再構成型アーキテクチャ特論 (3)

2016年度後学期 長名

osana@eee.u-ryukyu.ac.jp

前回の復習

- Verilog-HDL で組み合わせ回路を書く
  - 入出力ポートと中間の信号宣言 (wire)
  - 継続代入 (assign): 組み合わせ回路
- ちゃんと動かすためには合成される回路を想像することが重要
  - 演算器、マルチプレクサ、誘起遅延と伝播遅延、条件の完全性

論理回路の残り半分

* 組み合わせ回路は書けるようになりました
* あとは順序回路が書ければ完璧ですね？

順序回路

- フリップフロップと論理ゲートの組み合わせ
  - フロップフロップは事実上 D-FF だけ、あとは普通の論理ゲート
  - クロック信号が必要
    - なるべく単相同期回路にする: すべてのFFに同じくクロックを供給

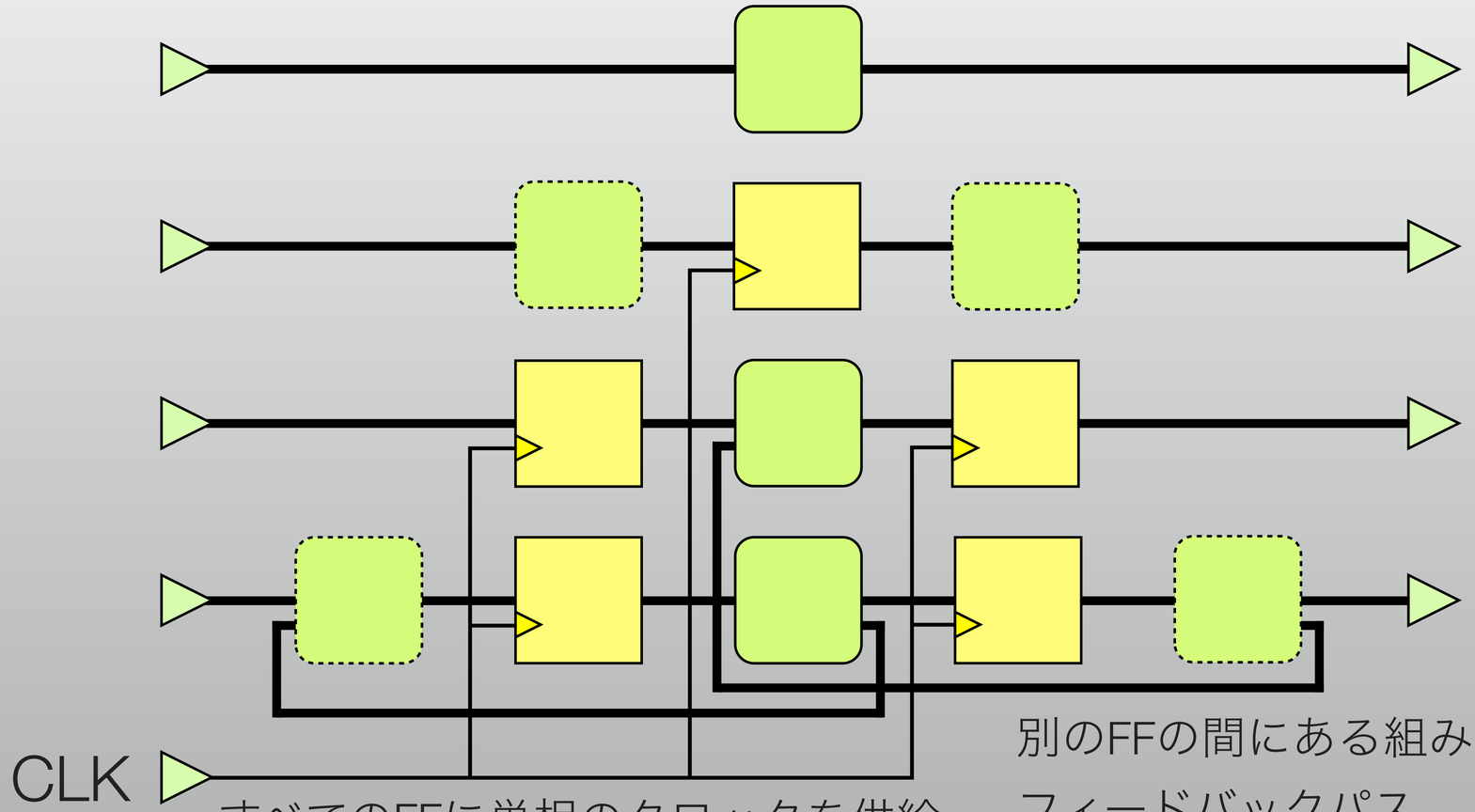
忘れてほしいこと

- こういうのは教科書の中だけのおとぎ話です
  - ひとつの大きなステートマシンで回路全体が動く
  - 状態遷移を一枚の紙に描ける
- その程度では実用的なものは作れません

忘れないでほしいこと

* 組み合わせ論理と D-FF でなんでもつくるのが実際の設計

デジタル回路のすべて



入力から出力まで組み合わせ回路

FFの前後 (あるいは片方) に
組み合わせ回路

FFの間に組み合わせ回路

FFの間と前後 (あるいは片方) に
組み合わせ回路

別のFFの間にある組み合わせ回路への
フィードバックパス

CLK

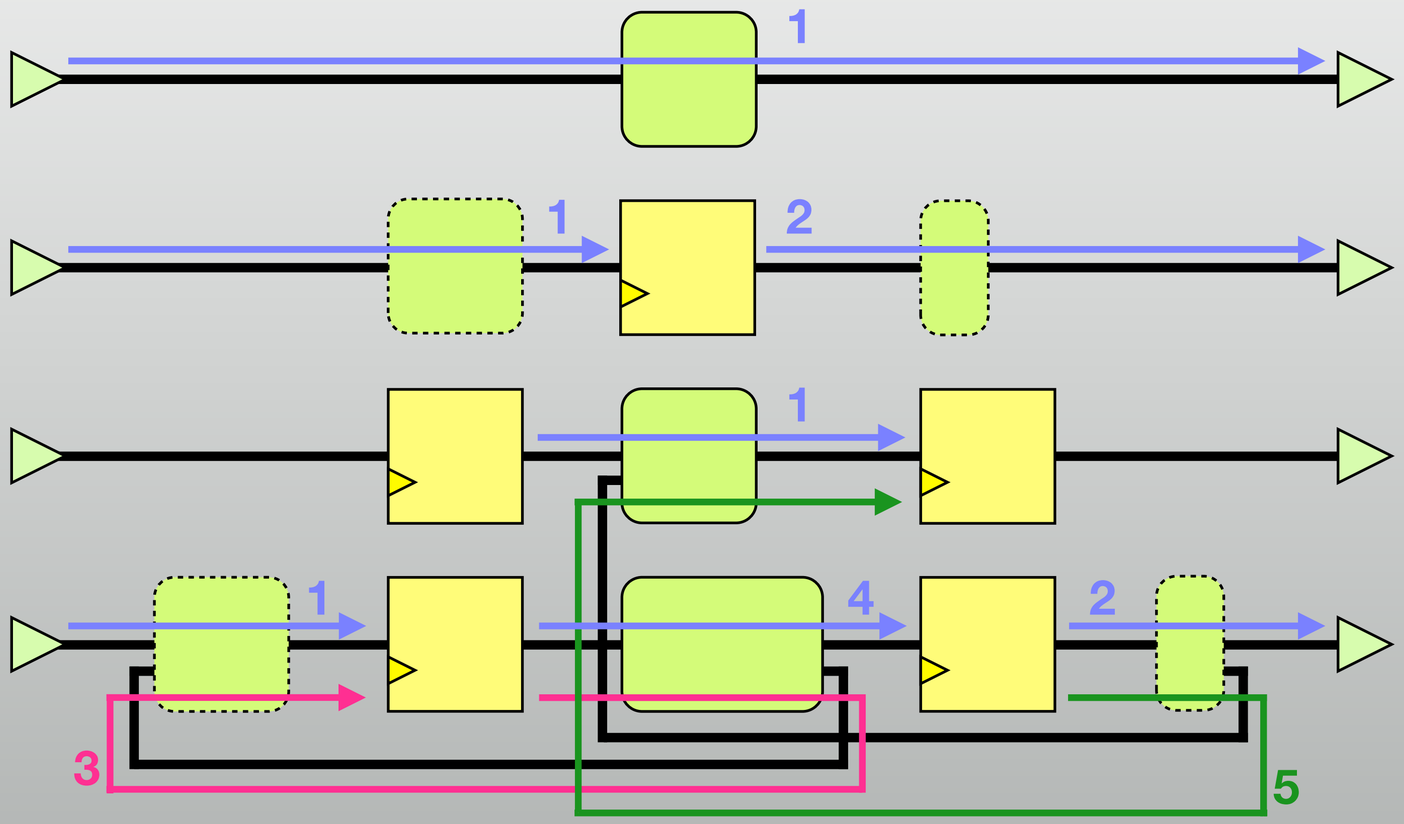
すべてのFFに単相のクロックを供給

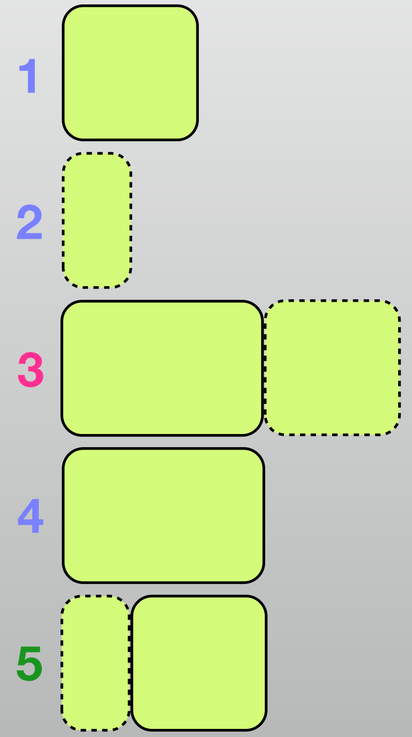
I/O Pad

組み合わせ回路

FF

パス遅延

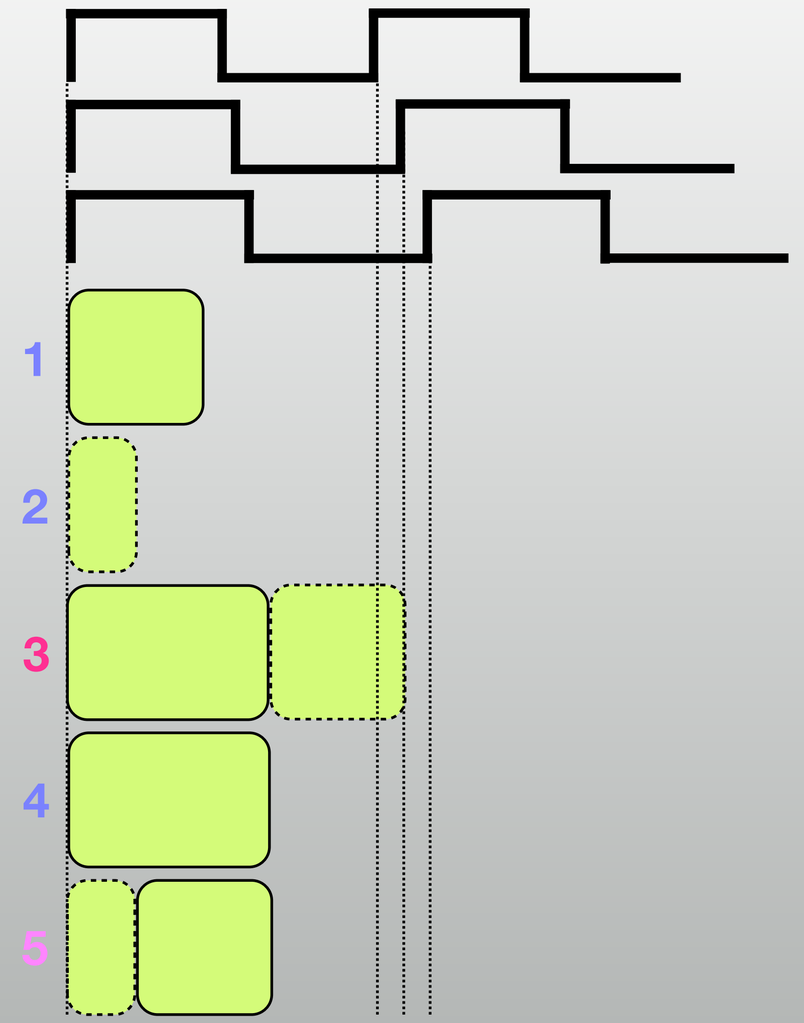




I/O Pad および FF を起点・終点とするパスの遅延を考える

単相同期回路

- 同じクロックの立ち上がりで
  - 入力信号をキャプチャ
  - 内部のFFを駆動
  - 出力はそれに間に合うように
- 一番長い遅延が周波数を律速



代入文と信号

- 継続的代入文: assign, wire
  - wire 型の変数を使う　（ここまでは前回やりました）
- 手続的代入文: **always, initial**
  - **reg** 型の変数を使う

wire と reg

- すべての信号(変数)は必ずregまたはwire宣言されていなければならない
- 手続代入文の中で代入される、左辺の信号は必ず reg 宣言されていなければならない
- 入出力宣言されている、**または1bitの信号は**wire宣言を省略してもよい
  - wire/reg は回路の何かというより、上記のルールによる文法的事項

ブロッキング vs ノンブロッキング

- ブロッキング代入演算子 “=“
  - 完了するまで次の文は実行されない: ソフトウェアでは普通
- ノンブロッキング代入演算子 “<=“
  - A <= B; B <= A; は同時に実行され、値が入れ替わる
  - ハードウェア的にはこれが普通

assign 文とブロッキング代入

- assign では = を使う: つまりブロッキング代入
  - 個々のassign文は独立しているのでassign文同士はブロックしない
- 通常のassign文では遅延0で代入される
  - 代入までに Δt の遅延を設定すると、Δt の間ブロックする
  - その間に右辺の値が変化しても無視される (遅延についてはまた後で)

C の {} と Verilog の begin - end

- ifなどの制御文の対象は文1つ
  - 対象が複数ならブロックに:　C では { から } まで
  - Verilog では begin から end までがブロック
    - この中にノンブロッキング代入文を並べる
    - ブロッキング代入文は基本的に使え (わ) ない

8bit カウンタの例

- クロックの立ち上がりで動作
  - 同期回路はこれが基本
  - always @ (posedge CLK)
- これだと初期値が不定

```
module counter8 ( input  wire       CLK,
                  output wire [7:0] COUNT );

  reg [7:0] CNT;

  always @ (posedge CLK)
    CNT <= CNT + 1;

  assign COUNT = CNT;

endmodule
```

リセット付き 8bit カウンタの例

- if (RST)
  - リセット信号RSTがhigh か
  - HighならCNTを0にする

```
module counter8 ( input  wire       CLK, RST,
                  output wire [7:0] COUNT );

  reg [7:0] CNT;

  always @ (posedge CLK) begin
    if (RST)
      CNT <= 0;
    else
      CNT <= CNT+1;
  end

  assign COUNT = CNT;

endmodule
```

LEDくるくる

- 肉眼で見えるスピードで　○→●→●→●　●→○→●→● …
- クロックは 100MHz
  - 2^{20}=1M (1,048,576), 2^{24}=16M (16,777,216)
- 24bit のカウンタがあれば 6Hz の信号が作れる
- LED の数と同じ FF を使って点灯状態を管理

まずカウンタ

- 24bit
  - 全部1になるのが6Hz
  - このときに1クロックだけ信号が出るようにする

```
reg [23:0] CNT;
wire STROBE = &CNT;

always @ (posedge CLK) begin
   if (RST)
      CNT <= 0;
   else
      CNT <= CNT+1;
end
```

LEDを点灯させる

* 初期状態は左端点灯
  * STROBEで右シフト

```
reg [23:0] CNT;
reg [15:0] LED;
wire STROBE = &CNT;

always @ (posedge CLK) begin
   if (RST) begin
      CNT <= 0;
      LED <= 16'b1000_0000_0000_0000;
   end else begin
      CNT <= CNT+1;
      if (STROBE)
        LED <= {LED[0], LED[15:1]}
  end
end
```

ちゃんとポートを接続する

* CLK は水晶発振器
* RST は押しボタン
* LED は output reg で出力ポートとして宣言

```
module led_test
   ( input  wire       CLK, RST,
     output reg [15:0] LED );

  reg [23:0] CNT;
  wire STROBE = &CNT;

  always @ (posedge CLK) begin
    if (RST) begin
      CNT <= 0;
      LED <= 16'b1000_0000_0000_0000;
    end else begin
      CNT <= CNT+1;
      if (STROBE)
        LED <= {LED[0], LED[15:1]}
    end
  end
endmodule
```

ここまでの文法的まとめ

- 入出力は暗黙の wire になるが、バグを防ぐためにちゃんと書くべき
- always @ (posedge CLK): FFはクロックで駆動する
  - ノンブロッキング代入
  - if 文とかは always ブロックの内側で使える
    - assign 文のように条件付き代入も使えます

ここまでの慣用的記述方法まとめ

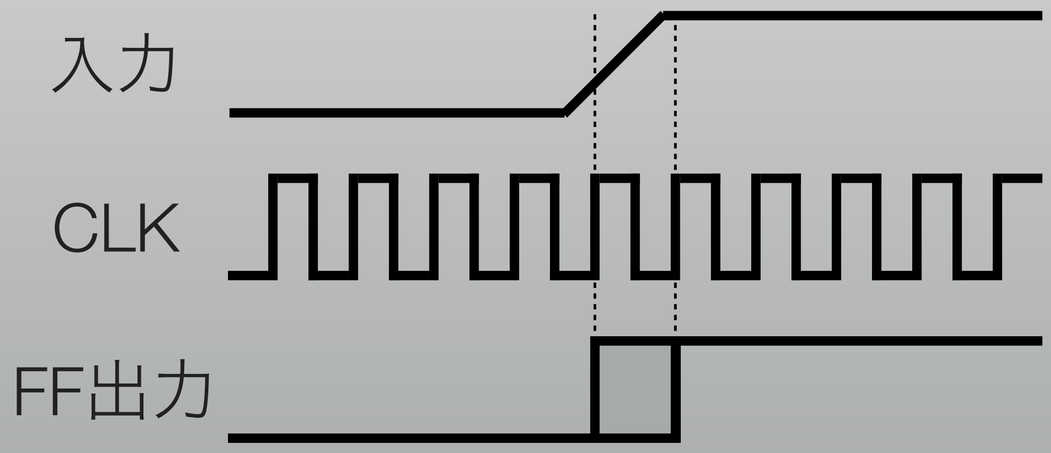
- if (RST)
  - リセット時とそれ以外の記述を分ける
  - 必ずしもすべての reg をリセットで初期化する必要はない
    - リセット信号のファンアウトが大きくなるとタイミングに影響

スイッチを使う: 困ったことが２つ

* メタステーブル
  * レベルが不確定な状態で値を取り込むと発生: 非同期入力で注意が必要
* チャタリング
  * 接点が機械的な振動で切り替え直後にon/offを繰り返す
  * 高速なクロックから見ると何度も切り替わったようにみえる

メタステーブルの回避

- ゲートの出力が H でも L でもない状態に落ちるのがメタステーブル
  - 確率は高くない (ように論理素子が作られている)
- FF を直列に入れて回避
  - 2段あれば事実上完全
  - 非同期入力には必ず入れるべき



Double Flopping

* FFふたつなので簡単
  * たいていちゃんと合成される

```
module double_flop
    ( input  CLK, IN,
      output OUT );

  reg FF1, FF2;

  always @ (posedge CLK) begin
    FF1 <= IN;
    FF2 <= FF1;
  end

  assign OUT = FF2;

endmodule
```

チャタリング

- きれいに off → on → off とはならない
  - 100MHz でサンプリングすれば何度も on/off する
  - 「指で押す」ことを前提にして波形を整形
    - 一度 on になったらしばらく on を無視する、とか

チャタリング除去の例

- 押したらパルスを1クロック
  - 最大 5Hz
  - 2^{25}=33,554,432
- リセットが必要なのが問題
  (ちゃんと書けば解決できる)

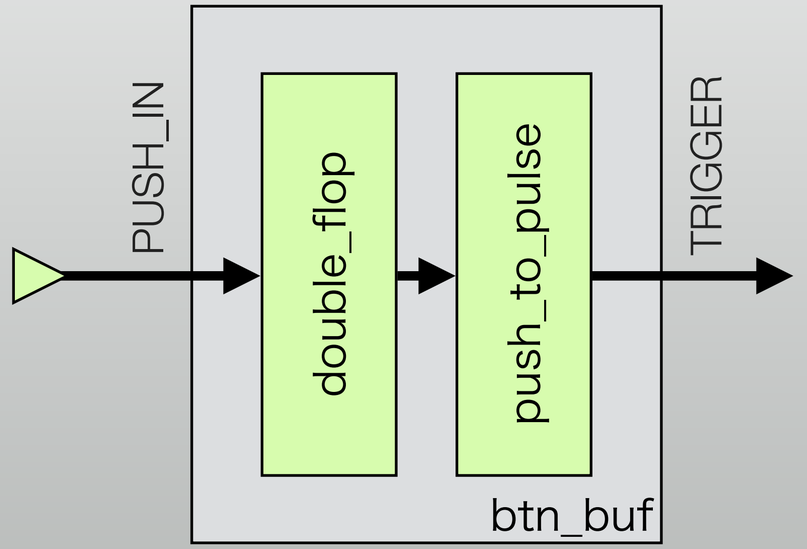
```
module push_to_pulse
   ( input  wire CLK, RST, BTN,
     output reg  TRIG );

  reg [24:0] CNT;

  always @ (posedge CLK) begin
    if (RST) begin
      CNT  <= 0;
      TRIG <= 0;
    end else begin
      if (CNT==0) begin
        if (BTN) begin
          CNT  <= 20_000_000;
          TRIG <= 1;
        end
      end else begin
        CNT  <= CNT-1;
        TRIG <= 0;
      end
    end
  end
endmodule
```

モジュール化

* ふたつのモジュールをひとつに
  * スイッチごとに接続したい
  * まとめておけば便利



サブモジュール

- インスタンス・ポート・信号
  - モジュール名
  - インスタンス名
  - .ポート
  - (信号)

```
module push_filter
   ( input  CLK, RST, BTN,
     output TRIGGER );

  wire BTN_INT;

  double_flop df
    (.CLK(CLK), .IN(BTN), .OUT(BTN_INT));

  push_to_pulse ptp
    (.CLK(CLK), .RST(RST), .BTN(BTN_INT),
     .TRIG(TRIGGER));

endmodule
```

LEDくるくるへの応用

* ボタンを押すとひとつ進む

```
module led_test
   ( input             CLK, RST, BTN,
     output reg [15:0] LED );

  wire STROBE;
  push_filter pf
    ( .CLK(CLK), .RST(RST), .BTN(BTN),
      .TRIGGER(STROBE) );

  always @ (posedge CLK) begin
    if (RST) begin
      LED <= 16'b1000_0000_0000_0000;
    end else begin
      if (STROBE)
        LED <= {LED[0], LED[15:1]}
    end
  end
endmodule
```

モジュール化のよい点

* 検証済みのモジュールを再利用することで設計を容易化
* 複製が簡単: スイッチの数だけ同じ回路を作る、など

すべては心がけ次第

- モジュールのインタフェイスは極力シンプルに
- どういうポートがどういう信号を取り扱うのかをよく考える
  - 1クロックのストローブ信号とかが大事
    - 何クロックサイクルも継続する信号は取り扱いにくい

もうひとつの制御構造: case

- if 文は優先度つきで評価される
  - 多重になると遅い
    - if-else-if-else-if-else…
  - case 文は並列に評価される
    - ステートマシンを作るのに便利

LEDを左右ボタンで制御

* 左右のボタンで点灯するLEDを移動
* 左右の端ではそれ以上向こうにいかない

外側とつながる部分を先に

* 左右のスイッチはpush_filterへ
  * これがないと100M回/秒移動
  * 「1クロックの出力」は大事！

```
module led_test2
   ( input  CLK, RST, SW_L, SW_R,
     output reg [15:0] LED );

  wire LEFT, RIGHT;

  push_filter pf_l
    (.CLK(CLK), .RST(RST), .BTN(SW_L),
     .TRIGGER(LEFT));

  push_filter pf_r
    (.CLK(CLK), .RST(RST), .BTN(SW_L),
     .TRIGGER(RIGHT));

  // ここは次のスライドで
endmodule
```

case - endcase

- LEDの状態
  - 次のクロックでどうするか
  - 現在のLEDとスイッチの状態で決定する

```
always @ (posedge CLK) begin
  if (RST) begin
    LED <= 16’b1000_0000_0000_000;
  end else begin
    case (LED)
      1’b1000_0000_0000_0000:
        if (RIGHT) LED <= {LED[0],LED[15:1]};
      1’b0000_0000_0000_0001:
        if (LEFT) LED <= {LED[14:0],LED[15]};
      default: begin
        case ({LEFT,RIGHT})
          2’b01: LED <= {LED[0],LED[15:1]};
          2’b10: LED <= {LED[14:0],LED[15]};
          default: LED <= LED;
        endcase
      end
    endcase
  end
end
```

まとめ

- always 文と reg による順序回路の記述
  - 制御構造は if と case が使える
- モジュールとサブモジュール