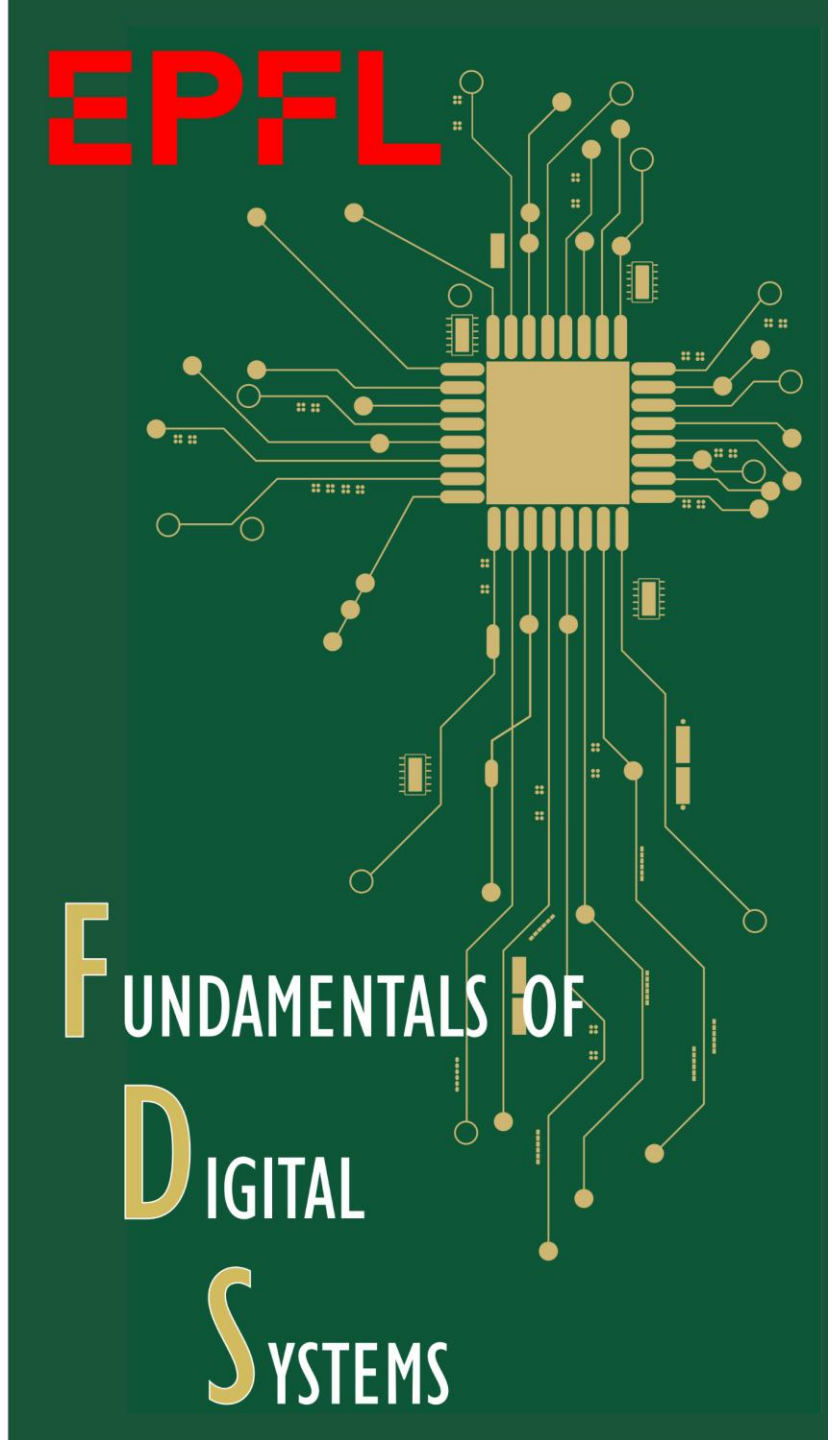


Digital Logic Circuits

Sequential Logic

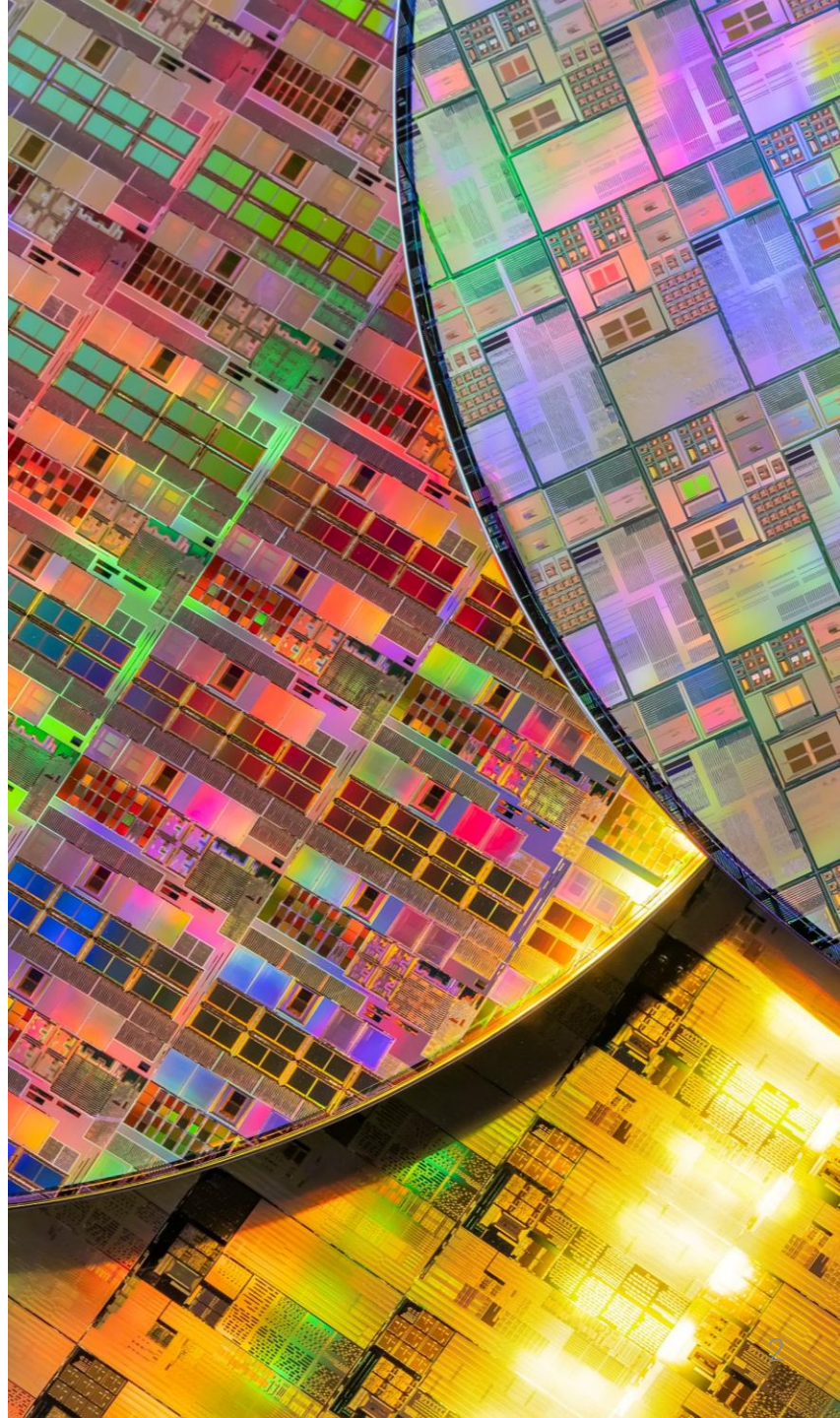
CS-173 Fundamentals of Digital Systems

Mirjana Stojilović

Spring 2025

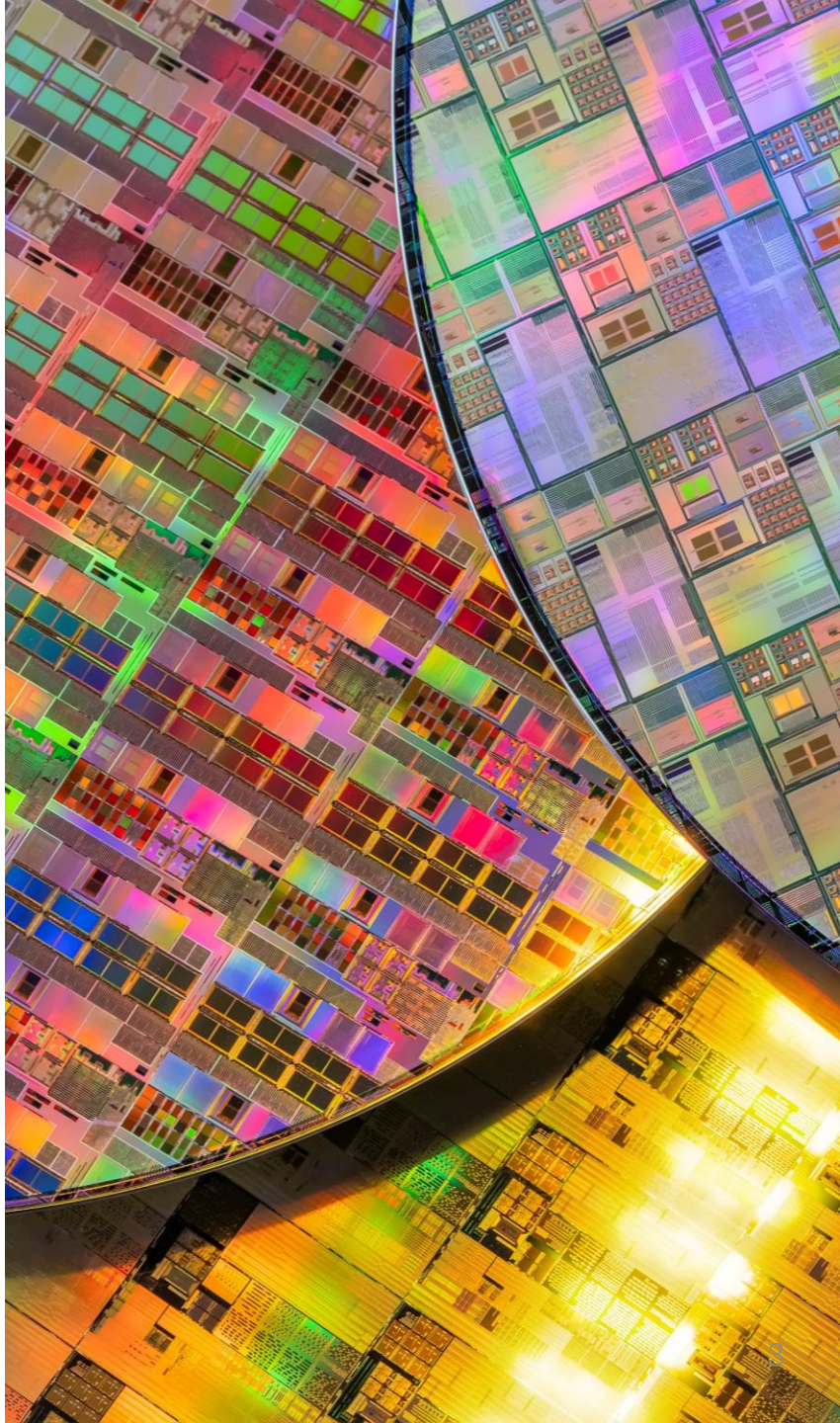
Previously on FDS

Transistors, CMOS logic gates,
real gate behavior, dynamic operation,
dynamic power dissipation



Previously

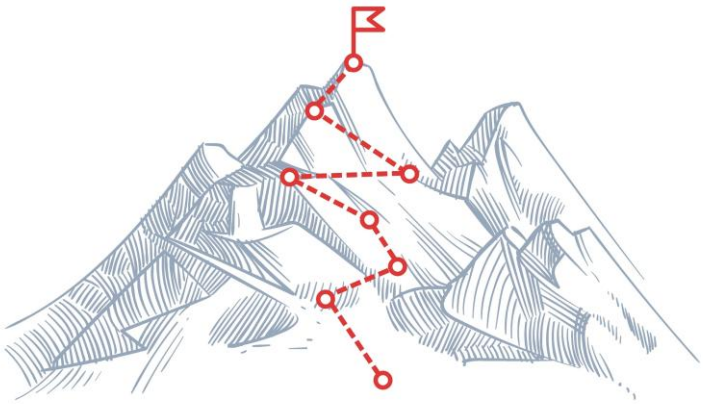
- Discovered NMOS and PMOS transistors from which real logic gates are built
  - CMOS examples (NOT, NOR, NAND, AND)
- Ideal vs. real gates
  - On-resistance and gate capacitance
  - Rise and fall times
  - Fan-in and fan-out
  - Propagation delays
- Unexpected transitions (hazards)
- Dynamic power consumption



Let’s Talk About…

…Sequential logic

Learning Outcomes



- Discover memory elements
  - Latches
  - Flip-flops
- Use a behavioral modeling approach to describe them
- Understand the difference between blocking and nonblocking assignments
  - Write `always@` blocks correctly, depending on what they are supposed to describe: combinational or sequential circuits

Quick Outline

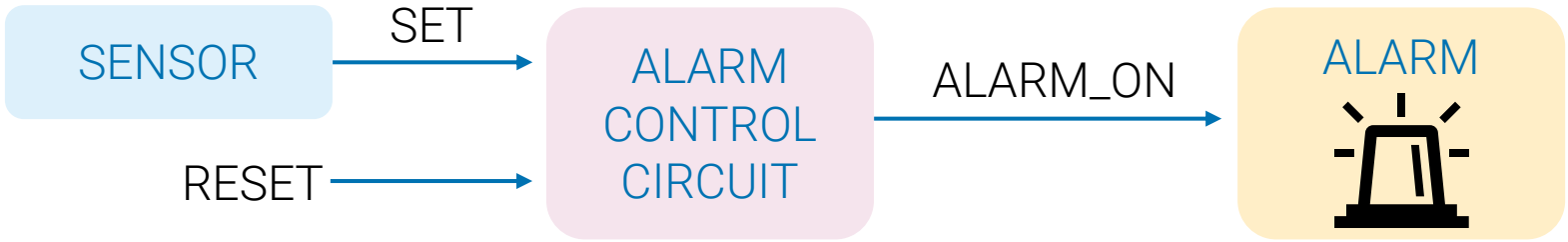
- Memory elements
  - Combinational vs. sequential
  - Sequential circuits
  - Basic memory element
  - SR latch
  - D latch
  - D flip-flop
- Clock
- Verilog, contd.
  - Blocking assignments
  - Nonblocking assignments
- Behavioral latch and FF models
- Practically useful notes
  - Avoid latches
  - Beware, latches can sneak in

Memory Elements

- Logic circuits that can store information

Example Application: Alarm System Control

- Suppose we wish to control an alarm system



- ALARM_ON = 1, the alarm is activated; ALARM_ON = 0, the alarm is deactivated

- *When the sensor generates a positive voltage signal (SET = 1) in response to some undesirable event, ALARM_ON becomes 1.* ***Once the alarm is triggered, it must remain active even if the sensor output returns to zero****. The alarm is turned off by means of a RESET input. The circuit requires a* ***memory element*** *to remember that the alarm has to be active until the RESET signal arrives.*

Combinational vs. Sequential

- Previously, we considered circuits where the value of each output depends solely and almost instantaneously on the values of signals applied to the inputs
  - Referred to as **combinational circuits**

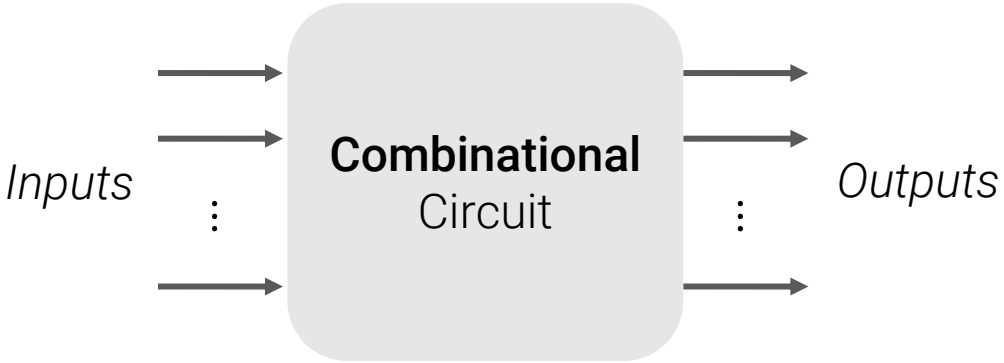
- There exists another class of logic circuits in which the values of the outputs depend **not only on the present** values of the inputs **but also on the past** behavior of the circuit
  - Contain memory elements
  - Referred to as **sequential circuits**

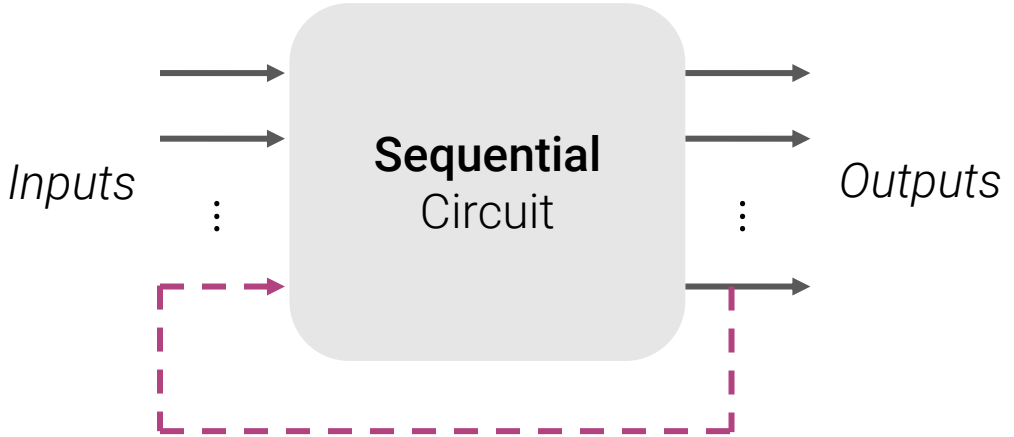
Combinational vs. Sequential

Contd.

- Combinational circuits are **memoryless**:
  - Outputs depend **only** on the present inputs

- Sequential circuits have **memory**:
  - Outputs depend on the **present** and the **previous** inputs





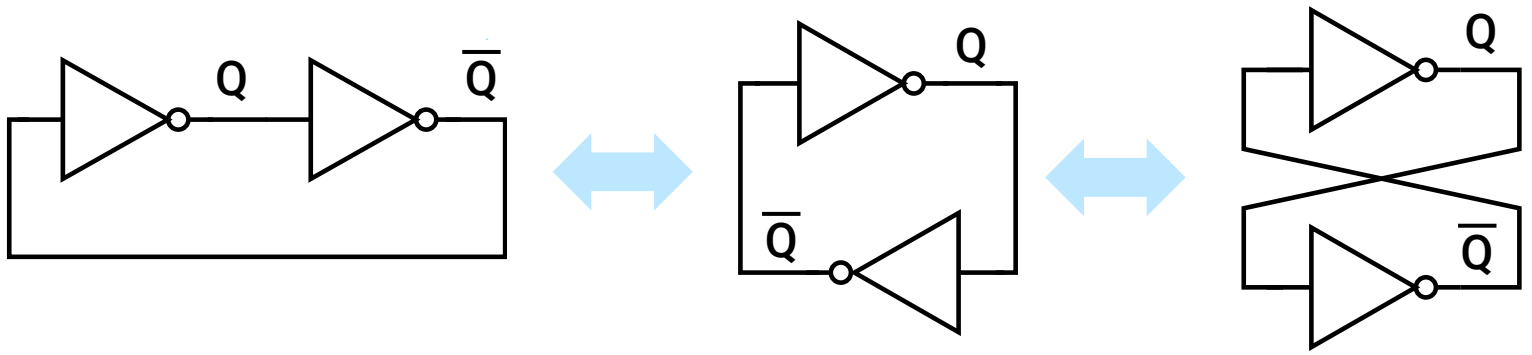
Sequential Circuits

- Include **storage elements** (memory elements) to store (i.e., memorize) the values of logic signals
- The contents of the storage elements are the **state of the circuit**
- When the inputs change, the new input values either leave the circuit in the **same state** or cause it to change to a **new state**
- Over time, the circuit changes through a **sequence of states** as a result of changes in the inputs

Basic Memory Element

Bistable Element

- Inverters with outputs connected to inputs



Q	$\overline{Q}$
0	1
1	0

- Not very practical: stores a “given” value indefinitely
- How to update the stored value? *We need additional inputs*

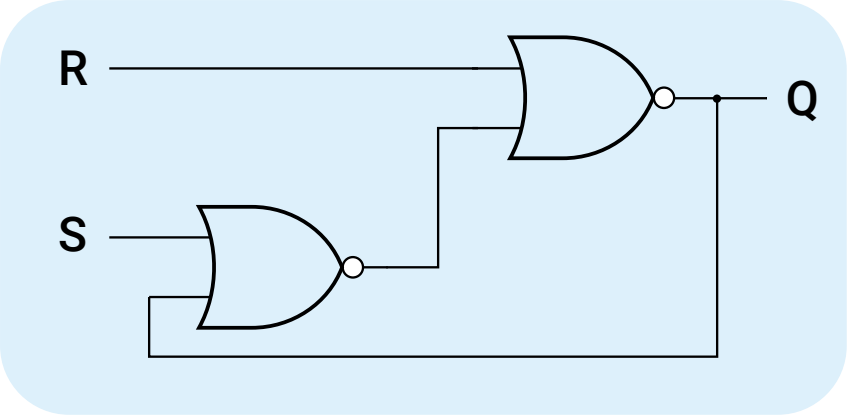
Memory Elements

- Latches

Basic Memory Element

Set-Reset Latch with Reset Priority

- A memory element with NOR gates
- Both NORs act as inverters in a bistable memory element



- A table describing a sequential circuit behavior is called a **characteristic** table
  - How the next state changes in the function of the inputs **and** the previous state

S	R	Q_{next}
0	0	
0	1	***f*(S, R, Q)**
1	0	
1	1	

A State of a Latch

Definition

- Depending on the value of the output Q, a latch can be in one of the two states (S):
  - S0: Q = 0
  - S1: Q = 1

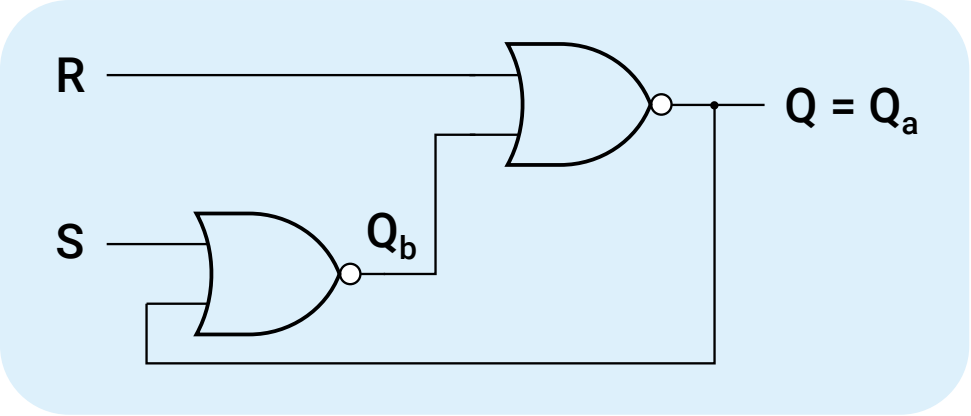
Note: State names S0 and S1 are chosen arbitrarily

- A state is a property of a memory element
- State is defined by the logic value kept by the memory element (Q)

Set-Reset Latch with Reset Priority

Basic Memory Element, Contd.

- Called Set-Reset Latch



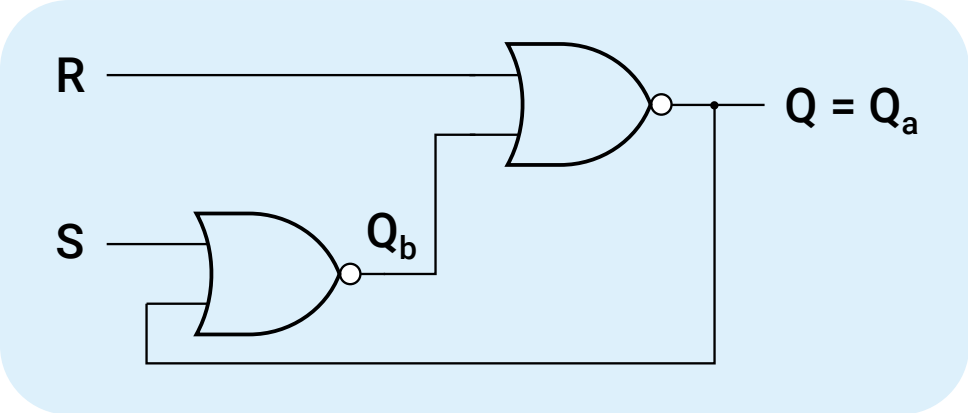
$$Q_b = \overline{S + Q_a}$$

$$\begin{aligned} Q_{a,next} &= \overline{R + Q_b} \\ &= \overline{R + \overline{S + Q_a}} \\ &= \overline{R} \cdot (S + Q_a) \\ &= \overline{R} \cdot S + \overline{R} \cdot Q_a = Q_{next} \end{aligned}$$

Set-Reset Latch with Reset Priority

Basic Memory Element, Contd.

- Called Set-Reset Latch

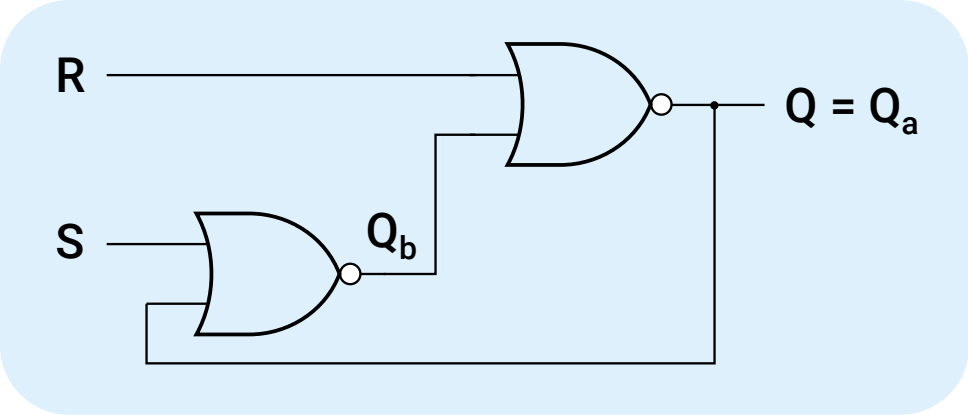


- Assume initially $Q_a = 0$, $Q_b = 1$
- **R inactive** (zero):
  - When S becomes 1:
    - Q_b becomes 0, and Q_a becomes 1
  - While S is 0:
    - Q_a and Q_b are complements of one another
    - Circuit state does not change (it is stable)

Set-Reset Latch with Reset Priority

Basic Memory Element, Contd.

- Called Set-Reset Latch

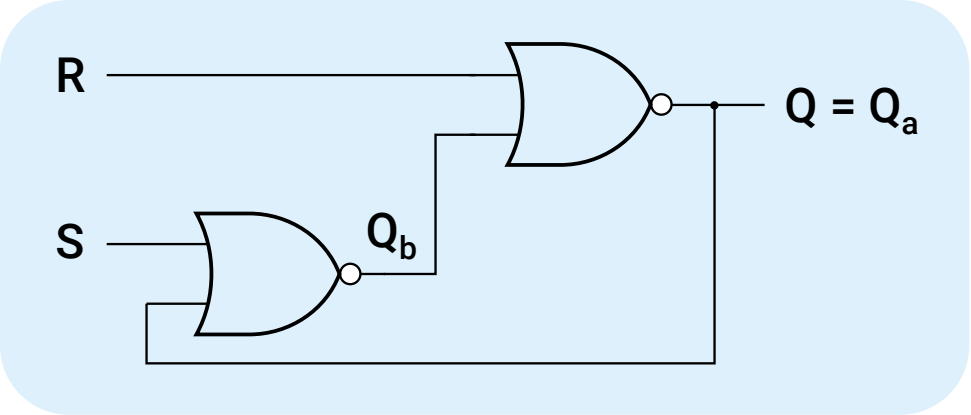


- Assume initially $Q_a = 0$, $Q_b = 1$
- **S inactive** (zero):
  - When R becomes 1:
    - Q_a becomes 0, and Q_b becomes 1
  - While R is 0:
    - Q_a and Q_b are complements of one another
    - Circuit state does not change (it is stable)

Set-Reset Latch with Reset Priority

Basic Memory Element, Contd.

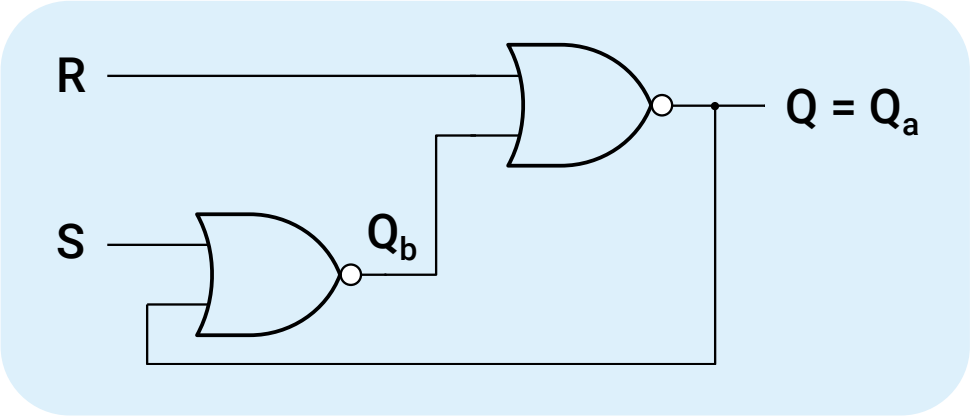
- Called Set-Reset Latch



- Assume initially $Q_a = 0$, $Q_b = 1$
- If both S and R are active
  - Q_a and Q_b become 0, both

Set-Reset Latch with Reset Priority

Basic Memory Element, Contd.



S	R	$\overline{R} \cdot S$	$\overline{R} \cdot Q_a$	Q_{next}	$Q_{b,next}$
0	0	0	Q_a	Q_a	$\overline{Q_a}$
0	1	0	0	0	1
1	0	1	Q_a	1	0
1	1	0	0	0	0

$$Q_b = \overline{S + Q_a}$$

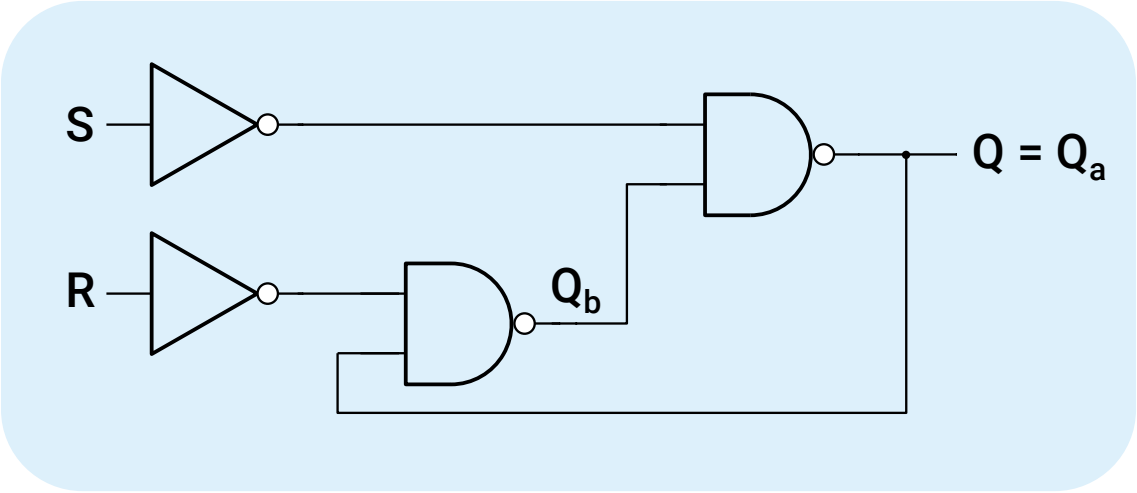
$$\begin{aligned} Q_{a,next} &= \overline{R + Q_b} \\ &= \overline{R + \overline{S + Q_a}} \\ &= \overline{R} \cdot (S + Q_a) \\ &= \overline{R} \cdot S + \overline{R} \cdot Q_a = Q_{next} \end{aligned}$$

- R active: Output Q_a is **reset**
- S active: Output Q_b is `0`
  - If R inactive: Output Q_a is **set**
- R and S inactive: Outputs constant

Note: We try to avoid activating both R and S

Set-Reset Latch with Set Priority

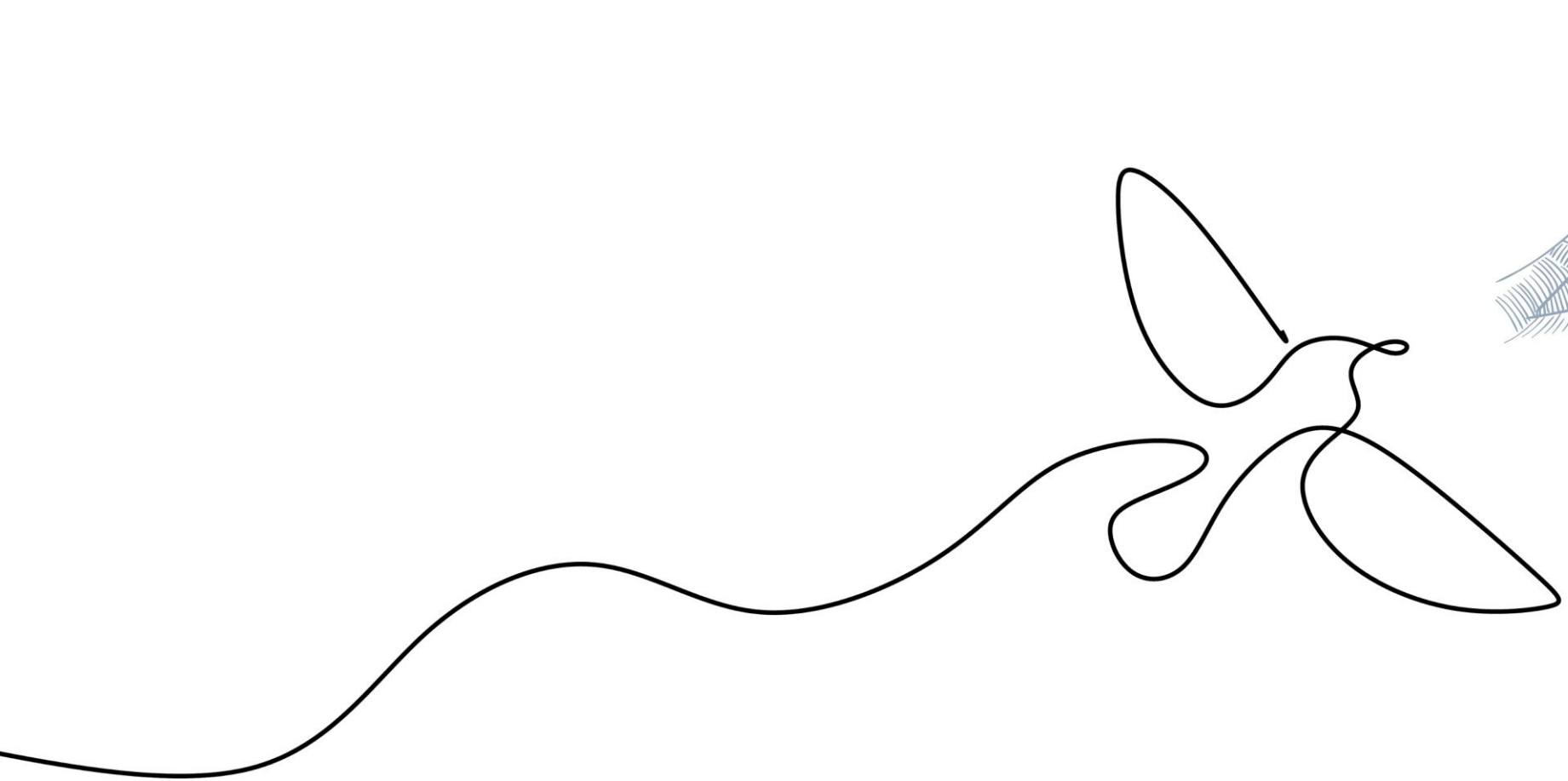
- The latch we saw was an SR latch with Reset priority
  - If both S and R are active, the reset "wins" (has higher priority)
- Below is an SR latch with a **Set priority**. Describe its operation.

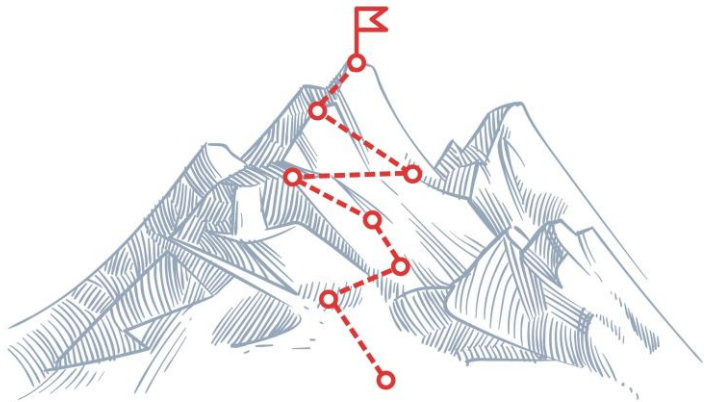


$$Q_{a,next} = S + \overline{R} \cdot Q_a$$

S	R	$\overline{R} \cdot Q_a$	$Q_{a,next}$	$Q_{b,next}$
0	0	Q_a	Q_a	$\overline{Q_a}$
0	1	0	0	1
1	0	Q_a	1	0
1	1	0	1	1

Note: We try to avoid activating both R and S





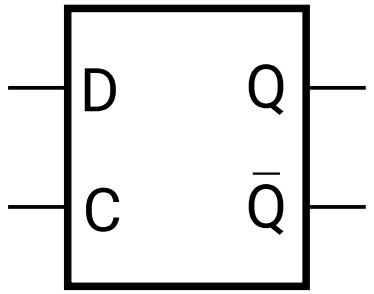
Latches with A Control Signal

- In practice, we want to be able to control **when** state changes occur
  To that purpose, we add a **control** signal
  - **When active, it enables the latch to operate normally**
  - **When inactive, it prevents state updates**
- A variety of latches exist, e.g., D latch (see next slide)

D Latch

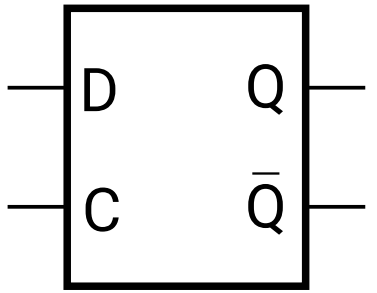
- **Level-sensitive** element
- While the controlling signal C is active (high **level**), the output Q follows all changes taking place on input D
- At all other times, the output Q stays **unchanged** (keeps its last value)

- Schematic symbol
  - Left: inputs D and C
  - Right: complementary outputs

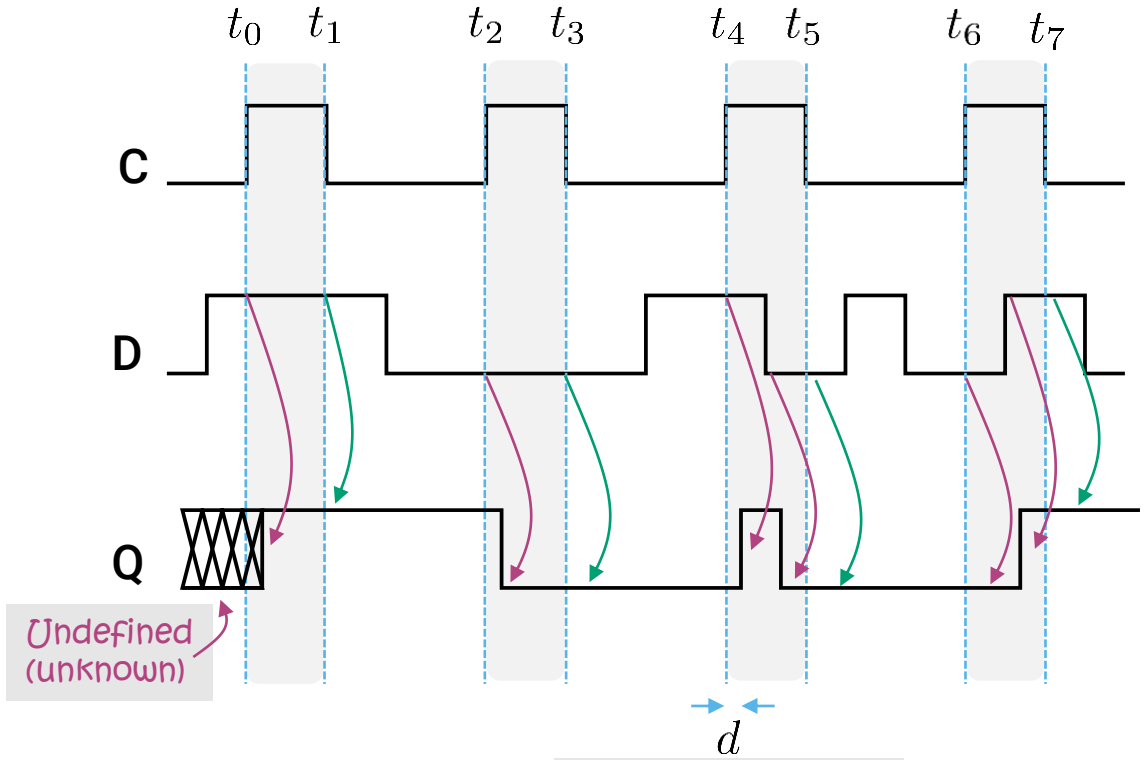


D Latch – Timing Diagram

Example



- For the given signals C and D, draw the output Q waveform



- The signal at C input is active (logic '1') in time intervals (t_0,t_1), (t_2, t_3), (t_4, t_5), (t_6, t_7)
  - Output Q follows input D

- The signal at C input is inactive (logic '0'), in time intervals (..., t_0), (t_1,t_2), (t_3, t_4), (t_5, t_6), (t_7, ...)
  - Output Q keeps its last value (stays constant, unchanged)

Memory Elements

- Flip-flops

Flip-Flops

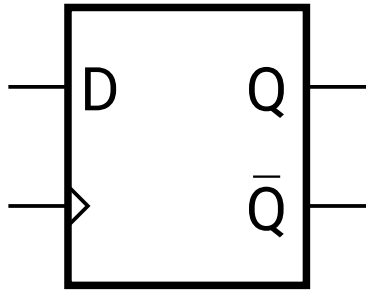
- **D latch may change its state many times while the control signal is active**; in practice, limiting the duration of time when the state changes can occur is very desirable
  - Also, we want to memorize (keep) a state for a while and not allow it to change an unlimited number of times while the control signal is active

- The circuit could be allowed to change the state only when the control signal **transitions** (e.g., rising or falling edge)
  - Such circuits are called **flip-flops**

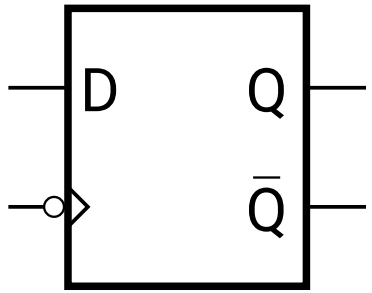
D Flip-Flop

- D flip-flop is an **edge-sensitive** memory element
  - It responds to the changes at the input D only when the controlling signal C transitions (rising edge or falling edge)
- Flip-flop is called an **FF, in short**

- Graphical symbols
  - DFF sensitive to the rising edge

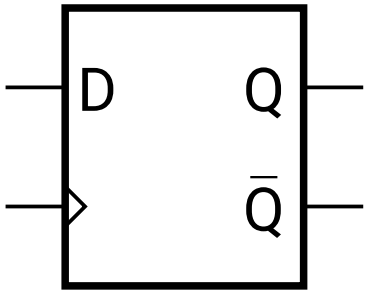


  - DFF sensitive to the falling edge

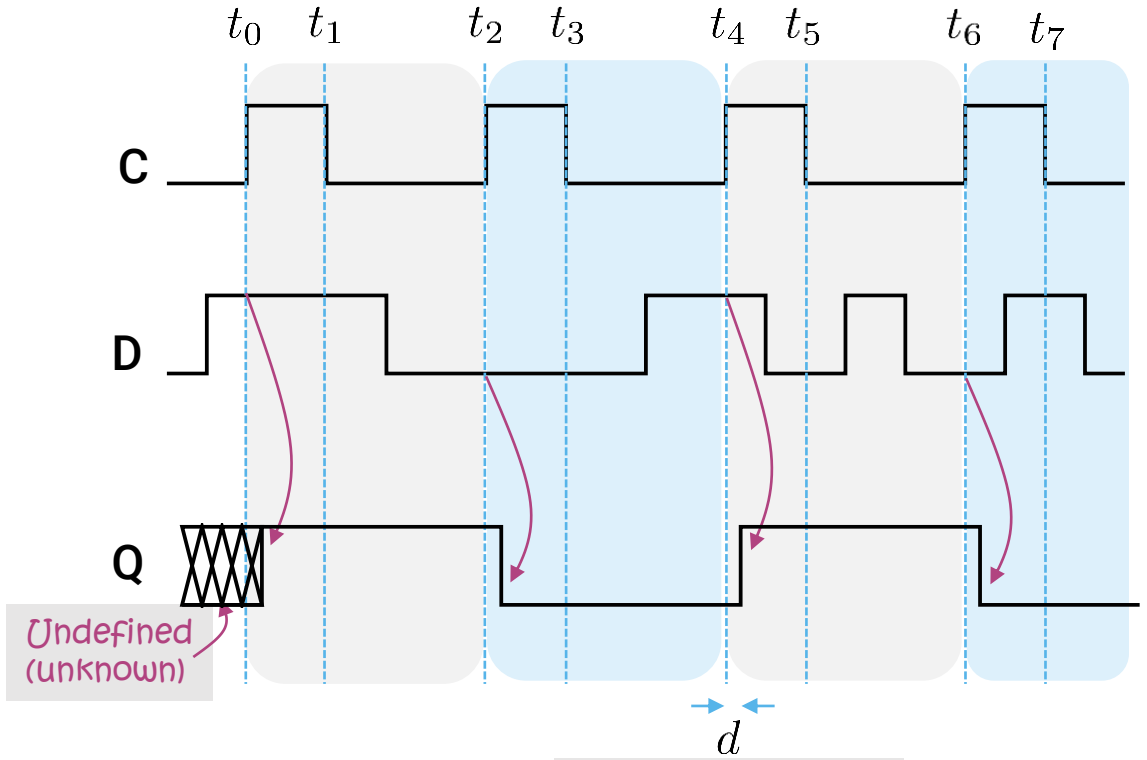


D Flip-Flop – Timing Diagram

Example



- For the given signals C and D, draw the output Q waveform



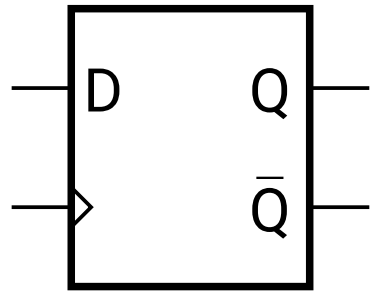
- The signal at the control input changes from '0' to '1' (rising edge) at times t_0, t_2, t_4, and t_6
  - Output Q changes, taking input D
- At any other moment
  - Output Q keeps its last value (stays constant, unchanged)

A State of a FF

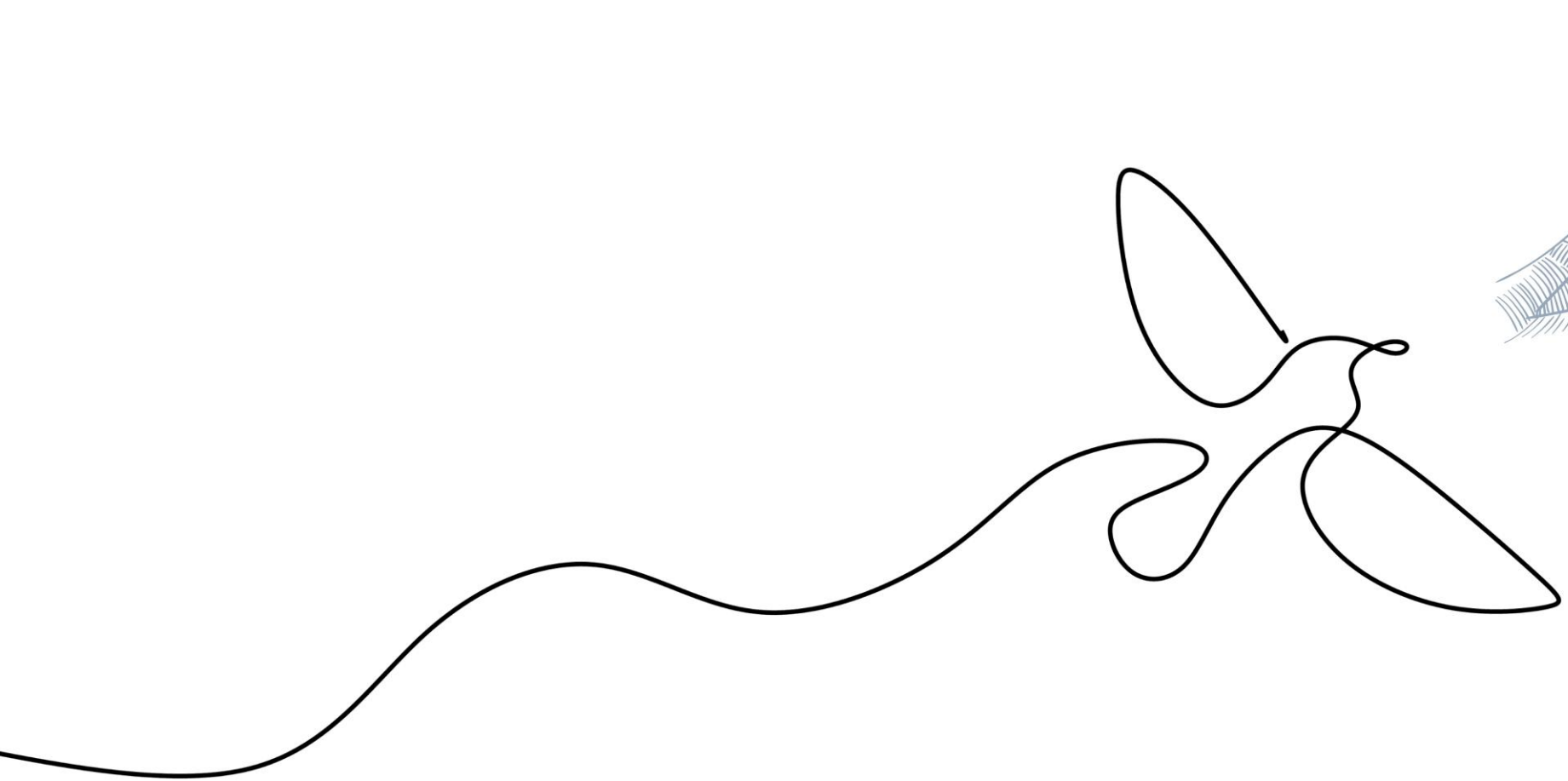
Definition

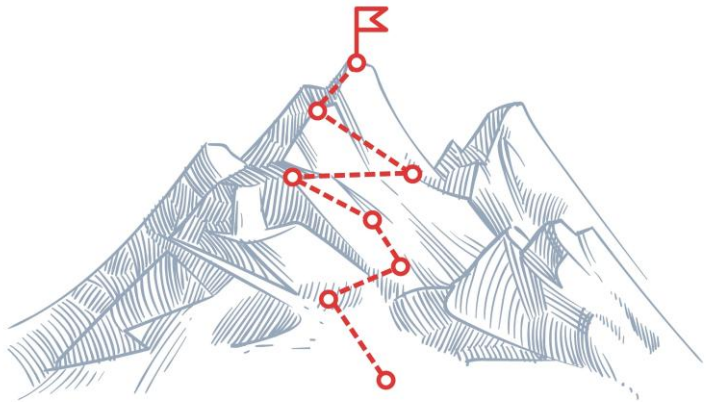
- Depending on the value of the output Q, a DFF can be in one of the two states (S):
  - S0: Q = 0
  - S1: Q = 1

Note: State names S0 and S1 are chosen arbitrarily



- A state is a property of a memory element (e.g., a latch, a FF)
- State is defined by the logic value kept by the memory element (Q)

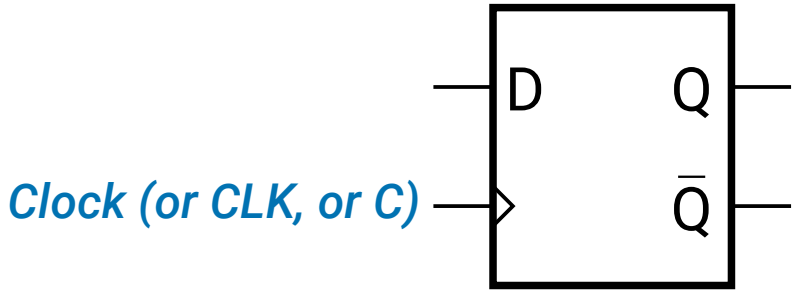




Clock

Clock Signal

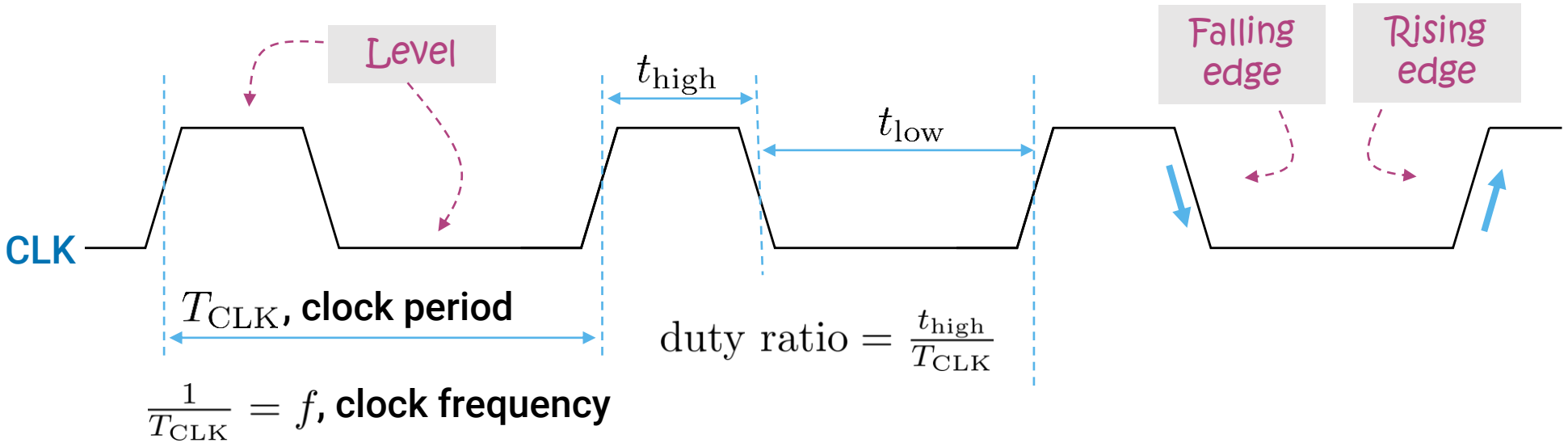
- Signal determining when the state changes occur is called **clock**
- Clock is a periodic signal defined by its frequency (or period) and duty ratio (typically 50%)
  - Example frequency: 100 MHz, 3.3 GHz, 1 kHz
  - Example period: 10 ns, 0.33 ns, 1 ms



- All digital systems use clocks to synchronize state changes
  - Typically, state changes are triggered by a rising edge of the clock

Clock Signal

- *Clock:* A periodic signal determining when the memory elements in a sequential logic circuit update their outputs
- Defined by its frequency f (or period T) and duty ratio (typically 50%)



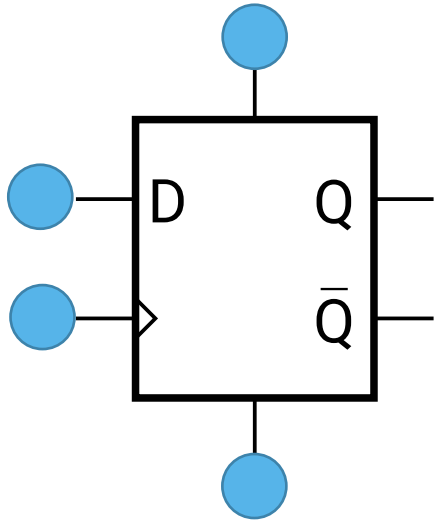
Note: Special circuits are used to generate clocks of desired frequencies.

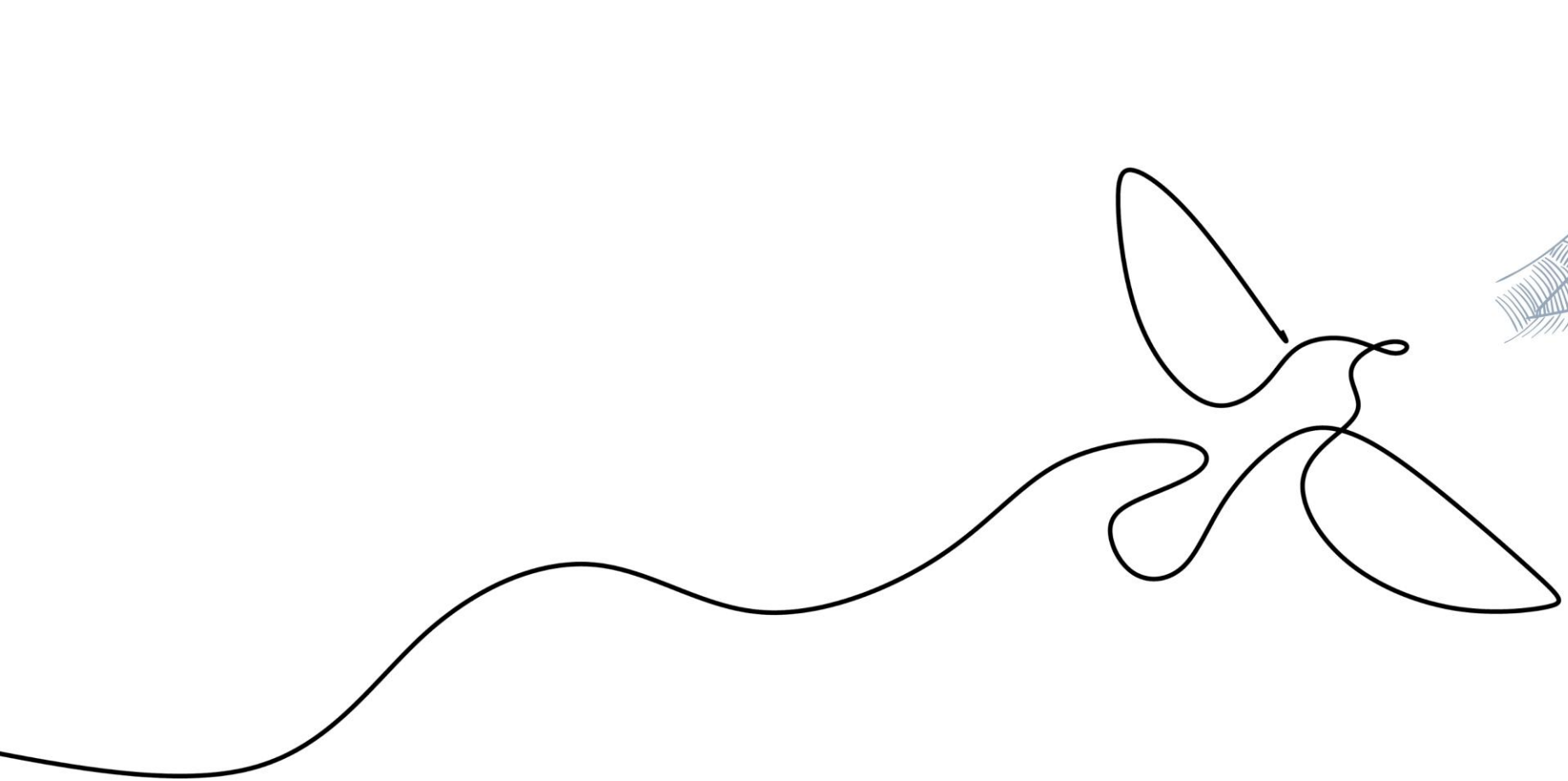
Synchronous vs. Asynchronous Signals

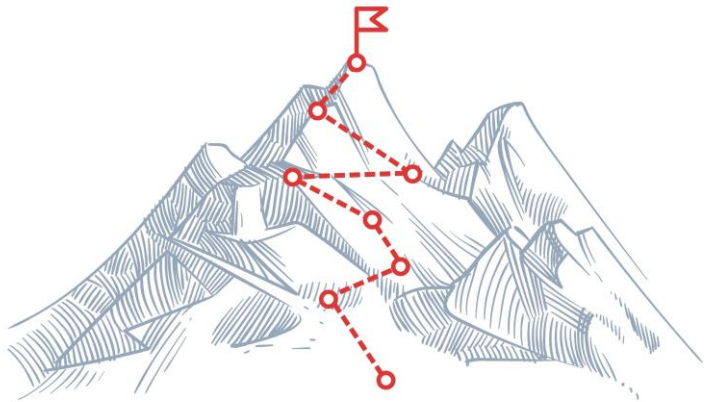
- A **synchronous signal** is one that is evaluated and updated only on the active edge (rising or falling) of the clock
  - Example: D (data) input
  - **Active clock edge** is the edge that triggers the state change (can be rising, or falling)
- An **asynchronous signal** is one that immediately affects the output (the state) regardless of the clock edge
  - It forces the output to a specific state, independent of the clock

Other Control Signals FFs Can Have

- Flip flops can have a variety of inputs, besides data (D) and clock (C)
- **Clear (or reset, CLR)**
  - **Asynchronous clear:** When active, irrespective of the clock, the Q output is cleared
  - **Synchronous clear:** When the clock transitions (e.g., rising edge), if the clear is active, the Q output is cleared
- **Preset (or set)**
  - The opposite effect of the clear
- **Clock Enable (CE)**
  - Typically synchronous: when the clock input transitions (e.g., rising edge), if CE is active, the Q output becomes D







Verilog, Contd.

- Blocking assignments

= (Blocking) Assignments

In Verilog

- Blocking (=) assignments ensure immediate updates; They are best for **combinational logic**
- Blocking assignments happen **sequentially**
- If an `always@` block contains multiple blocking **(=)** assignments, they are evaluated one after another in the current simulation step (simulation cycle)

Simulation Execution Flow

With Blocking Assignments

- For the Verilog model at the right, assume the following sequence at the inputs A/C/E/D during simulation steps (cycles):
  - Step 0, initial state:
    A = B = C = D = E = F = 0
  - Step 1: A = 1
  - Step 2: C = 1
  - Step 3: E = 1
  - Step 4: A = 0
- Compute the values of B, D, and F throughout the simulation cycles

```
always @ (*)
begin
  B = ~A;     // B depends on A
  D = B & C; // D depends on B and C
  F = E | D; // F depends on E and D
end
```

Simulation Execution Flow

Contd.

- **Time step 0 (Initial State):**
  - A = 0, B = 0, C = 0, D = 0, E = 0
  - Execution of the always @(*) block:
    - B = ~A = ~0 = 1
    - D = B & C = 1 & 0 = 0
    - F = E | D = 0 | 0 = 0
- **Time step 1 (Change in A):**
  - A = 1 (Changed from 0 to 1)
  - Execution of the always @(*) block:
    - B = ~A = ~1 = 0
    - D = B & C = 0 & 0 = 0
    - F = E | D = 0 | 0 = 0
- **Time step 2 (Change in C):**
  - C = 1 (Changed from 0 to 1)
  - Execution of the always @(*) block:
    - B remains 0 (already calculated)
    - D = B & C = 0 & 1 = 0
    - F = E | D = 0 | 0 = 0
- **Time step 3 (Change in E):**
  - E = 1 (Changed from 0 to 1)
  - Execution of the always @(*) block:
    - B remains 0 (already calculated)
    - D = B & C = 0 & 1 = 0
    - F = E | D = 1 | 0 = 1
- **Time step 4 (Change in A):**
  - A = 0 (Changed from 1 to 0)
  - Execution of the always @(*) block:
    - B = ~A = ~0 = 1
    - D = B & C = 1 & 1 = 1
    - F = E | D = 1 | 1 = 1

= (Blocking) Assignments

In Verilog

Takeaways

- B/D/F are updated based on the current inputs at each time step
- Each time the inputs A/C/E change, the `always@(*)` block recalculates B/D/F in order of the statements
- Notice how the order of assignment matters, as D depends on B (which was just updated), and F depends on D (which also changes depending on B and C)

`always@(*)` Blocks

In Verilog

- Most often used to describe **combinational logic**
- Use `(*)` for the sensitivity list of the combinational circuit
  - You want your outputs to trigger on a change of any relevant input
  - Use only **`always@(*)`** block when wanting to infer elements that change value as soon as one or more of the inputs change
- Use only `=` (blocking) assignments in combinational circuits

Verilog, Contd.

- Nonblocking assignments

`<=` (Nonblocking) Assignments

In Verilog

- Nonblocking assignments (`<=`) cause all assignments in the `always` block to occur **in parallel**, without affecting each other in the current simulation time step
- Nonblocking (`<=`) assignments are used to model sequential logic behavior

Simulation Execution Flow

With Nonblocking Assignments

- For the Verilog model at the right, assume the following sequence at the inputs A/C/E/D during simulation steps (cycles):
  - Step 0, initial state:
    A = B = D = 0; C = E = F = 1
  - Step 1: A = 1
  - Step 2: C = 0
  - Step 3: E = 0
  - Step 4: A = 0
- Compute the values of B, D, and F throughout the simulation cycles

```
always @ (*)
begin
  B <= ~A;     // B depends on A
  D <= B & C; // D depends on B and C
  F <= E | D; // F depends on E and D
end
```

Hint: Given the nonblocking assignments, B, D, and F will all be updated in the next time step after all the expressions in the always block are evaluated, in parallel.

Simulation Execution Flow

With Nonblocking Assignments

- **Time step 0 (Initial State):**
  - Before execution: A = 0, B = 0, C = 1, D = 0, E = 1, F = 1
  - Evaluating nonblocking assignments (using old values):
    - B <= ~A = ~0 = 1
    - D <= B & C = 0 & 1 = 0
    - F <= E | D = 1 | 0 = 1
  - After update: B = 1, D = 0, F = 1
- **Time step 1 (Change in A, A becomes 1):**
  - Before execution: A = 1, B = 1, C = 1, D = 0, E = 1, F = 1
  - Evaluating nonblocking assignments (using old values):
    - B <= ~A = ~1 = 0
    - D <= B & C = 1 & 1 = 1
    - F <= E | D = 1 | 0 = 1
  - After update: B = 0, D = 1, F = 1

```
always @ (*)
begin
  B <= ~A;     // B depends on A
  D <= B & C; // D depends on B and C
  F <= E | D; // F depends on E and D
end
```

Simulation Execution Flow

With Nonblocking Assignments, Contd.

- **Time step 2 (Change in C, C becomes 0):**
  - Before execution: A = 1, B = 0, C = 0, D = 1, E = 1, F = 1
  - Evaluating nonblocking assignments (using old values):
    - B <= ~A = ~1 = 0
    - D <= B & C = 0 & 0 = 0
    - F <= E | D = 1 | 1 = 1
  - After update: B = 0, D = 0, F = 1
- **Time step 3 (Change in E, E becomes 0):**
  - Before execution: A = 1, B = 0, C = 0, D = 0, E = 0, F = 1
  - Evaluating non-blocking assignments (using old values):
    - B <= ~A = ~1 = 0
    - D <= B & C = 0 & 0 = 0
    - F <= E | D = 0 | 0 = 0
  - After update: B = 0, D = 0, F = 0

```
always @ (*)
begin
  B <= ~A;     // B depends on A
  D <= B & C; // D depends on B and C
  F <= E | D; // F depends on E and D
end
```

Simulation Execution Flow

With Nonblocking Assignments, Contd.

- **Time step 4 (Change in A, A becomes 0):**
  - Before execution: A = 0, B = 0, C = 0, D = 0, E = 0, F = 0
  - Evaluating non-blocking assignments (using old values):
    - B <= ~A = ~0 = 1
    - D <= B & C = 0 & 0 = 0
    - F <= E | D = 0 | 0 = 0
  - After update: B = 1, D = 0, F = 0

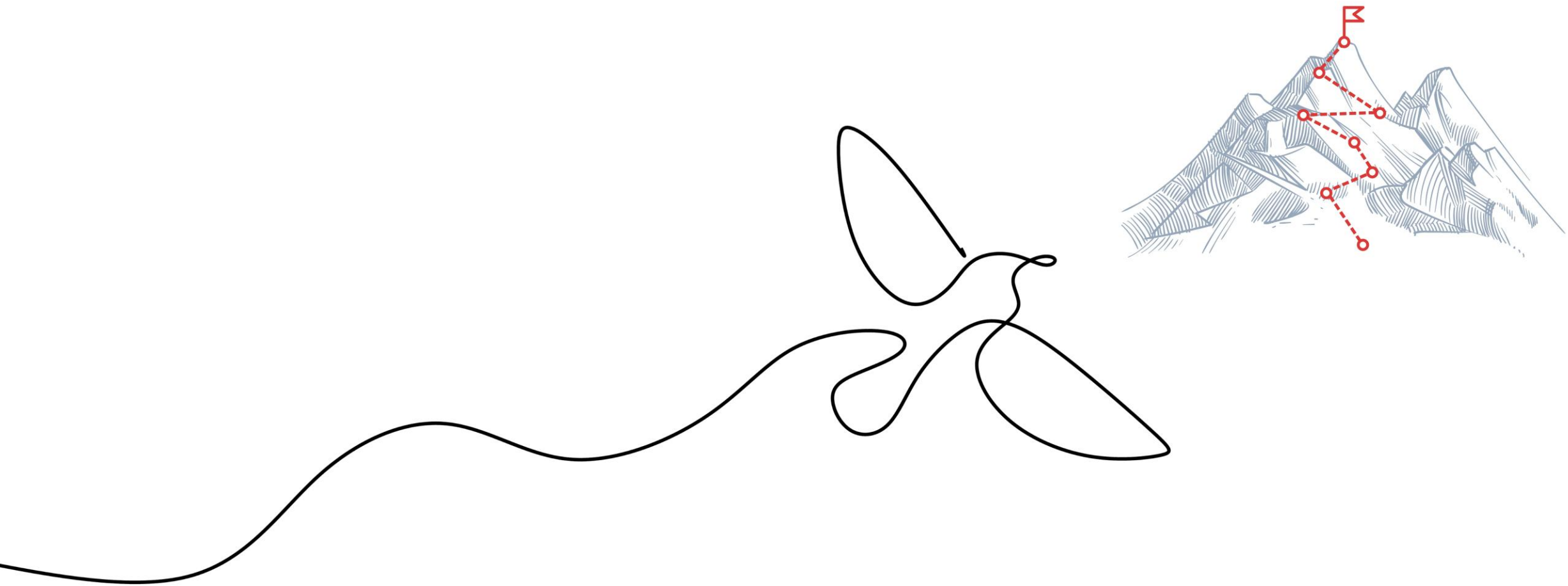
```
always @ (*)
begin
  B <= ~A;     // B depends on A
  D <= B & C; // D depends on B and C
  F <= E | D; // F depends on E and D
end
```

<= (Nonblocking) Assignments

In Verilog

Takeaways

- The nonblocking assignments ensure that each signal B/D/F in our example is updated in parallel at the **next** simulation time step, and that they don't affect each other within the same simulation time step
- Changes in A/C/E propagate through B/D/F, but only at the next simulation time step, which allows for parallel evaluation and update of all signals



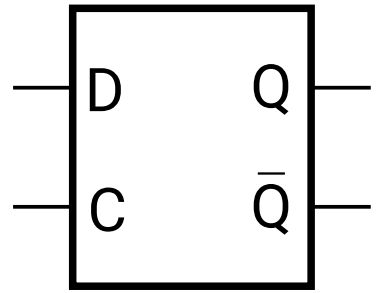
Behavioral Latch and Flip-Flop Models

Modeling Latches and FFs in Verilog

- Latches and FFs are typically specified via behavioral modeling
- Verilog compilers are designed to recognize very specific patterns for these behaviors, and the synthesis tools will infer the appropriate component

Basic D Latch

Behavioral Model




```
module Dlatch (
  input D,
  input C,
  output reg Q
  );

  always @ (*)
  begin
    if (C == 1)
      Q <= D;
  end

endmodule
```

- The output may be affected whenever inputs D or C change
  Hence, full sensitivity list: always @ (*)
- Note: the `if` statement does not have a corresponding `else` clause
  - **On purpose! Omitting the else clause means that we want to infer a latch**
    The Verilog simulator recognizes that Q should **not** change if C is 0; as a result, a latch is inferred
  - Equivalent to adding a redundant `else Q <= Q` clause

D Latch w/ Enable (CE) and Reset (CLR)

Behavioral Model

```
module Dlatch (
  input D,
  input C,
  input CE,
  input CLR,
  output reg Q
  );

  always @ (*)
  begin
    if (CLR == 1)
      Q <= 0;
    else if ((C == 1) && (CE == 1))
      Q <= D;
  end
endmodule
```

- The output may be affected whenever inputs D, C, CE, and CLR change

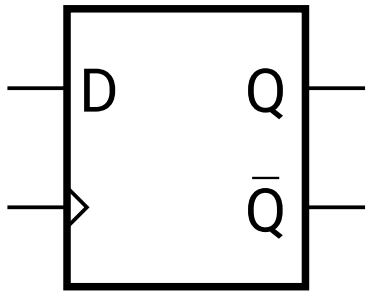
  Hence, full sensitivity list: always @ (*)

  - When asserted, CLR overrides other inputs
  - C and CE inputs have equivalent functions – they are ANDed to open the latch
- The `if` statement does not have a corresponding `else` clause, on purpose for the latch

`always @ (posedge Clock)`

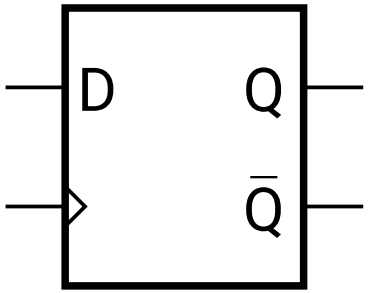
In Verilog

- Used to describe **sequential logic containing flip-flops (FFs)**
  - `always @ (posedge Clock)` – always at the rising clock edge
  - `always @ (negedge Clock)` – always at the falling clock edge
- Only `<=` (nonblocking) assignments should be used
  - Never use `=` (blocking) assignments
  - Recall that the logic symbol for an FF has a little wedge on the clock input. Have that remind you to use **`<=`** operator!



Positive-Edge-Triggered DFF

Behavioral Model




```
module Dff (
  input D,
  input CLK,
  output reg Q
  );

  always @ (posedge CLK)
  begin
    Q <= D;
  end

endmodule
```

- The `always` block is executed on the positive (rising) CLK edge
  - Q gets D
- Nothing happens at other times
  - Q is held at the same value at least until the next positive CLK edge

DFF with Synchronous Reset

Positive-Edge-Triggered, Reset is Active High

```
module Dff (
  input D,
  input CLK,
  input CLR,
  output reg Q
  );

  always @ (posedge CLK)
    if (CLR == 1)
      Q <= 0;
    else
      Q <= D;

endmodule
```

- On the rising edge of the CLK
  - Clear (reset) signal is evaluated
    - If high, output is cleared (zero)
    - Else, the output is unchanged
    - Here, reset is **active when high**
- If we wanted a negative-edge triggered DFF
  - Replace `posedge CLK` with `negedge CLK`

DFF with Asynchronous Reset

Positive-Edge-Triggered, Reset is Active High

```
module Dff (
  input D,
  input CLK,
  input CLR,
  output reg Q
  );

  always @ (posedge CLK or posedge CLR)
  begin
    if (CLR == 1)
      Q <= 0;
    else
      Q <= D;
  end

endmodule
```

- The sensitivity list now includes the reset input CLR
  - Whenever CLR is to be asserted (rising), the `always` block executes, and the `if` statement clears the Q output and exits
  - If the `always` block is executing and CLR was not asserted, then it must be executing because a positive edge occurred on CLK, and therefore, Q takes D

DFF with Asynchronous Reset

Positive-Edge-Triggered, Reset is Active High

```
module Dff (
  input D,
  input CLK,
  input CLR,
  output reg Q
  );

  always @ (posedge CLK or posedge CLR)
  begin
    if (CLR == 1)
      Q <= 0;
    else
      Q <= D;
  end

endmodule
```

- Why `posedge` CLR?
- It would be a mistake to omit `posedge` and execute `always` block on any change in CLR
  - Problem: On a '1'-to-'0' transition on CLR, the `else` clause would execute and set Q to D even though no CLK edge had occurred

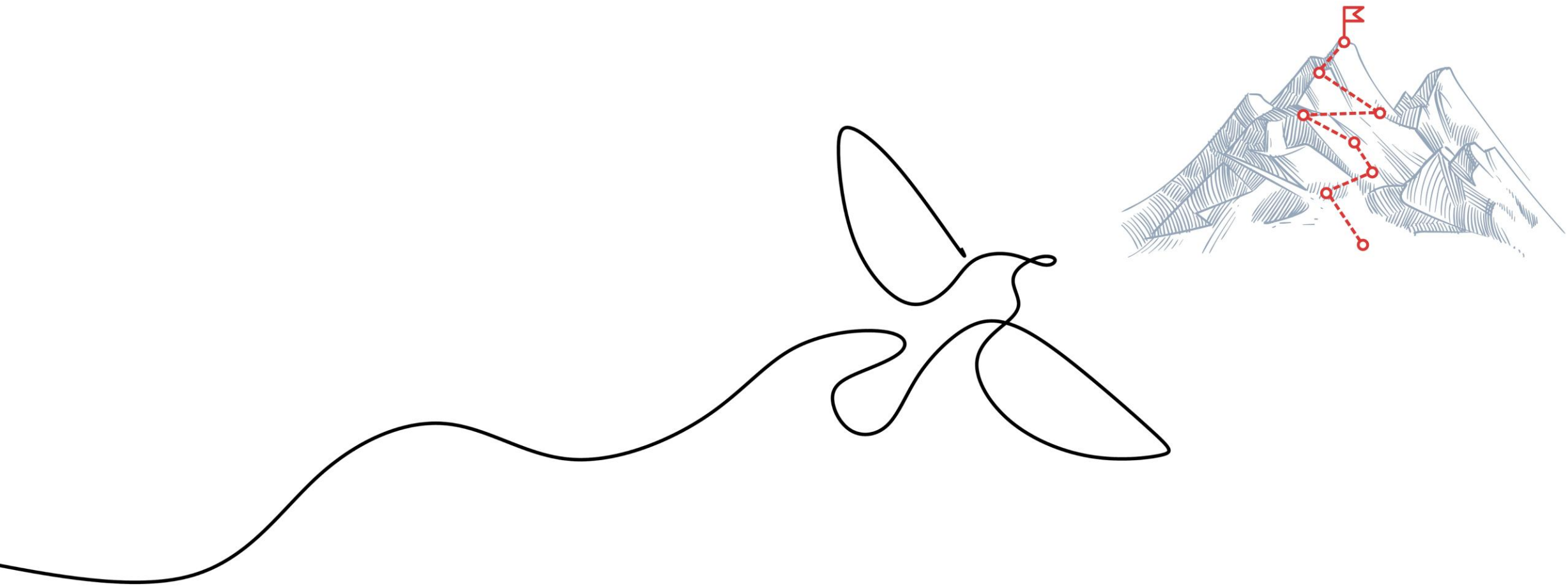
DFF with Asynchronous Reset and ~Q Output

Positive-Edge-Triggered, Reset is Active High, Contd.

```
module Dff (
  input D, CLK, CLR,
  output reg Q, QN
  );

  always @ (posedge CLK or posedge CLR)
  begin
    if (CLR == 1) begin
      Q <= 0;
      QN <= 1; // complementary output
    end
    else begin
      Q <= D;
      QN <= ~D; // complementary output
    end
  end
endmodule
```

- Asynchronous reset
- Two FF outputs
  - Original, Q
  - Complementary (negated)



Practically Useful Notes

Avoid Latches

- For practical sequential systems, avoid latches
  - Latches are sensitive to glitches (hazards)
  - Latch outputs can oscillate
  - Latches are level (not edge!) sensitive and may change output many times during one clock period
- **Use only D flip-flops and combinational logic**
  - Write independent `always` blocks
    - Some dedicated to D flip-flops (keep them extremely simple)
    - Some dedicated to combinational logic (as complex as needed)

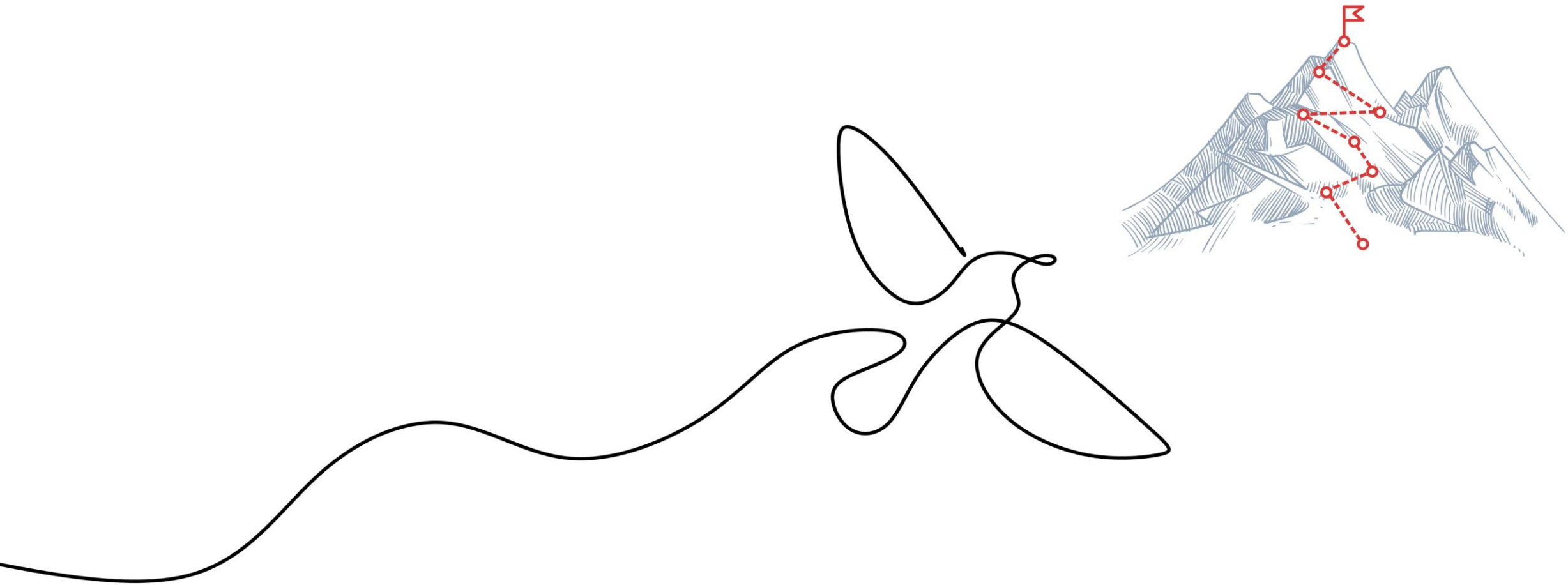
Beware, Latches can Sneak In

- To avoid latches to sneak in in your **combinational circuits**, make sure to assign every wire that can be assigned inside your `always@(*)` block
- Recommended practice:
  - Start your `always@(*)` block with the initialization of all wires to some default logic values to ensure they **all** take a value (1 or 0) regardless of the code that follows
  - Only afterward proceed with your desired assignments

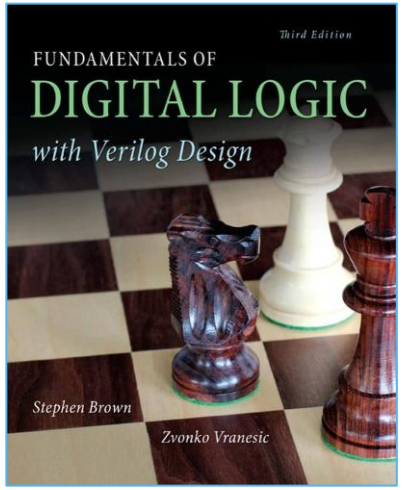
Nonblocking Assignments

- Always use nonblocking assignments in sequential `always` blocks (nonblocking assignment operator <=)
  - Why?

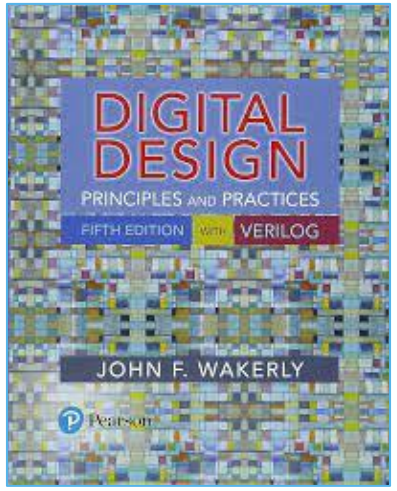
    In models with multiple sequential always blocks using blocking assignments, the simulation results can vary depending on the order in which the simulator chooses to execute those blocks. Using nonblocking assignments ensures that the righthand sides of all assignments are evaluated before new values are assigned to any of the lefthand sides. This makes the results independent of the order in which the righthand sides are evaluated



Literature



- Chapter 5: Fli-flops, Registers, and Counters
  - 5.1-4, 5.12, 5.13



- Chapter 10: Latches and Flip-Flops in Verilog
  - 10.3.2, 10.4.2

- Verilog: always@ Blocks by Chris Fletcher: https://inst.eecs.berkeley.edu/~eecs151/fa19/files/verilog/always_at_blocks.pdf