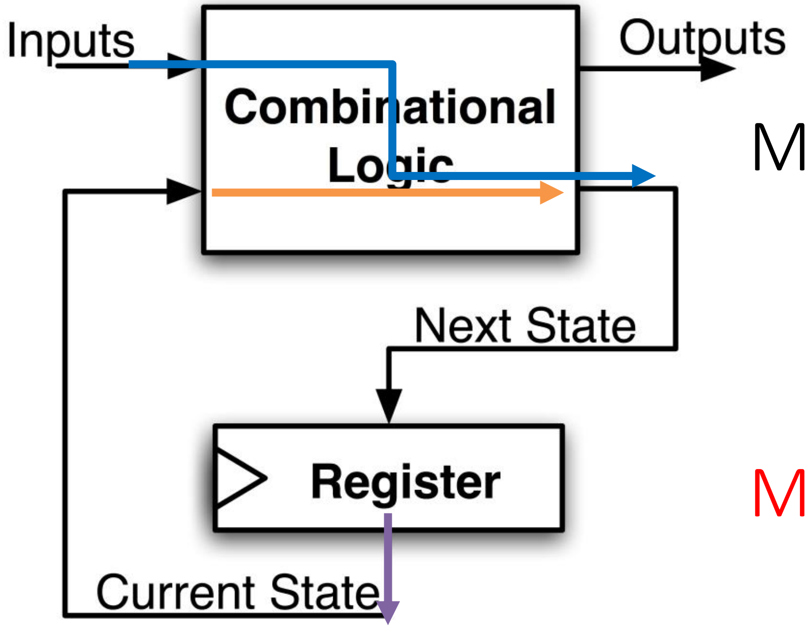
When Can the Input Change?

- When a register input changes shouldn't violate hold time (t_{hold}) or setup time (t_{setup}) constraints within a clock period (t_{period})
- Let $t_{input,i}$ be the time it takes for the input of a register to change for the i-th time in a single clock cycle, measured from the CLK trigger:
  - Then we need $t_{hold} \leq t_{input,i} \leq t_{period} - t_{setup}$ for all i
  - Two separate constraints!

Minimum Delay

- If shortest path to register input is too short, might violate hold time constraint
  - Input could change before state is "locked in"
  - Particularly problematic with *asynchronous* signals

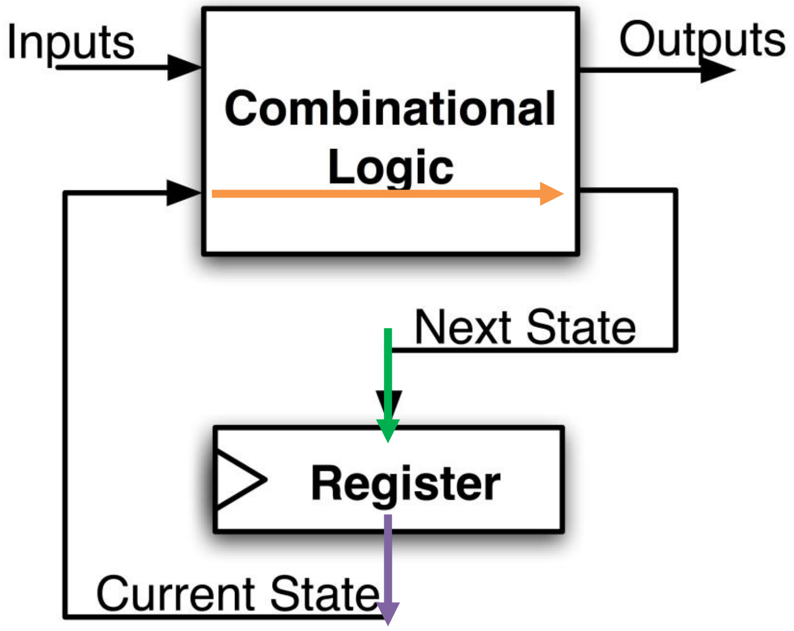


Min Delay = min(CLK-to-Q Delay + Min CL Delay, Min CL Delay)

Min Delay ≥ Hold Time

Maximum Clock Frequency

- What is the max frequency of this circuit?
  - Limited by how much time needed to get correct Next State to Register (t_{setup} constraint)



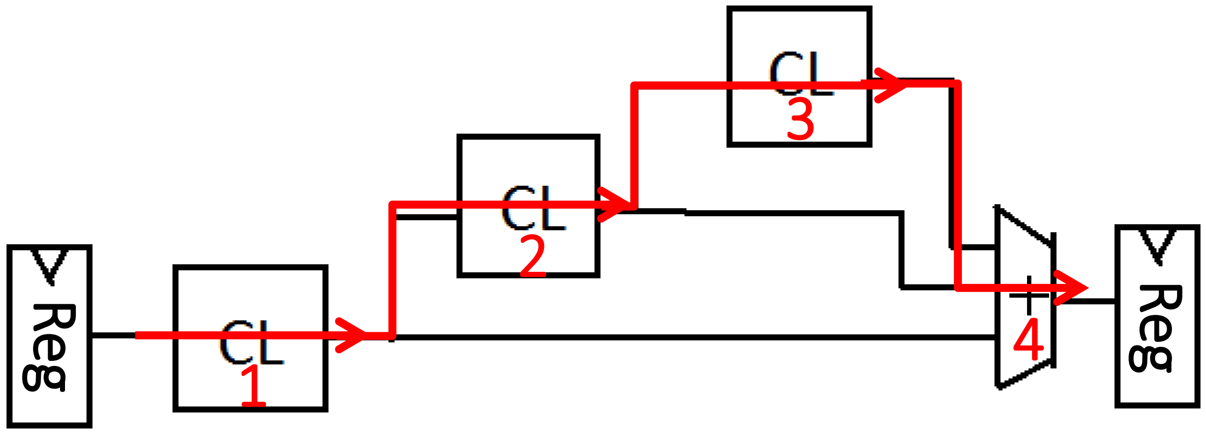
Max Delay = CLK-to-Q Delay
+ Max CL Delay
+ Setup Time

Min Period = Max Delay

Max Freq = 1/Min Period

The Critical Path

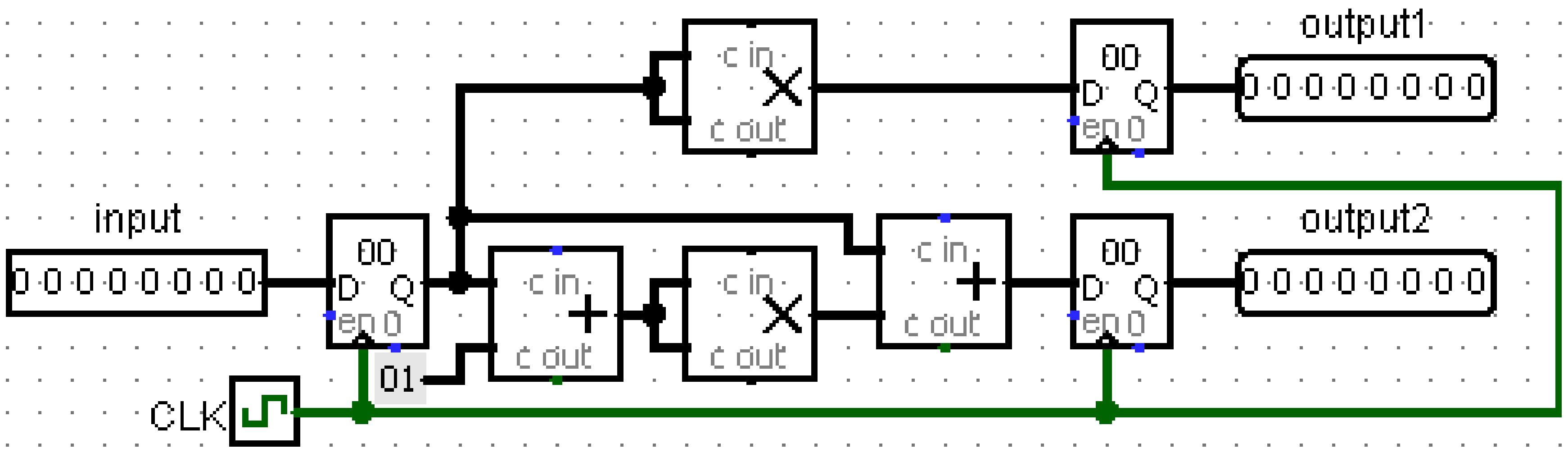
- The *critical path* is the longest delay between *any* two registers in a circuit
- The clock period must be *longer* than this critical path, or the signal will not propagate properly to that next register



Critical Path =
CLK-to-Q Delay
+ CL Delay 1
+ CL Delay 2
+ CL Delay 3
+ Adder Delay
+ Setup Time

Practice Question

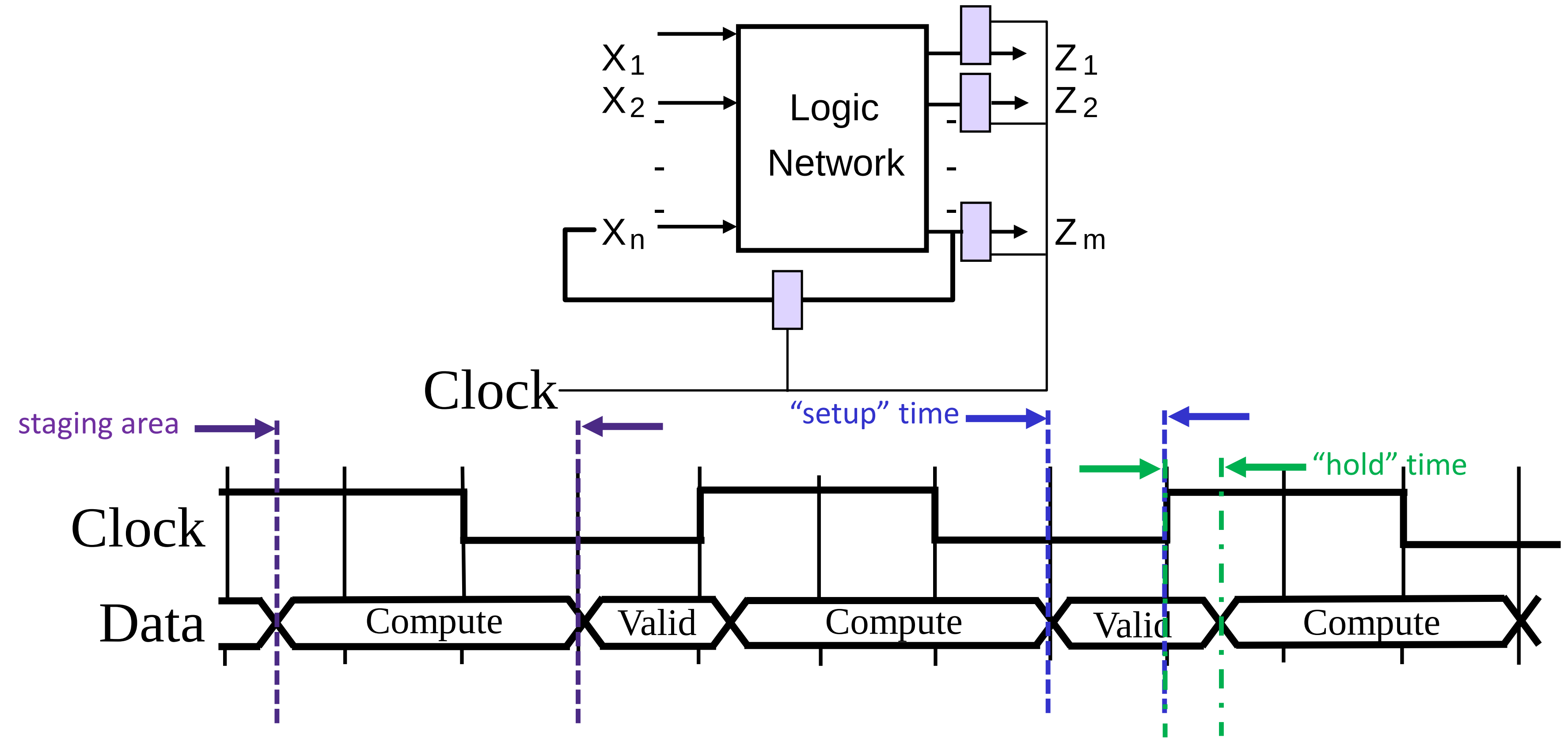
- We want to run on 1 GHz processor. t_{add} = 100 ps. t_{mult} = 200 ps. t_{setup} = t_{hold} = 50 ps. What is the maximum $t_{clk\text{-}to\text{-}q}$ we can use?



(A) 550 ps (B) 750 ps (C) 500 ps (D) 700 ps

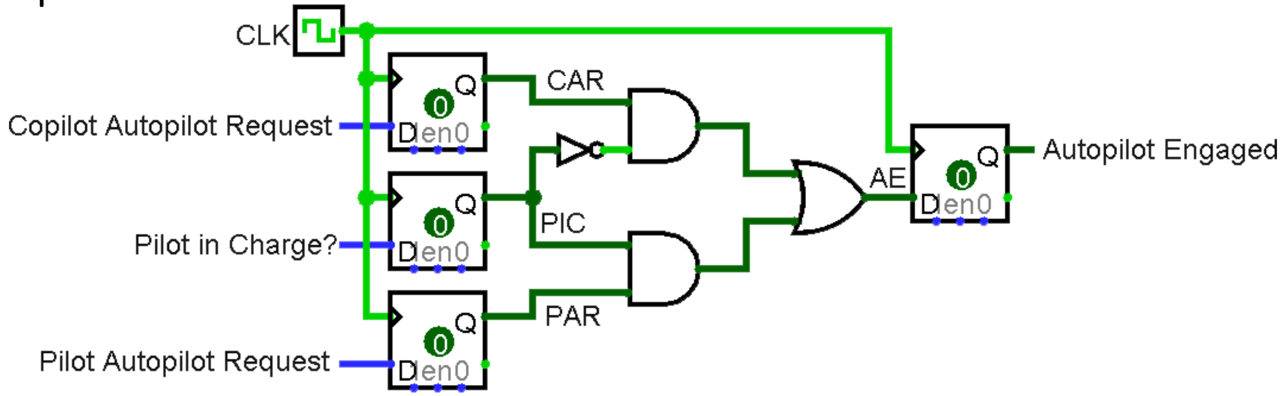
Safe Sequential Circuits

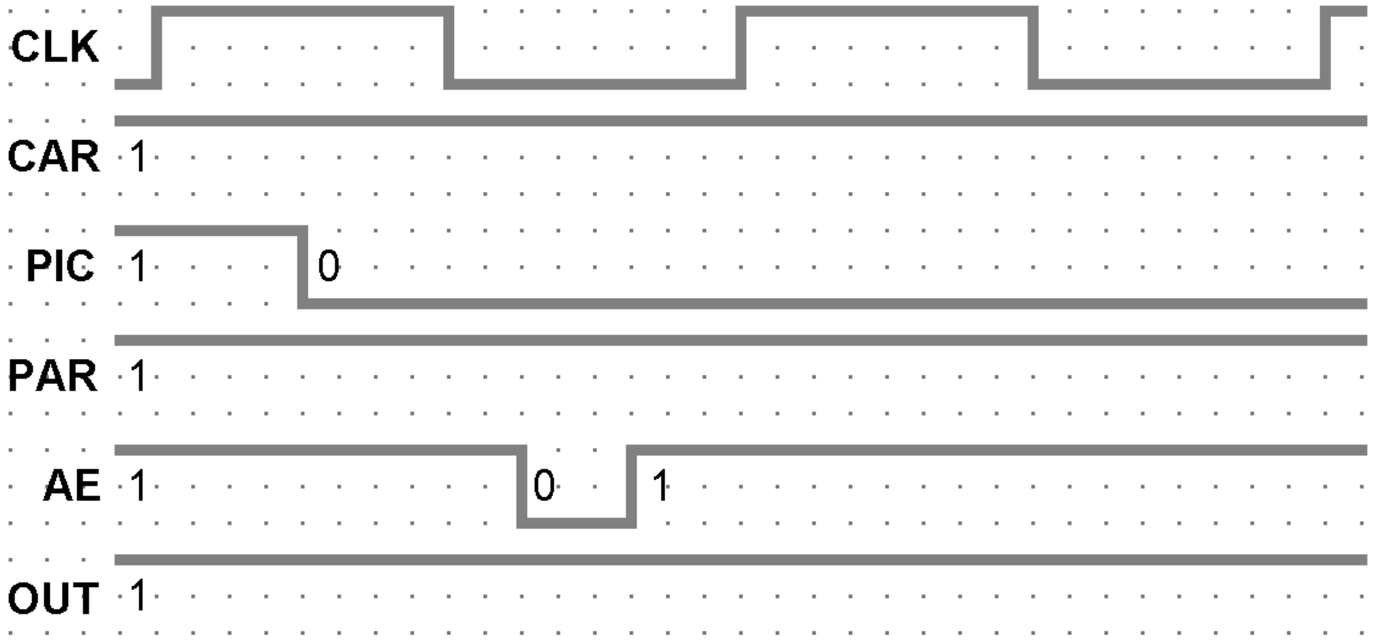
- Clocked elements on feedback, perhaps outputs
  - Clock signal synchronizes operation
  - Clocked elements hide glitches/hazards



Autopilot Revisited

- Flip-flops can "filter out" unintended behavior:





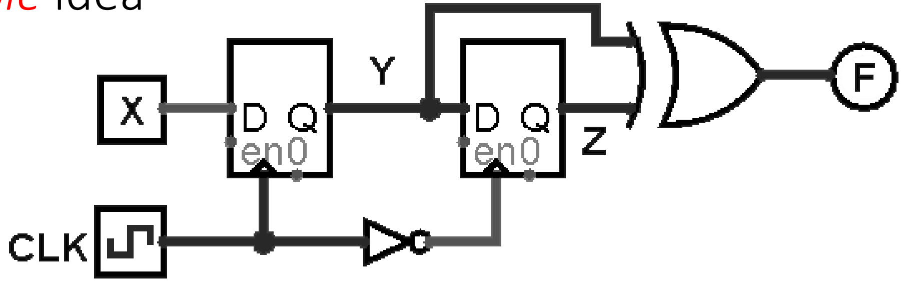
Waveform Diagrams Revisited

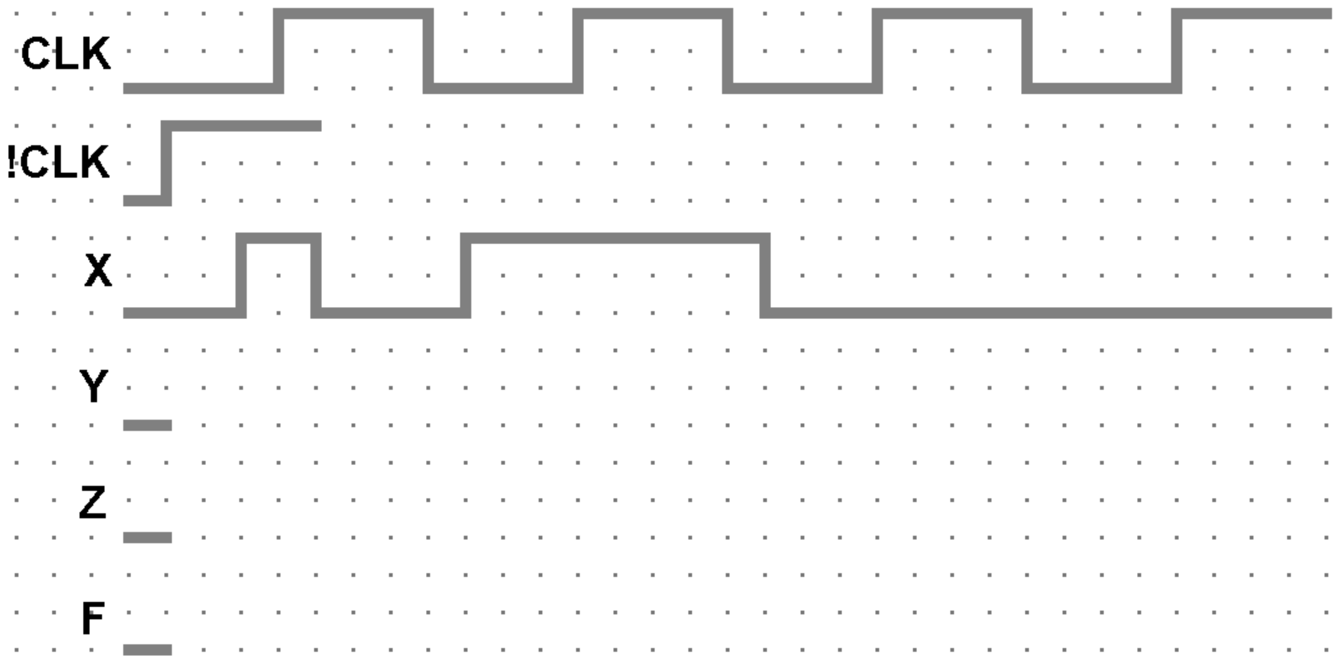
- Easiest to start with CLK on top
  - Solve signal by signal, from inputs to outputs
  - Can only draw the waveform for a signal if *all* of its input waveforms are drawn

- When does a signal update?
  - A *state element* updates based on CLK triggers
  - A *combinational element* updates ANY time ANY of its inputs changes

Example: SDS Waveform Diagram

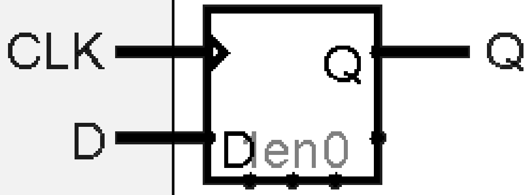
- Assume: t_{C2Q} = 3 ticks, t_{XOR} = 2 ticks, t_{NOT} = 1 tick; t_s = t_h = 0
  - Note: clocking the gate is a *terrible* idea



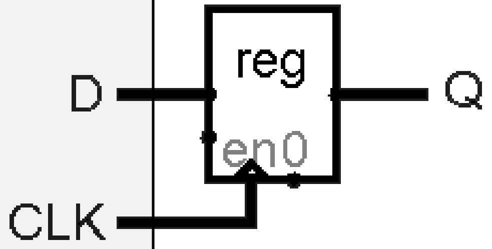


Verilog: Basic D Flip-Flop, Register

```
module basic_D_FF (q, d, clk);
  output logic q; // q is state-holding
  input  logic d, clk;

  always_ff @(posedge clk)
    q <= d; // use <= for clocked elements
endmodule
```



```
module basic_reg (q, d, clk);
  output logic [7:0] q;
  input  logic [7:0] d;
  input  logic       clk;

  always_ff @(posedge clk)
    q <= d;
endmodule
```



Procedural Blocks

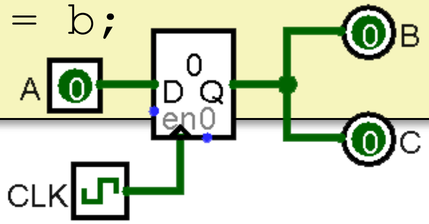
- `always`: loop to execute over and over again
  - Block gets triggered by a *sensitivity list*
  - Any object that is assigned a value in an `always` statement must be declared as a variable (`logic` or `reg`).
  - Example:
    - `always @ (posedge clk)`

- `always_ff`: special SystemVerilog for SL
  - *Only for use with sequential logic – signal intent that you want flip-flops*
  - Example:
    - `always_ff @ (posedge clk)`

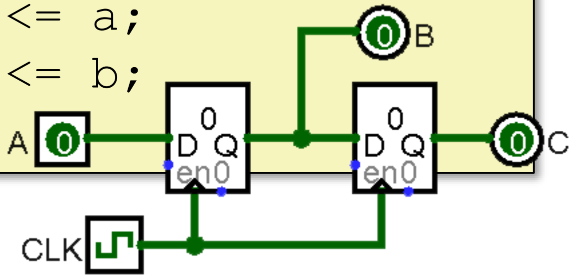
Blocking vs. Nonblocking

- **Blocking** statement (=): statements executed sequentially
  - Resembles programming languages
- **Nonblocking** statement (<=): statements executed "in parallel"
  - Resembles hardware
- Example:

```
always_ff @ (posedge clk)
begin
    b  = a;
    c  = b;
end
```



```
always_ff @ (posedge clk)
begin
    b <= a;
    c <= b;
end
```



Verilog Coding Guidelines

1) When modeling sequential logic, use *nonblocking* assignments

2) When modeling combinational logic with an `always_comb` block or assign statements, use *blocking* assignments

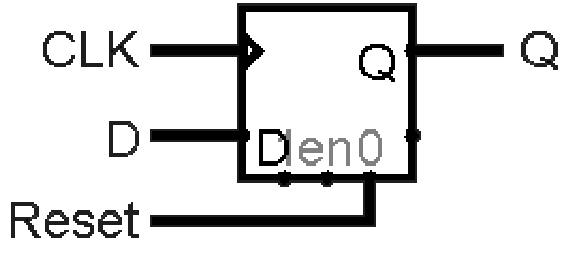
3) Avoid complex logic in `always_ff` blocks – instead, compute complex logic in `always_comb` blocks.

4) Do not mix *blocking* and *nonblocking* assignments in the same `always` block

5) Do not make assignments to the same variable from more than one `always` block

Verilog: D Flip-Flop w/Synchronous Reset

```
module D_FF (q, d, reset, clk);
  output logic q;   // q is state-holding
  input  logic d, reset, clk;

  always_ff @(posedge clk)
    if (reset)
      q <= 0;        // on reset, set to 0
    else
      q <= d;        // otherwise pass d to q

endmodule
```



Verilog: Simulated Clock

- For simulation, you need to generate a clock signal:
  - For entirety of simulation/program, so use `always` block

Explicit Edges:

```
initial
   clk = 0;
always begin
   #50  clk <= 1;
   #50  clk <= 0;
end
```

Toggle:

```
initial
   clk = 0;
always
   #50  clk <= ~clk;
```

- Define clock period:
  - Define **parameter**

```
parameter period = 100;
initial
   clk = 0;
always
   #(period/2)  clk <= ~clk;
```

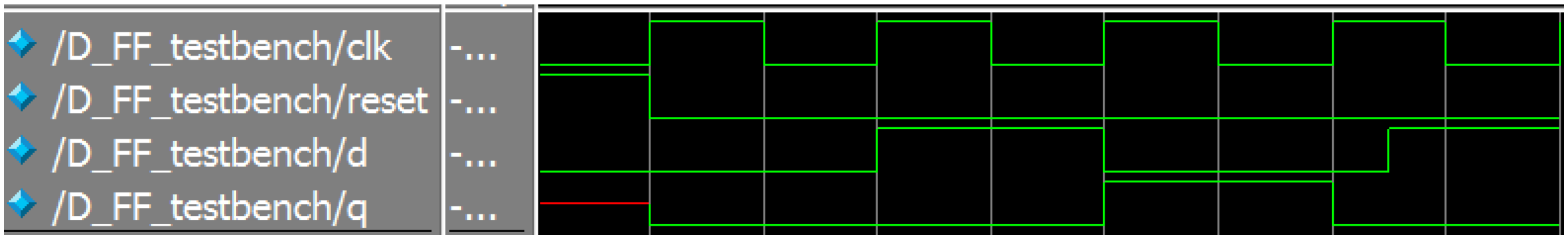
Verilog Testbench with Clock

```
module D_FF_testbench;
  logic CLK, reset, d;
  logic q;

  parameter PERIOD = 100;

  D_FF dut (.q, .d, .reset, .CLK); // Instantiate the D_FF

  initial CLK <= 0;                // Set up clock
  always #(PERIOD/2) CLK<= ~CLK;

  initial begin                    // Set up signals
                  d <= 0; reset <= 1;
    @(posedge CLK);       reset <= 0;
    @(posedge CLK); d <= 1;
    @(posedge CLK); d <= 0;
    @(posedge CLK); #(PERIOD/4) d <= 1;
    @(posedge CLK);
    $stop();                       // end the simulation
  end
endmodule
```

Timing Controls

- Delay: `#<time>`
  - Delays by a specific amount of simulation time
  - Can do calculations in `<time>`
  - Examples: `#(PERIOD/4), #50`
- Edge-sensitive: `@(<pos/negedge> signal)`
  - Delays next statement until specified transition on signal
  - Example: `@(posedge CLK)`
- Level-sensitive Event: `wait(<expression>)`
  - Delays next statement until `<expression>` evaluates to TRUE
  - Example: `wait(enable == 1)`

ModelSim Waveforms



```
initial begin
                        d <= 0; reset <= 1;
   @(posedge CLK);              reset <= 0;
   @(posedge CLK); d <= 1;
   @(posedge CLK); d <= 0;
   @(posedge CLK); #(PERIOD/4) d <= 1;
   @(posedge CLK);
   $stop();
end
```

Summary (1/2)

- State elements controlled by clock
  - Store information
  - Control the flow of information between other state elements and combinational logic
- Registers implemented from flip-flops
  - Triggered by CLK, pass input to output, can reset
- Critical path constrains clock rate
  - Timing constants: setup time, hold time, clk-to-q delay, propagation delays

Summary (2/2)

- Generating a clock
  - Manually create using `always` block
  - Need to decide on period
- Blocking vs. Non-blocking
  - Blocking: Statements executed “in series”
  - Non-blocking: Statements executed “in parallel”
  - Always use non-blocking for clocked elements
- Synchronous vs. Asynchronous
  - Whether signals are controlled by clock or not