログラム（check_sim_digit_counter.c）

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #define BUF_SIZE 256
5  #define TEST_VECTOR_PREFIX "# time="        /* 検査する行の先頭文字列 */
6
7  const char SegmentPattern[3][64] = {        /* 7-segment LED 点灯パターン */
8    " _       _   _       _   _   _   _   _       _       _  _ ",
9    "| |   |  _|  _| |_| |_  |_    | |_| |_| |_| |_  |   _| |_ |_ ",
10   "|_|   | |_   _|   |  _| |_|   | |_|  _| | | |_| |_ |_| |_ |  "
11 };
12
13 int main(int argc, char *argv[])
14 {
15   FILE *fp;
16   char buf[BUF_SIZE];
17   char *sp[BUF_SIZE];
18   int  i;
19   int  lineNumber = 1;      /* 行数カウンタ */
20   int  checkedLines = 0;    /* 検査済みテストベクタ数 */
21   int  expectedCount = 0; /* カウンタの期待値 */
22   int  hexPattern[16];      /* LED パターンの期待値 */
23
24   /* ログファイルが指定されていなければ終了 */
25   if (argc != 2) {
26     fprintf(stderr, "error: ログファイルが未指定\n");
27     exit(1);
28   }
29
30   /* ログファイルがオープンできなれば終了 */
31   if ((fp = fopen(argv[1], "r")) == NULL) {
32     fprintf(stderr, "error: ファイルオープン失敗: %s\n", argv[1]);
33     exit(1);
34   }
35
36   /* 文字列で与えた各数字に対するLED 点灯パターンを 2 進数に変換 */
37   for (i = 0; i < 16; i++) {
```

```
38      hexPattern[i]  = (SegmentPattern[0][4 * i + 1] != ' ')? 0x01: 0; /* seg0 */
39      hexPattern[i] |= (SegmentPattern[1][4 * i + 2] != ' ')? 0x02: 0; /* seg1 */
40      hexPattern[i] |= (SegmentPattern[2][4 * i + 2] != ' ')? 0x04: 0; /* seg2 */
41      hexPattern[i] |= (SegmentPattern[2][4 * i + 1] != ' ')? 0x08: 0; /* seg3 */
42      hexPattern[i] |= (SegmentPattern[2][4 * i    ] != ' ')? 0x10: 0; /* seg4 */
43      hexPattern[i] |= (SegmentPattern[1][4 * i    ] != ' ')? 0x20: 0; /* seg5 */
44      hexPattern[i] |= (SegmentPattern[1][4 * i + 1] != ' ')? 0x40: 0; /* seg6 */
45      hexPattern[i] |= (SegmentPattern[2][4 * i + 3] != ' ')? 0x80: 0; /* seg7 */
46    }
47
48    /* ログファイルから 1行ずつ読み期待値と比較 */
49    while (fgets(buf, BUF_SIZE, fp) != NULL) {
50      /* 指定された文字列で始まる行だけ処理 */
51      if (strncmp(buf, TEST_VECTOR_PREFIX, strlen(TEST_VECTOR_PREFIX)) == 0) {
52        int count, hex;
53
54        /* 空白で区切られた各文字列の切り出し */
55        i = 0;
56        for (sp[i] = strtok(buf, " "); sp[i] != NULL; sp[++i] = strtok(NULL, " "))
57          ;
58
59        /* 出力値の読み込み */
60        count = atoi(sp[4]);
61        hex = strtol(sp[6], NULL, 2);
62
63        /* 出力と期待値を比較し異なれば終了 */
64        if (count == expectedCount && hex == hexPattern[expectedCount]) {
65          checkedLines++;
66          /* 次のカウンタ期待値の計算 */
67          expectedCount = (expectedCount + 1) % 6;
68        }
69        else {
70          fprintf(stderr, "error: %d 行目: ", lineNumber);
71          fprintf(stderr, "count=%d count 期待値=%d ", count, expectedCount);
72          fprintf(stderr, "hex=0x%02x hex 期待値=0x%02x\n",
73                  hex, hexPattern[expectedCount]);
74          exit(1);
75        }
76      }
77      lineNumber++;
78    }
79
80    /* 検査したテストベクタ数が 0なら終了 */
81    if (checkedLines == 0) {
82      fprintf(stderr, "error: テストベクタが見つかりませんでした\n");
83      exit(1);
84    }
85
86    /* 検査したテストベクタ数を表示して正常終了 */
87    printf("success: 検査テストベクタ数: %d\n", checkedLines);
88    return (0);
89  }
```