Reconfigurable Architecture (3)

osana@eee.u-ryukyu.ac.jp

Review for last week:

* Combinational logic in Verilog HDL
  * Ports and intermediate signals (wire)
  * Continuous assignment (assign): combinational logic
* Always think about the resulting hardware
  * Arithmetic units, multiplexors, contamination/propagation delay, mutually exclusive conditions …

The other half of logic circuit

* We’ve done for combinational logic
* Sequential logic is the rest half

Sequential logic

* Flip-Flops + logic gates
  * D-FF (delay FF) + combinational logic
  * Clock signal required
    * Single-phase synchronous circuit is easier and better: all FFs are driven by single clock signal

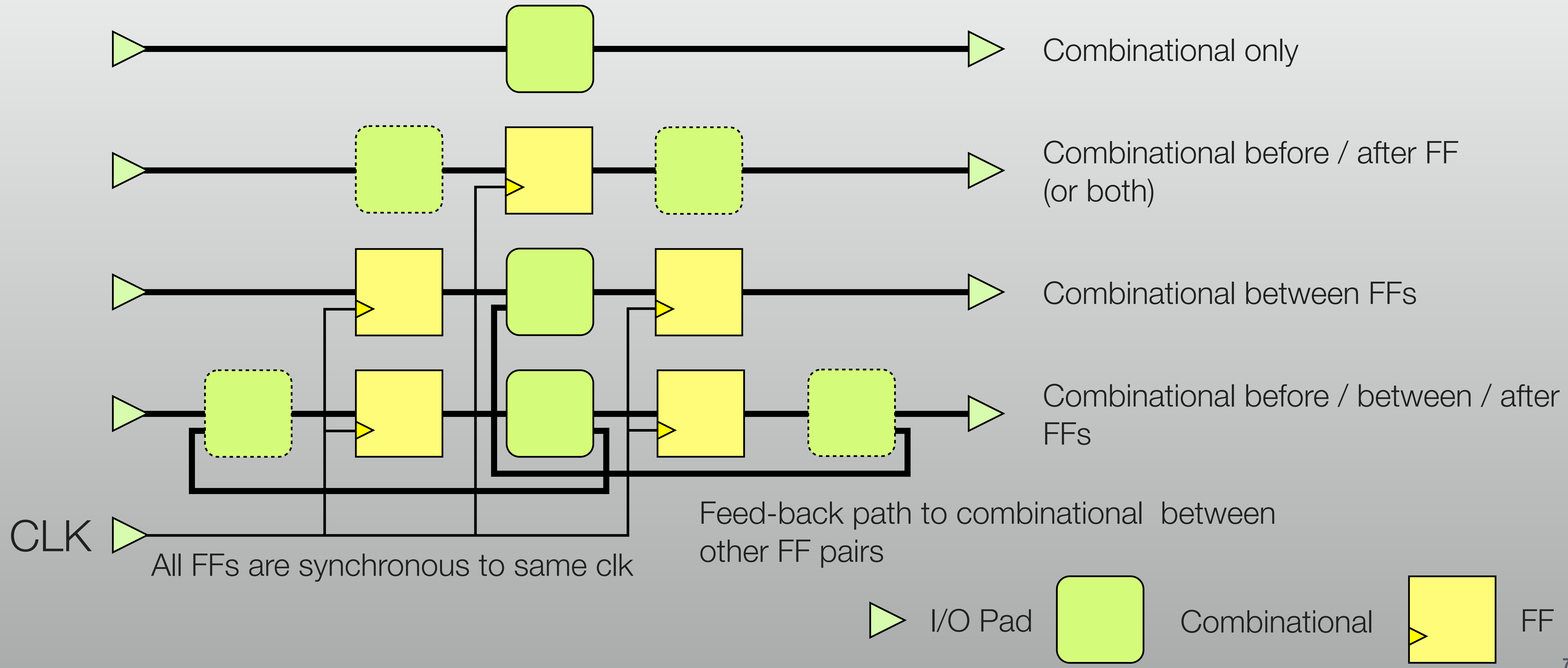
Please forget:

* Fantasy in many textbooks
    * A single, large finite-state machine controls everything
    * or, all state transition of the system can be drawn in A4 paper
* Reality is far more complicated

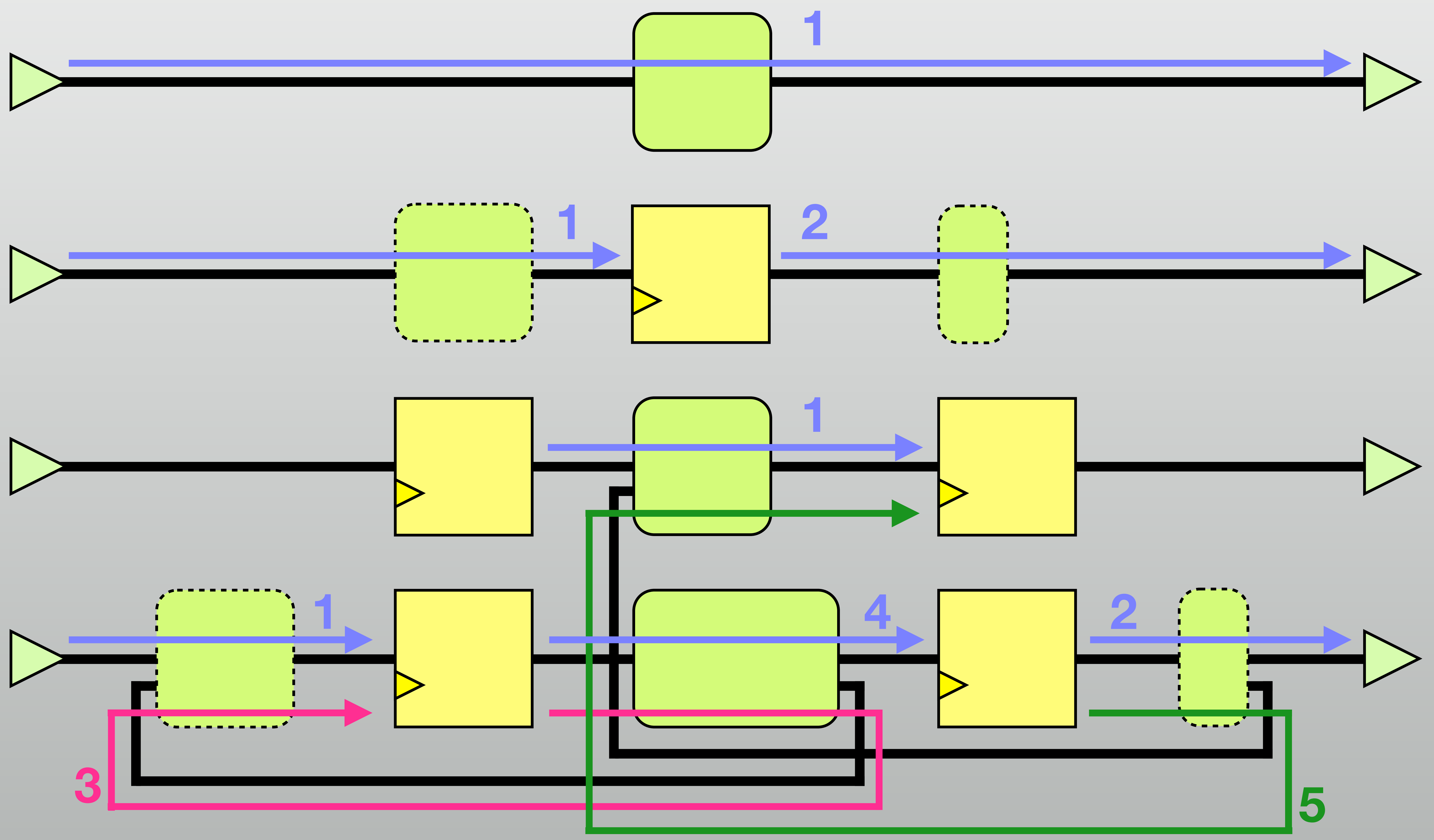
Please don’t forget:

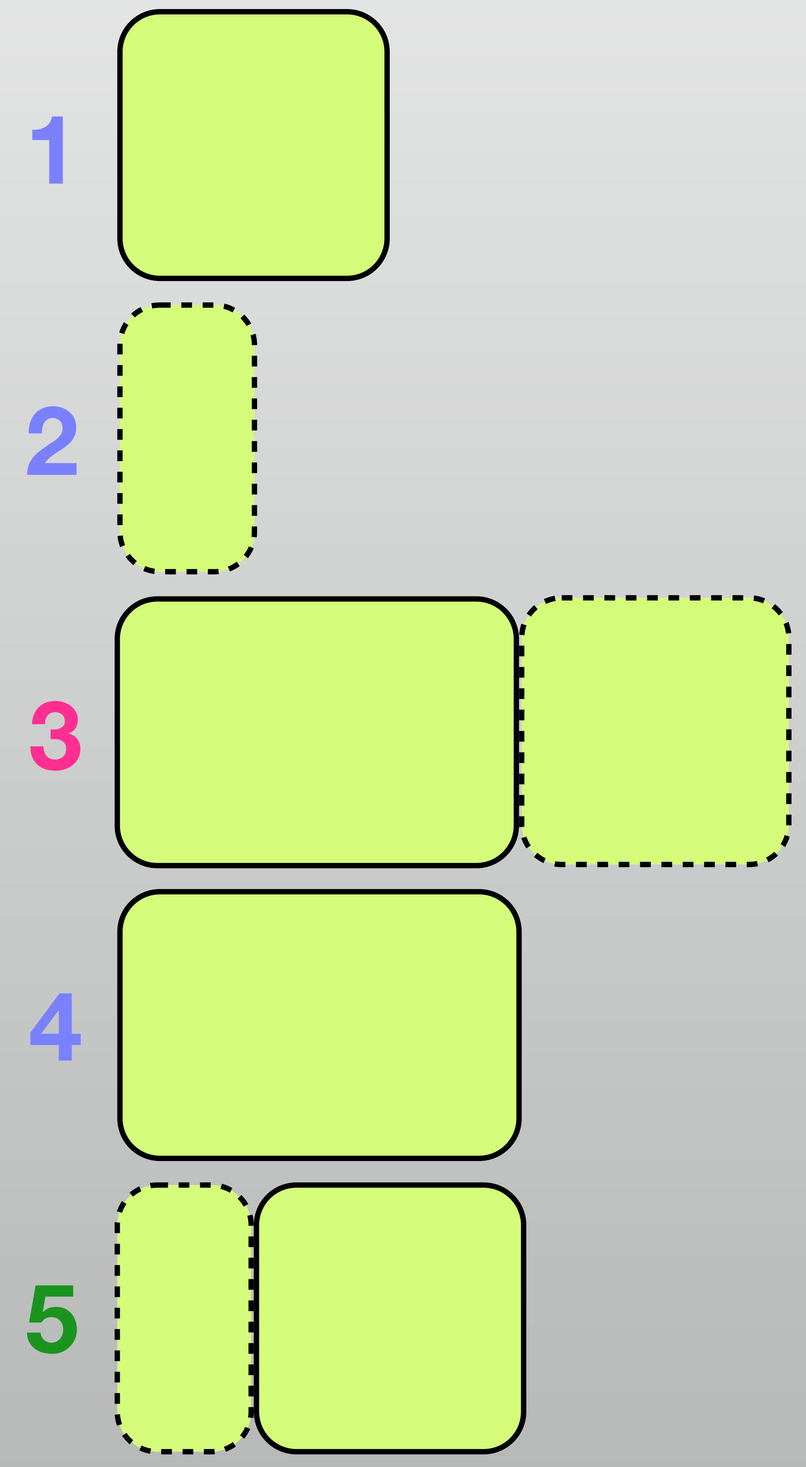
* D-FF + combinational logic = everything
  * System consists of multiple (small) finite state machines

Logic circuit



Path delay

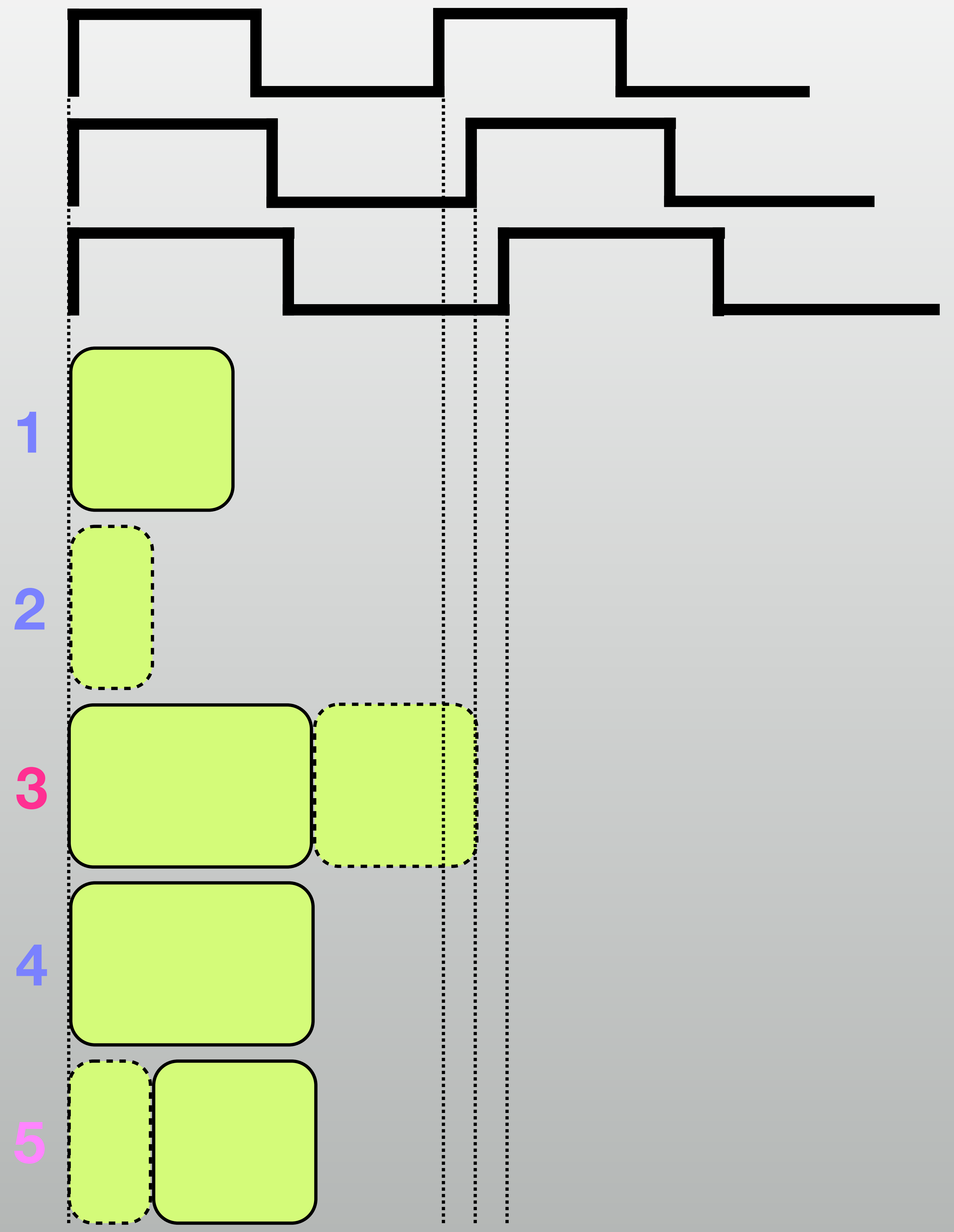




How about path delay from/to I/O Pads and FFs?

Single-clock design

* On same clock rising edge:
    * FFs capture input signals
    * This changes inputs to combinational logics
* Longest delay determines clock frequency



Assignments and signals

* Continuous assignments
  * “**assign**” statement and “**wire**” type variable (as seen last week)
* Procedual assignments:
  * “**always**” and “**initial**” statements”
  * “**reg**” type variable

Rule on wire and reg

* All signals (variables) must be declared as reg or wire
* In **procedural** assignments, left side value must be **reg** type
* Wire declaration can be omitted for I/O ports or **1bit signals**
    * wire/reg is not literally wires and registers, but something defined as above

Blocking vs Non-blocking

* Blocking assignment “=“
    * Usually used in software: execution is blocked until completed
* Non-blocking assignment “<=“
    * A <= B; B <= A; is done at the same time, values exchanged
    * Naturally done in hardware

assign statement and "="

- "=" is used in assign statements: blocking
  - assign statements are independent each other: they don't block each other
- assign statement has no delay by default
  - assign statement with delay blocks "within" the statement
  - Input transitions within delay Δt is ignored (details later)

{} in C, begin - end in Verilog

- “if” statement control only 1 statement
  - For multiple statements, use blocks: {} in C
  - In Verilog, begin-end makes a block
    - Multiple non-blocking assignments in a block
    - No blocking assignments are allowed

Example: 8bit counter

- Driven by clock rising edge
  - always @ (posedge CLK)
  - negedge is less used
- This code has no initial value: results unknown

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

  reg [7:0] CNT;

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

  assign COUNT = CNT;

endmodule
```

8bit counter with reset

* if (RST)
  * RST is reset signal
  * If high, set CNT to 0
    If not, run the counter

```
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 flashing

* ○→●→●→● ●→○→●→● … in visible speed
* Clock running @ 100MHz
    * 2^{20}=1M (1,048,576), 2^{24}=16M (16,777,216)
    * 24bit counter makes 6Hz pulse
* Same # of FFs and LEDs to manage illumination state

Counter first

* Free-running 24bit counter
  * Makes “all-1” pulse @ 6Hz

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

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

Flash the LEDs

- On RST, Left-most LED is on
  - Right rotate on STROBE

```
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
```

Connect the ports

- CLK is crystal oscillator
- RST is push button
- LED is output port, declared as **output reg**

```
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
```

Syntax summary

* Ports are implicit wires, but write explicitly to avoid bugs
* always @ (posedge CLK): FF is driven by CLK
    * Use non-blocking assignments “<=“
    * “if” statement is available in “always" block

“Well-used” syntax

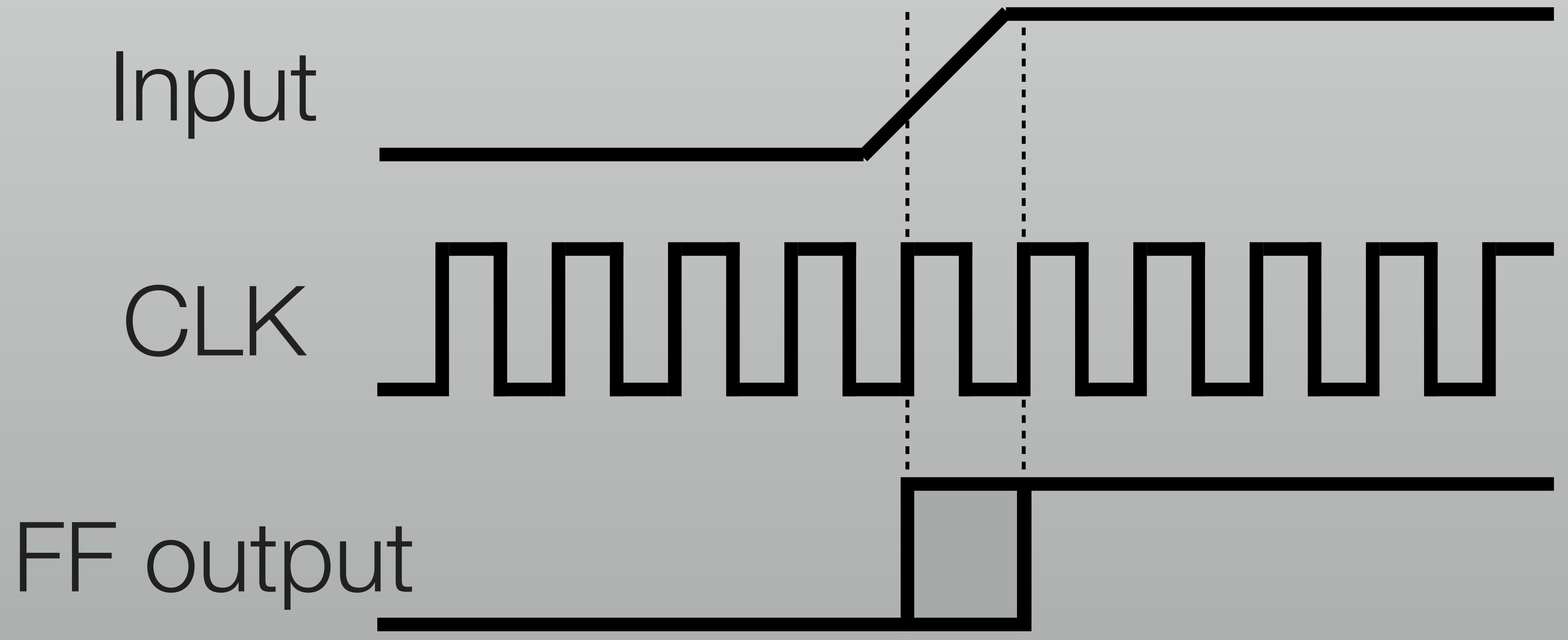
* if (RST)
  * Define “reset” and “running” behavior
  * Not all FFs must be initialized on reset
    * “smaller” reset contributes less load on RST signal

Real-world example: using SWs

- Problem #1: Metastability
  - Caused by “intermediate” voltage level (not H or L) into logic gate: be careful to asynchronous inputs
- Problem #2: Chattering
  - Contact bounces on opening/closing switches
  - From 100MHz clock, looks like a sequence of fast on/off

Metastability

* Strange voltage level of gate output (not H or L)
  * Gates are designed to lower the probability
* Avoided by a series of 2 FFs
  * 2 levels is considered perfect
  * “Must” be placed in async input



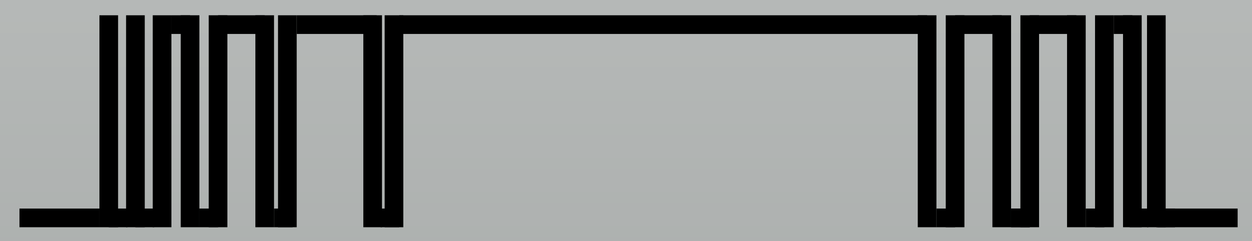
Double Flopping

* Just a series of 2 FFs

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

  reg FF1, FF2;

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

  assign OUT = FF2;

endmodule
```

Chattering

* Switch output is not very clean as “off → on → off”
  * Multiple on/offs may be observed at 100MHz
  * Can be filtered with assumption that “pressed by finger”
    * ex) once turned on, ignore transitions for a while



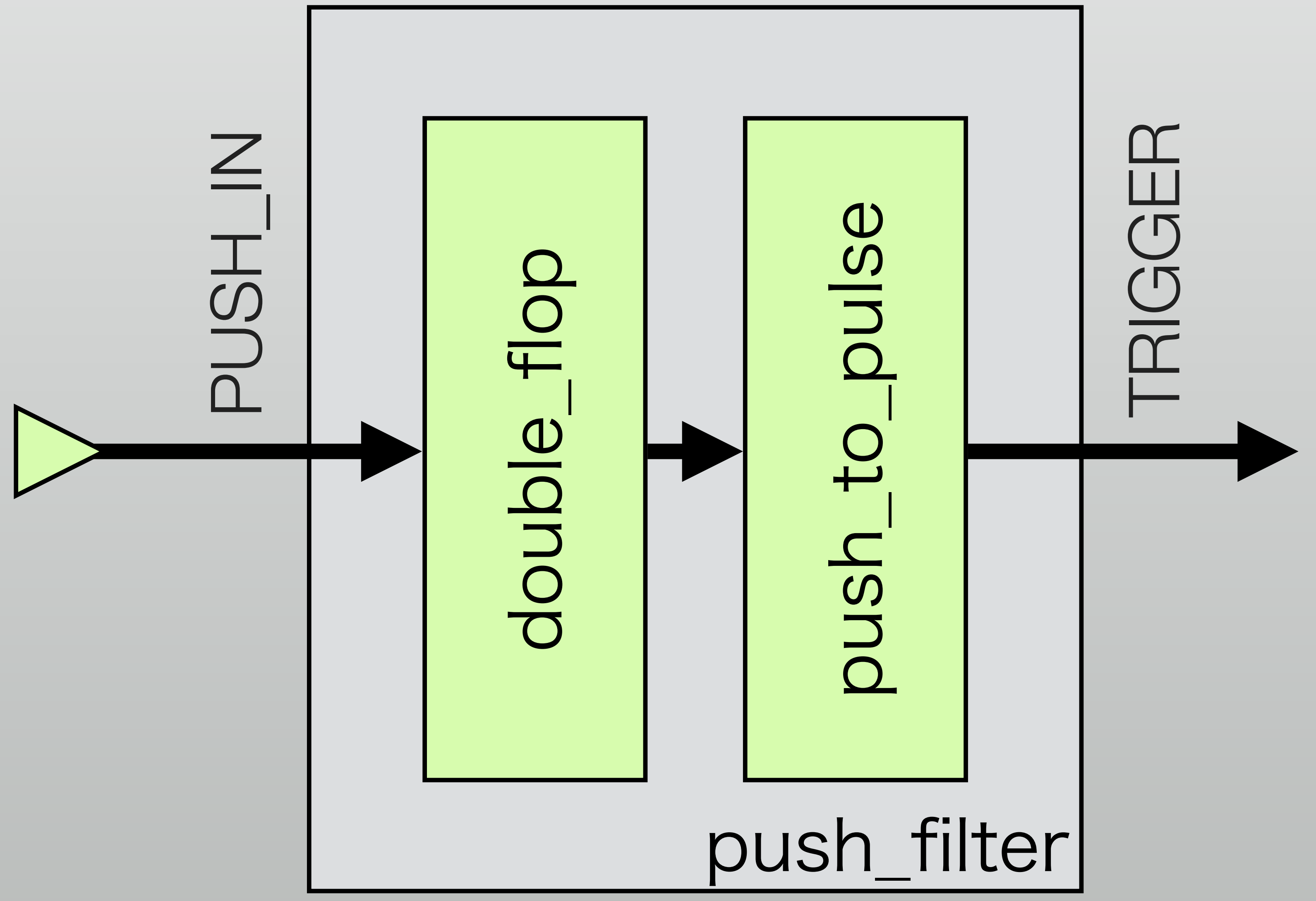
Chattering removal example

- 1 clk pulse on button press
  - 5Hz max.
  - 2^{25}=33,554,432
- Reset required

```
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
```

Modularize

* Pack 2 modules into 1
    * Double flop + Chattering
    * If packed, easily connected to every switch



Using submodule

* Instance, port, signal
  * Module name
  * Instance name
  * .port
  * (signal)

```
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
```

Ex: Use in LED flashing

* Press button to move

```
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
```

Modular design is important

* Beauty of duplication: like attaching same filter on every switch
* Re-use of well-tested module reduces design effort

Simpler is better

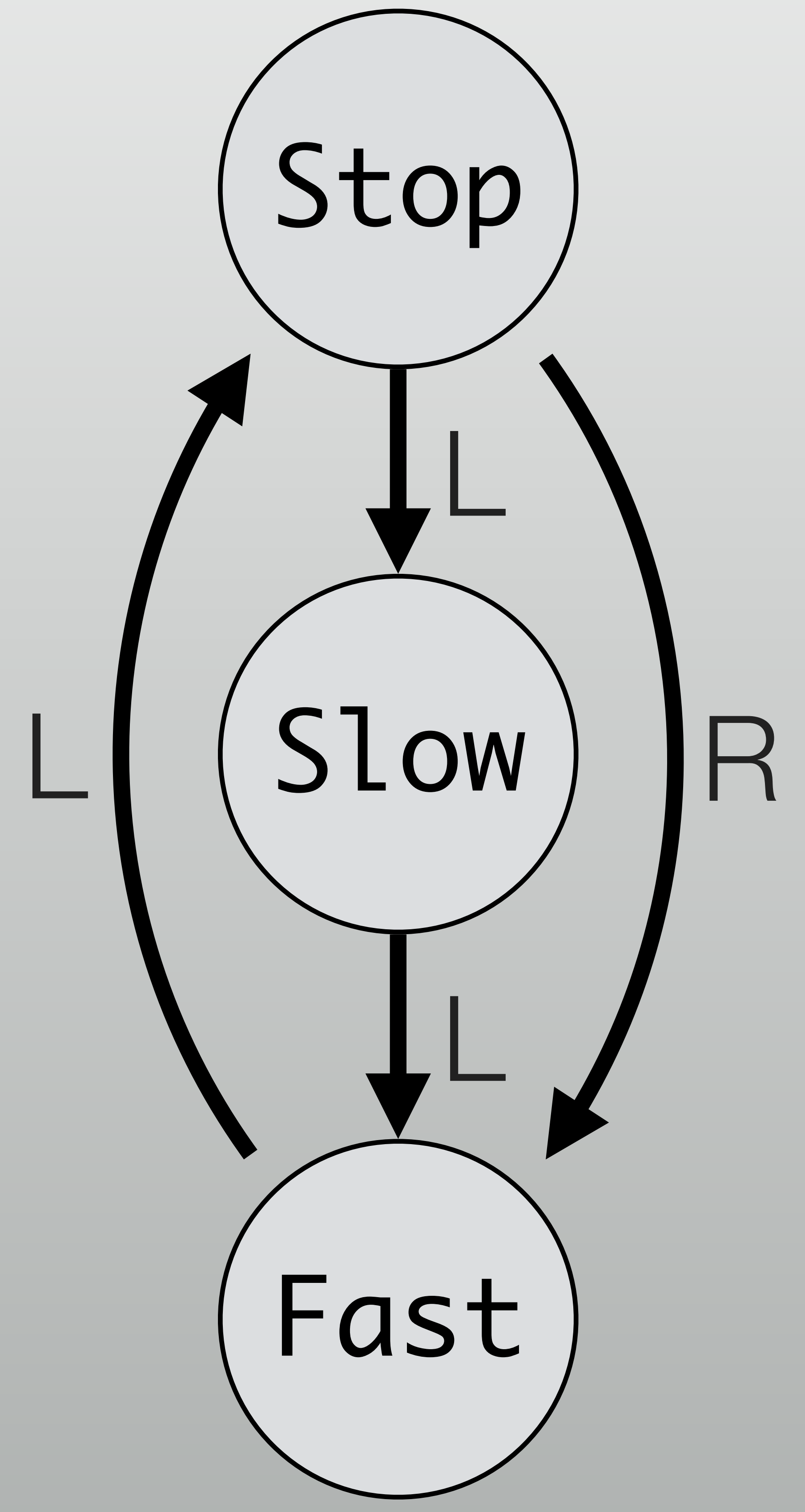
* Always simplify module interface signals
* Consider how the ports work to communicate
  * Simple and predictable interface is better:
    “1 clock pulse on button press” is a good example

Another control structure: case

* “if” statement is evaluated with priority
  * Slower if nested
    * if-else-if-else-if-else…
  * “case” is evaluated in parallel
    * Suitable for writing state machine

Ex: LED flashing state machine

* “Stop” -> “Slow” -> “Fast” circulates on left button
  * “Stop” -> “Fast” shortcut on right button



Interface first

* L & R switch to push_filter
  * Without this, 100M press/s
  * “1 clock output” is good

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

  wire MODE_L, MODE_R;

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

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

  // LED control in next page
endmodule
```

The main FSM

* NO direct control on LED
* Important design principle: to decompose control FSM & data pathes into reasonably small pieces

```
reg [2:0] MODE;

always @ (posedge CLK) begin
  if (RST) begin
    MODE <= ‘b001;
  end else begin
    case (MODE)
      ‘b001: begin // stop
          if (SW_R) MODE <= ‘b100;
          if (SW_L) MODE <= ‘b010; end
      ‘b010: if (SW_L) MODE <= ‘b100;  // slow
      ‘b100: if (SW_L) MODE <= ‘b001;  // fast

      default: MODE <= ‘b001;
    endcase
  end
end

// LED control in next page
```

LED control

- 27-bit counter ≒ 1Hz @ 100MHz clk
  - $2^{27} = 134,217,728$
- Slow mode @ 1Hz, Fast mode @ 0.5Hz

```
reg [26:0] CNT;
wire STROBE;

always @ (posedge CLK) begin
  if (RST) begin
    LED <= ‘b01;
    CNT <= 1;
  end else begin
    CNT <= CNT+1;
    if (STROBE) LED <= {{LED[0], LED[15:1]}};
  end
end

assign STROBE = MODE[2] ? &CNT[25:0] : // fast
                MODE[1] ? &CNT[26:0] : // slow
                0; // stop
```

Summary

* Sequential logic with **always** statement and **reg** variables
    * Within always statement, control structures such as **if** and **case** are available
* Also learned about modules and submodules