情報工学実験 II
第 3 回： D フリップフロップとカウンタ

柴田裕一郎・松尾堅太郎・元島晃伸

shibata@cis.nagasaki-u.ac.jp
情報工学コース

2018 年 12 月 20 日

今日の概要： 順序回路の設計（カウンタ編）

1. 基本事項の復習
   - 組み合わせ回路と順序回路
   - D フリップフロップ
2. カウンタの設計
   - 3 ビットカウンタ
   - 7 セグメント LED 用デコーダ付き 6 進カウンタ
3. 新しい SystemVerilog の文法
   - reg 変数
   - always_ff ブロックと posedge
   - 連接演算
   - repeat 文
   - case 文

組み合わせ回路と順序回路

- 組み合わせ回路
  - 出力が入力だけによって決まる回路
  - 入力の値が決まれば出力の値も一意に決まる
- 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異なる値を出力

組み合わせ回路と順序回路の例

- 加減算器（組み合わせ回路）
  - 入力： 2 つの数値（ain，bin）と加算・減算の制御入力（sub）
  - 出力： 演算結果
  - 入力が同じなら出力結果も必ず同じ
- コーヒーの自動販売機（順序回路）
  - 入力： ボタン
  - 状態： お金がいくら入っているか
  - 出力： コーヒーとお釣り

$\Rightarrow$ 順序回路では状態を『記憶』する仕組みが必要になる

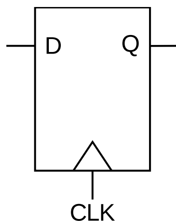
記憶素子： ラッチとフリップフロップ

- ディジタル回路の基本的な記憶素子
  - SR ラッチ
  - D ラッチ
  - JK フリップフロップ
  - T フリップフロップ
  - D フリップフロップ
  - ...

$\Rightarrow$ ほとんど D フリップフロップ（D-FF）を用いる

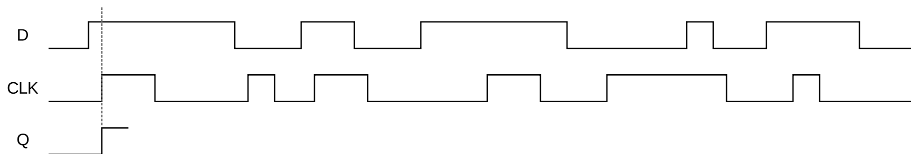
D フリップフロップの動作

- D-FF はエッジトリガ動作
- クロックの立ち上がりエッジ（‘0’ から ‘1’ への変化時）
  - 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



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

練習： D-FF のタイミングチャート



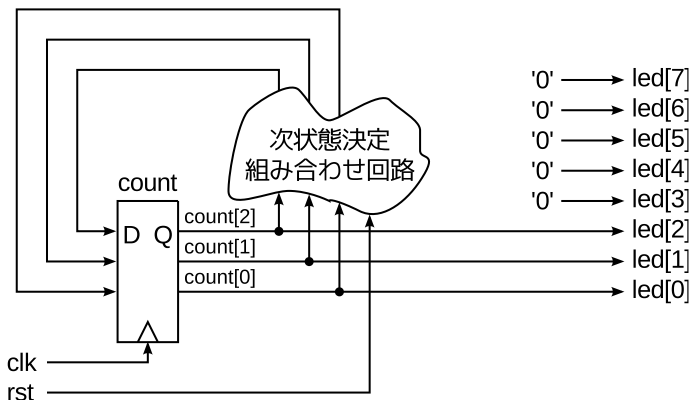
クロックとリセット

- クロック
  - 0 と 1 を周期的に繰り返す発振子を利用
  - 実験ボードには周波数 24 MHz の発振子が搭載
- リセット
  - 電源投入直後の D-FF の出力は 0 になるか 1 になる分からない
    - 不定値
    - SystemVerilog では「x」で表す
  - 初期化用の「リセット入力」を D-FF に設ける
    - アクティブ High / アクティブ Low
    - 同期リセット / 非同期リセット

例題 1： 3 ビットカウンタ

- 入出力ポート
  - `clk`: クロック入力
  - `rst`: アクティブ High の同期リセット入力
  - `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，$000_{(2)}$，$001_{(2)}$，$010_{(2)}, \ldots$ カウントアップ
  - そのまま 3 個の LED に出力
- LED 出力は 8 ビットあるが，上位 5 ビットは常に 0
- カウンタの値が $111_{(2)}$ まで行ったら次は $000_{(2)}$ に戻す

3 ビットカウンタの構成



- 状態を保持する FF
  - 状態は「現在のカウンタの値」⇒ 3 つの D-FF が必要
- 次状態を決める組み合わせ回路
  - 「1 を足す」回路
  - $111_{(2)}$ まで行ったりリセットがかかったら $000_{(2)}$ に戻す

順序回路記述のための SystemVerilog 文法

- レジスタ変数
  - reg 型
- always_ff ブロック
  - クロックが立ち上がるたびに begin〜end が 1 度実行
  - クロックの立ち上がりは @(posedge ...) で記述
  - ブロック内の文は逐次的に実行
  - reg 型変数への手続き型代入
  - ブロック内で代入する変数は，ブロックの外では代入不可
- 連接演算
  - 複数の信号線をカンマ（「,」）で区切って「{」と「}」で囲む
  - それらを並べて 1 つのバス信号線を作ることを意味
  - 左に書いた信号線が上位になる

3 ビットカウンタ（binary_counter.sv）

```
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
```

シミュレーションモジュール

- 2 つの initial ブロック
  - クロックとリセットへの入力
  - それぞれ並列に動作
- repeat 文
  - 指定回数の繰り返し
- シミュレーション時間
  - `$time` システムタスクで取得
- 内部信号の階層指定
  - インスタンス名.信号名

sim_binary_counter.sv (1/2)

```
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;
```

sim_binary_counter.sv (2/2)

```
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
```

シミュレーションの実行

```
% make sim
```

```
...
# 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
...
```

検証用 C プログラム

- 1 行ずつログファイルを読み込んで値をチェック
  - 基本的な流れは前回のものと同じ
- 1 行チェックするごとに次のカウントの期待値を計算
  - 変数 `expectedCount`
- `led` と `count` がそれぞれ `expectedCount` と等しいかチェック

check_sim_binary_counter.c (1/2)

```
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  }
```

check_sim_binary_counter.c (2/2)

```
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);
```

検証の実行

```
% make check
```

```
...
success: 検査テストベクタ数: 25
```

のように最後に「success」が表示されれば OK

実機走行確認

```
% make
```

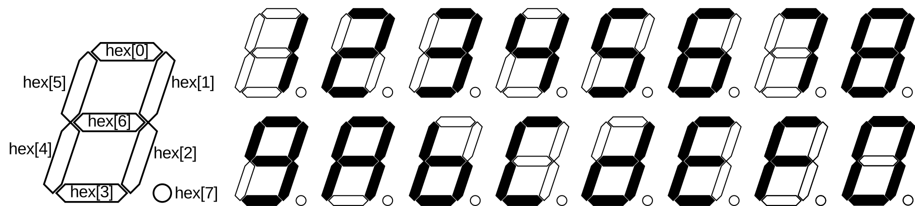
```
% make config
```

- 3 ビットの LED の点灯動作
- クロックは確認しやすいように 24 MHz を分周して利用
  - マトリックスキーの左下とその隣のキーで変更可
- リセットは「System Reset」のボタン（SW10）

例題 2： 7 セグメントデコーダ付き 6 進カウンタ

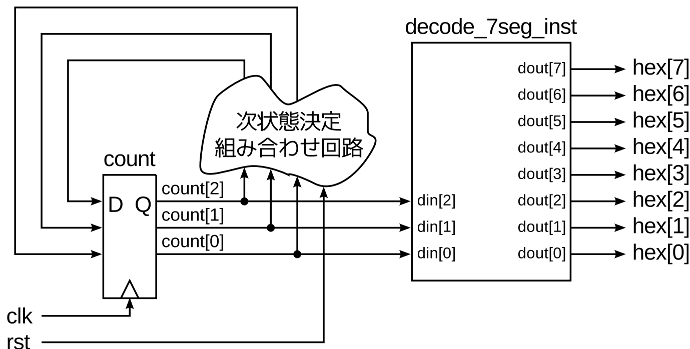
- 3 ビットカウンタに以下の変更を加えたものを実装
  - カウンタ値の 2 進数をそのまま出力するのではなく，7 セグメント LED を使って数字を表示する
  - 0 から 5 まで数えたら 6 には行かずに 0 に戻す（6 進カウンタにする）
- 入出力ポート
  - `clk`: クロック入力
  - `rst`: アクティブ High の同期リセット入力
  - `hex`: 7 セグメントの LED への出力端子（8 ビット）

7 セグメント LED



- 「1」を表示したいときは $00000110_{(2)}$ を出力すればよい

7 セグメントデコーダ付き 6 進カウンタの構成



- カウンタの値（2 進数）を LED の表示パターンに変換する組み合せ回路が必要
  - デコーダ
- このデコーダのモジュールを「部品」として扱う
  - モジュールの中にモジュールを作る（階層構造）

SystemVerilog 記述

- 6 進カウンタの制御
  - 5 になったら次は 0 にする
- デコーダモジュール（decode_7seg）のインタフェース
  - 入力： din （3 ビット）
  - 出力： dout （8 ビット）
- サブモジュールのインスタンシエーション（実体化）
  - シミュレーションモジュールでやっているのと同様
  - インスタンス名は decode_7seg_inst としている

digit_counter.sv (1/2)

```
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));
```

case 文

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

- 多分岐条件を記述できる手続き的構文
- まず式 0 が式 1 と比較し，一致すれば文 1 を実行
- 一致しなければ式 0 と式 2 を比較し，一致すれば文 2 を実行
- これも一致しなければ，さらに式 0 と式 3 を比較し…（以下略）
- どれも一致しなかったら default に対応した文 n を実行
- default は省略することもできる

digit_counter.sv (2/2)

```
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
```

sim_digit_counter.sv (1/2)

```
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;
```

sim_digit_counter.sv (2/2)

```
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
```

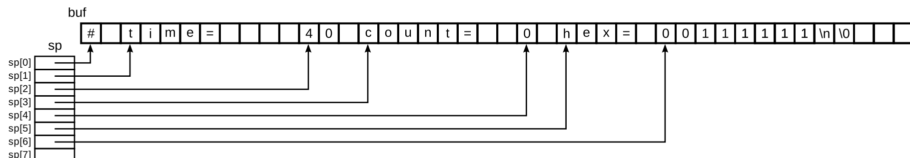
シミュレーションの実行

```
% make sim
```

```
...
# 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
# time=   360 count=  2 hex= 01011011
# time=   400 count=  3 hex= 01001111
...
```

検証プログラム

- LED 点灯パターンを文字列配列から自動変換
- 検査対象の先頭文字列をマクロ定義
- 空白で区切られた文字列の切り出し
  - 各フィールドの先頭アドレスをポインタ配列 sp に 格納



check_sim_digit_counter.c (1/3)

```
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 */
```

check_sim_digit_counter.c (2/3)

```
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         ;
```

check_sim_digit_counter.c (3/3)

```
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) {
```

やってみよう

- 検証

```
% make check
```

- 論理合成・配置配線

```
% make
```

- コンフィギュレーション

```
% make config
```

練習： 10 進カウンタに変更

- SystemVerilog 設計ファイル（digit_counter.sv）
  - digit_counter モジュール
    - レジスタ変数 count を 3 ビットから 4 ビットに変更
    - 15 行目の if 文の条件を変更
  - decode_7seg モジュール
    - 入力 din を 3 ビットから 4 ビットに変更
    - case 文に LED パターンを追加
- 検証プログラム（check_sim_digit_counter.c）
  - 67 行目の expectedCount の計算式を変更

課題 1： 24 時間タイマの実装

- クロック信号の立上りに同期して「00 時 00 分 00 秒」から 1 クロックごとにウントアップする（「23 時 59 分 59 秒」の次は「00 時 00 分 00 秒」）

  hex5: 10 時の桁の数字を表す 7 セグ LED 出力（8 ビット）
  hex4: 1 時の桁の数字を表す 7 セグ LED 出力（8 ビット）
  hex3: 10 分の桁の数字を表す 7 セグ LED 出力（8 ビット）
  hex2: 1 分の桁の数字を表す 7 セグ LED 出力（8 ビット）
  hex1: 10 秒の桁の数字を表す 7 セグ LED 出力（8 ビット）
  hex0: 1 秒の桁の数字を表す 7 セグ LED 出力（8 ビット）

- 入力はクロック（clk）とリセット（rst）
  - リセットがかかったら「00 時 00 分 00 秒」に戻る
- モジュール名は timer とする
- シミュレーションは少なくとも 86500 クロック行って検証する

課題 2： 12 時間タイマの実装

- クロック信号の立上りに同期して「00 時 00 分 00 秒」から 1 クロックごとにウントアップする（「11 時 59 分 59 秒」の次は「00 時 00 分 00 秒」）

  `hex5:` 10 時の桁の数字を表す 7 セグ LED 出力（8 ビット）
  `hex4:` 1 時の桁の数字を表す 7 セグ LED 出力（8 ビット）
  `hex3:` 10 分の桁の数字を表す 7 セグ LED 出力（8 ビット）
  `hex2:` 1 分の桁の数字を表す 7 セグ LED 出力（8 ビット）
  `hex1:` 10 秒の桁の数字を表す 7 セグ LED 出力（8 ビット）
  `hex0:` 1 秒の桁の数字を表す 7 セグ LED 出力（8 ビット）

- 入力はクロック（`clk`）とリセット（`rst`）
  - リセットがかかったら「00 時 00 分 00 秒」に戻る
- モジュール名は `timer` とする
- シミュレーションは少なくとも 86500 クロック行って検証する

設計の方針（timer.sv）

- 各桁に対応するレジスタを定義
  - count5
  - count4
  - count3
  - count2
  - count1
  - count0
- 各レジスタの動作を 6 つの always_ff ブロックで記述
  - それぞれの always_ff ブロックは並列に動作
- デコーダモジュールも 6 つインスタンシエーションする
  - 同じインスタンス名を付けないように注意する