ECE 2300
Digital Logic & Computer Organization

Spring 2025

More Sequential Logic
Verilog

Cornell University

Announcements

- Lab 1 due tomorrow

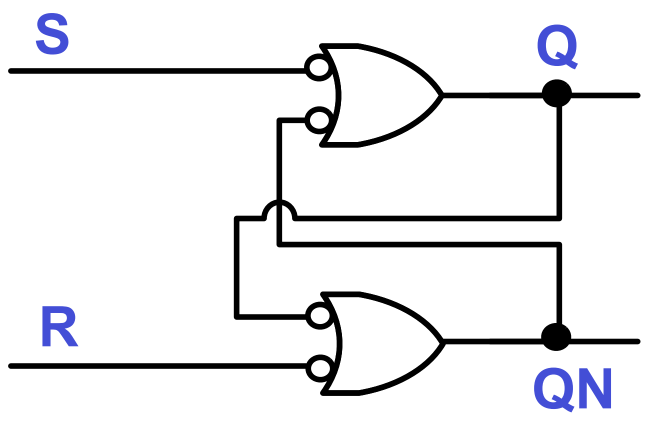
- Lab 2 will be released today

Sequential Logic: True or False

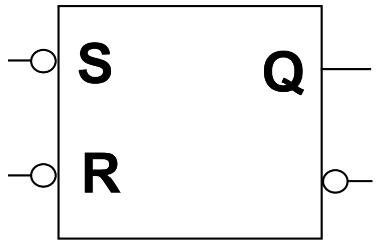
- 4 transistors are required to build an S-bar-R-bar latch
- A rising clock edge is also called a positive edge
- D latch changes its state when input changes
- D Flip-Flop is edge sensitive

$\overline{S}$-$\overline{R}$ Latch

- ***S-bar-R-bar* latch**
  - **Built from NAND gates**
  - **Inputs are active low rather than active high**
  - **When both inputs are 0, Q = QN = 1 (avoid!)**

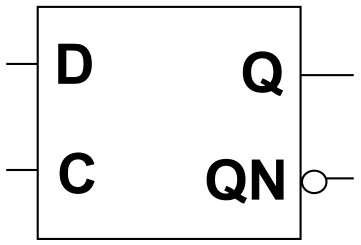


S	R	Q	QN
0	0	1	1
0	1	1	0
1	0	0	1
1	1	Last Q	Last QN

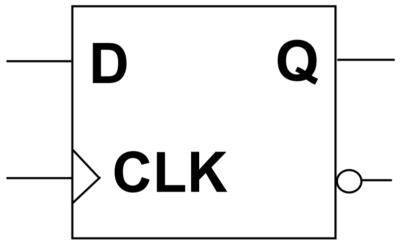


D Latch and Flip-Flop

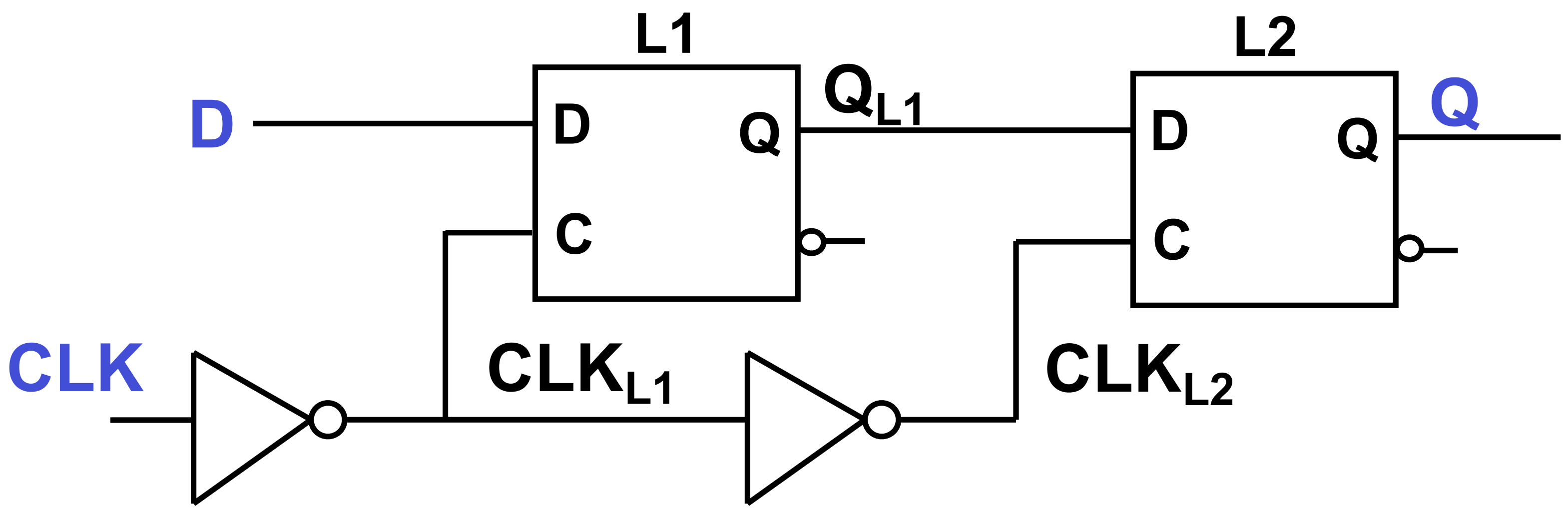
- D Latch: level sensitive
  - Captures the input when enable signal asserted

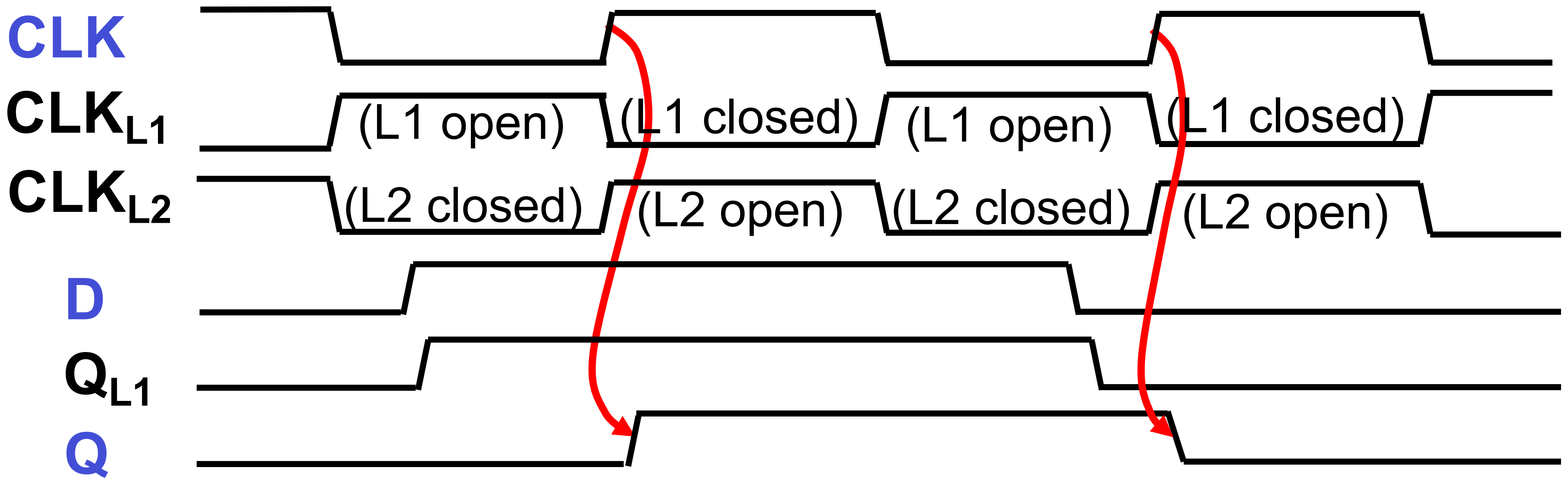


- D Flip-Flop (DFF): edge sensitive
  - Captures the input at the triggering clock edges (e.g., L→H)
  - A single FF is also called a one-bit register



Recap: DFF Timing

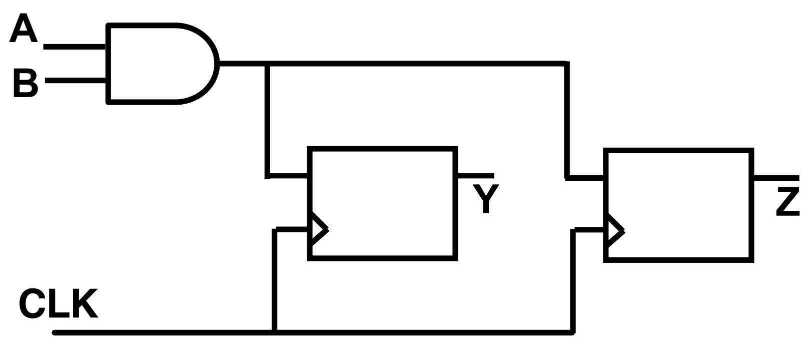




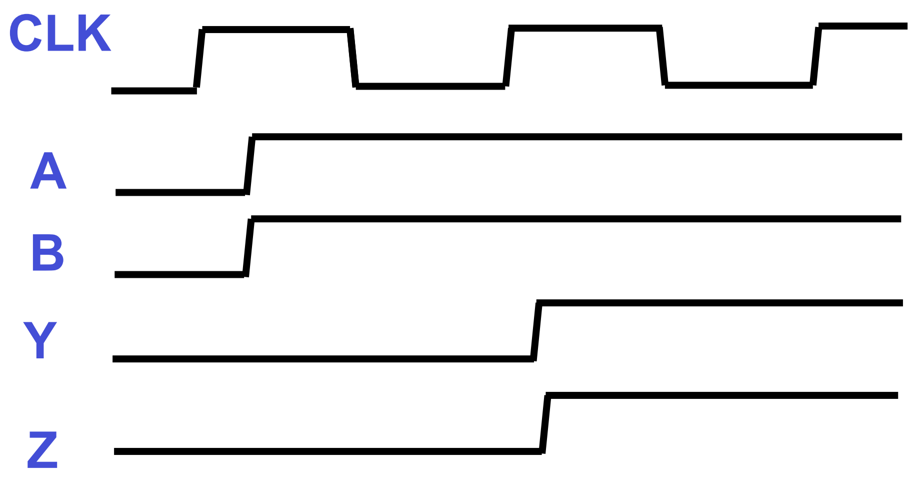
Input D copied to Q on the rising edge of the clock

Another DFF Timing Example

Circuit diagram

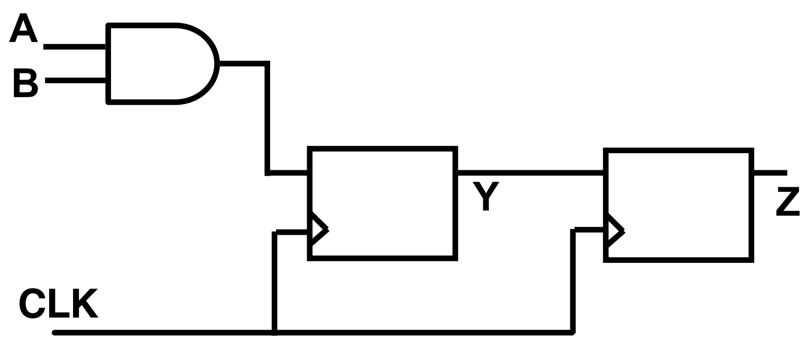


Waveform
(assume both DFFs hold 0s initially)



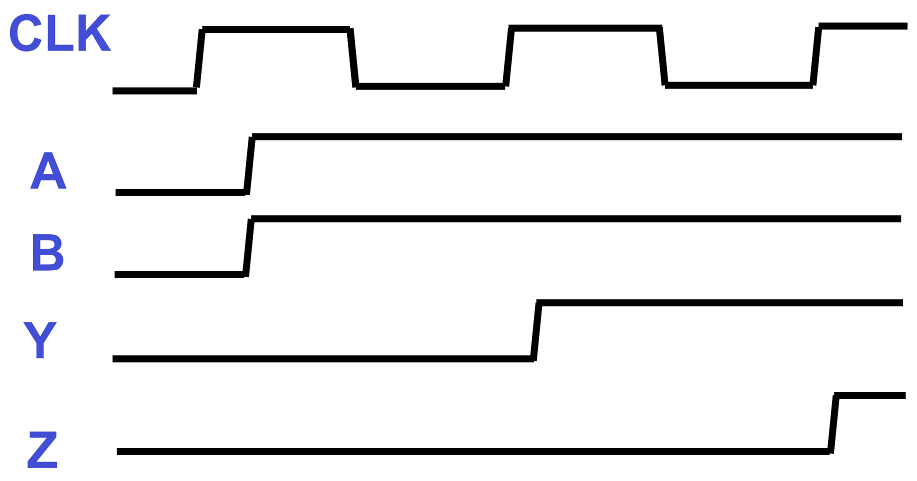
Yet Another DFF Timing Example

Circuit diagram



Waveform[1]

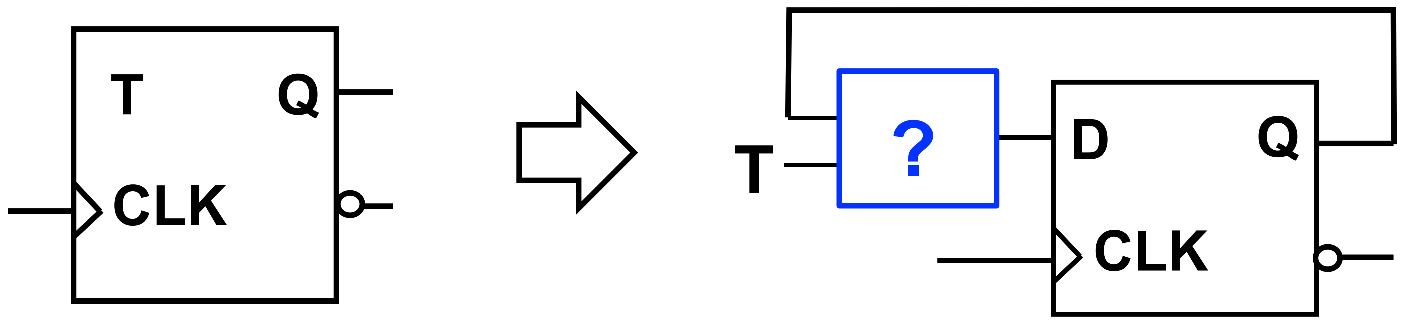
(assume both DFFs hold 0s initially)



[1] A DFF is often termed a “delay element” because it introduces a delay in data propagation. This delay results from updating the DFF state/output only on a clock edge. Beyond its important role as a storage element for holding states, the delay introduced by a DFF is also crucial for timing control and sequencing in digital systems.

T (*Toggle*) Flip-Flop

- Output toggles only if T=1
- Output does not change if T=0
- Useful for building counters



(when T=1)

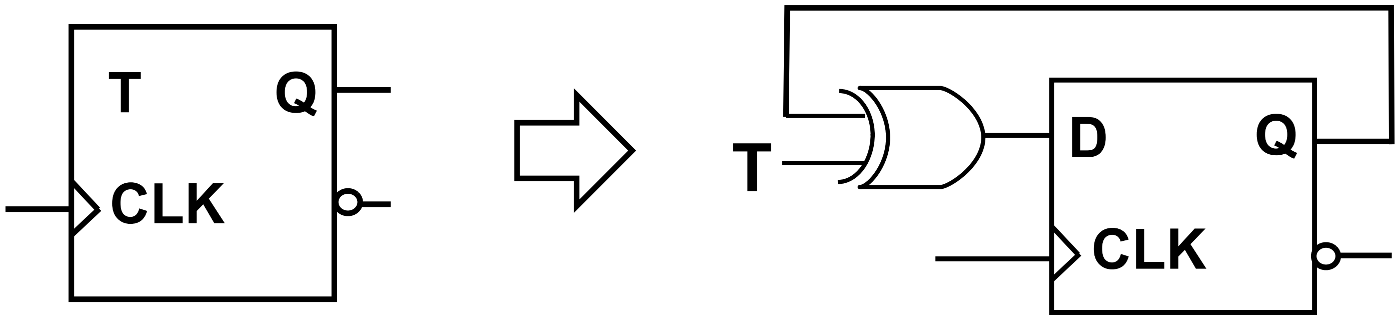
Q: 0, 1, 0, 1, 0, 1, 0, ...

Can we build a T flip-flop using a DFF as the building block?

T (*Toggle*) Flip-Flop

- **Output toggles only if T=1**
- **Output does not change if T=0**
- **Useful for building counters**

T	Q	Q_{next}
0	0	0
0	1	1
1	0	1
1	1	0



(when T=1)

Q: 0, 1, 0, 1, 0, 1, 0, ...

$$Q_{next} = T \bullet Q' + T' \bullet Q$$

Binary Counters

- Counts in binary in a particular sequence
- Advances at every tick of the clock
- Many types

Up	Down	Divide-by-n	n-to-m
0 0 0	1 1 1	0 0 0	n
0 0 1	1 1 0	0 0 1	n+1
0 1 0	1 0 1	0 1 0	n+2
0 1 1	1 0 0	0 1 1	⋮
1 0 0	0 1 1	1 0 0	m-1
1 0 1	0 1 0	⋮	m
⋮	⋮	n-1	n
		0 0 0	n+1
		0 0 1	⋮

Up Counter Sequence

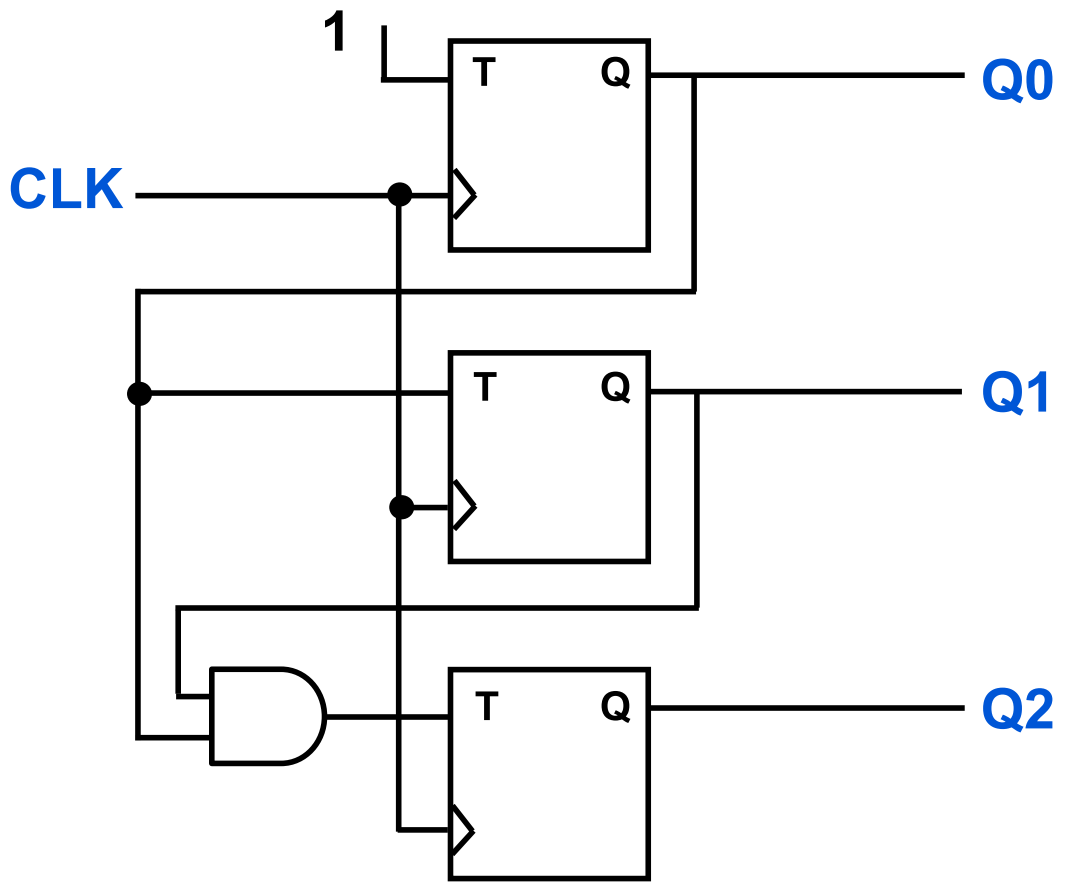
```
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

Toggles every clock tick that two right bits = 11

Toggles every clock tick that right bit = 1

Toggles every clock tick

Building Binary Up Counter



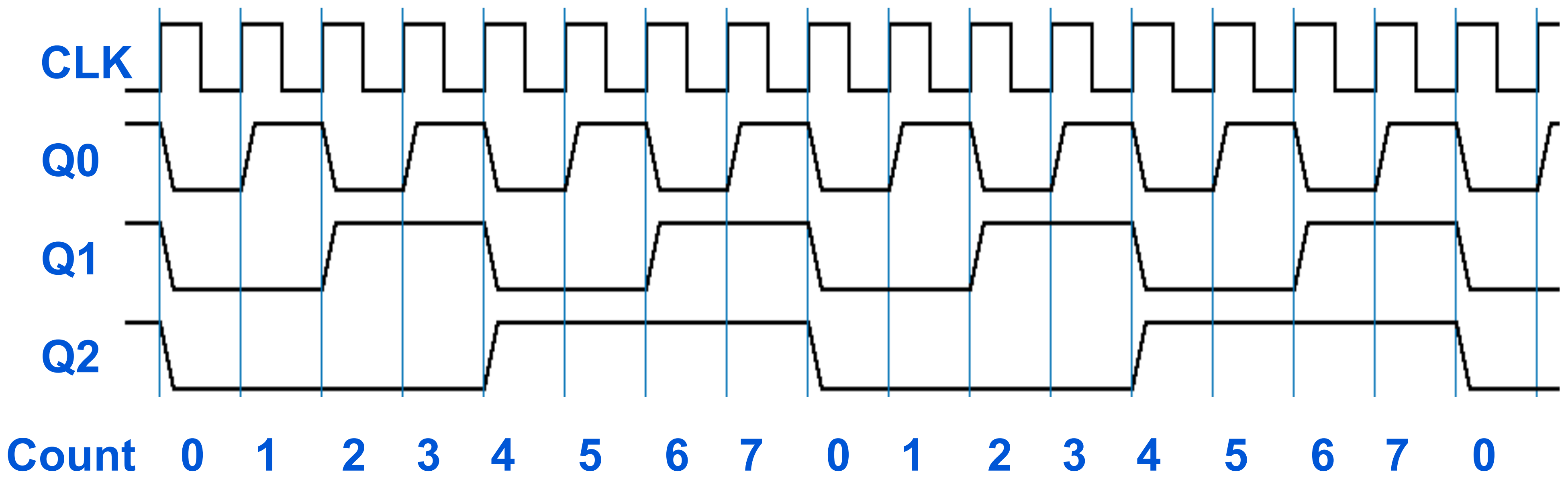
Q_2	Q_1	Q_0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1
0	0	0
0	0	1

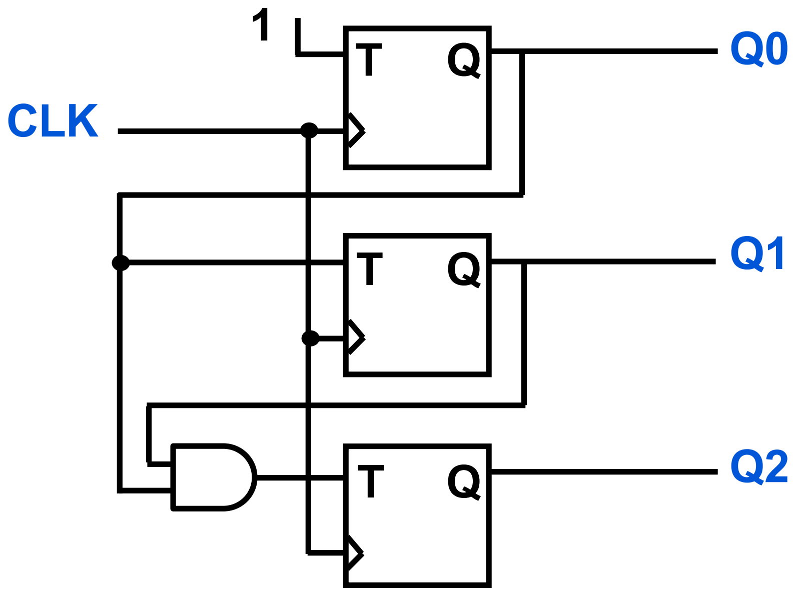
Q0 toggles at every rising edge

Q1 toggles at the rising edge when Q0=1

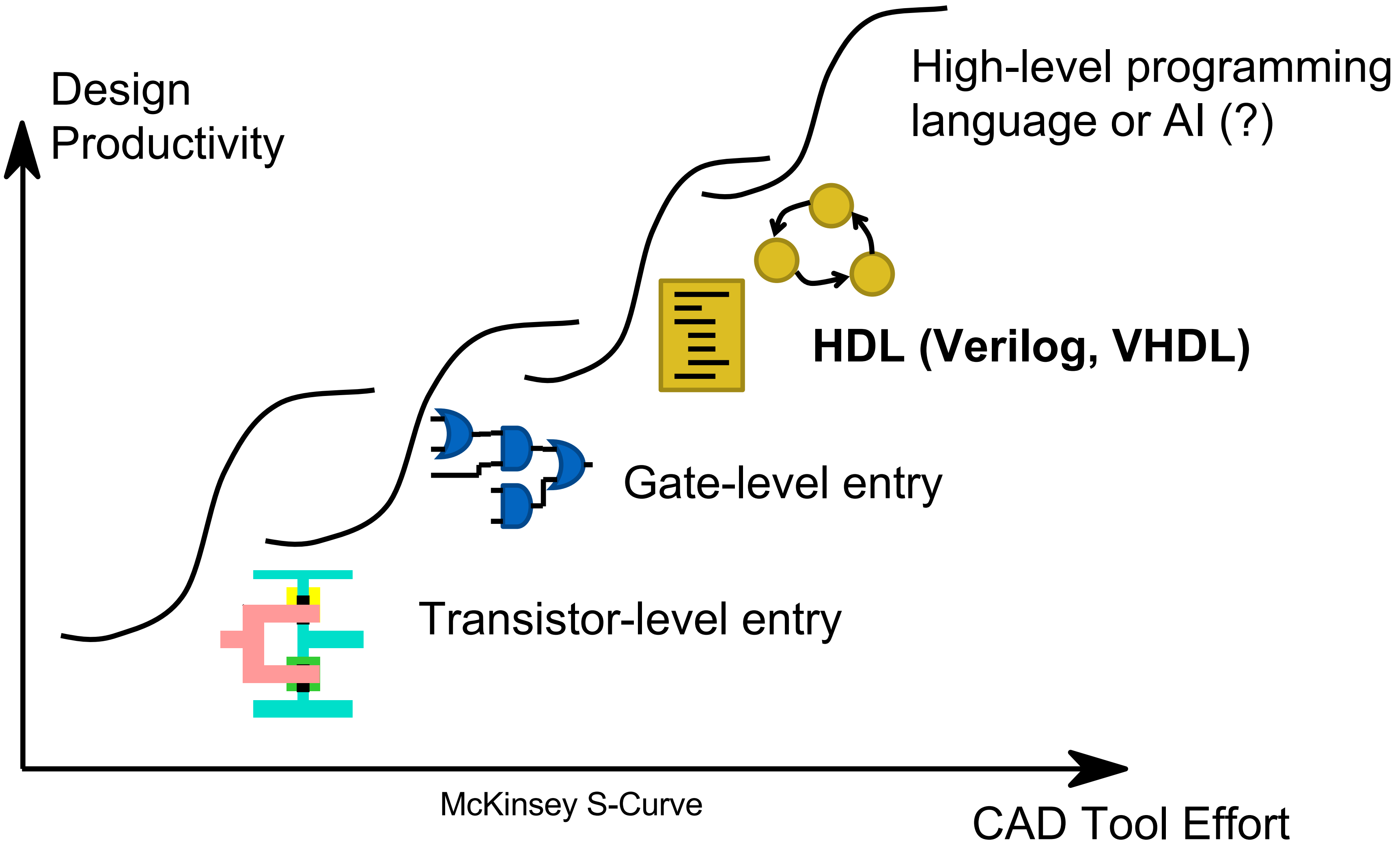
Q2 toggles at the rising edge when Q0=Q1=1

Up Counter Timing Diagram





Evolution of Design Abstractions



[Figure credit: Kurt Keutzer]

Hardware Description Languages

- Hardware Description Language (HDL): a language for describing hardware
  - Efficiently code large, complex designs
    - Programming at a more abstract level than schematics
  - CAD tools can automatically synthesize circuits

- Industry standards:
  - Verilog: We start using it from Lab 2
  - SystemVerilog: Successor to Verilog, gaining wide adoption
  - VHDL (Very High Speed Integrated Circuit HDL)

Verilog

- Developed in the early 1980s by Gateway Design Automation (later bought by Cadence)

- Supports modeling, simulation, and synthesis
  - Simulation verifies the functionality of the design by executing the model (i.e., Verilog design) and testing its behavior over time
  - Synthesis converts the Verilog design into an optimized circuit for implementation on physical hardware
  - We will use a (synthesizable) subset of the language features

- Major language features (in contrast to software programming languages)
  - Structure and instantiation
  - Concurrency
  - Bit-level behavior

Values

- **Verilog signals can take 4 values** (for simulation purpose)

  0 Logical 0, or false

  1 Logical 1, or true

  x Unknown logical value

  z High impedance (Hi-Z), floating/non-connected

x means unknown/uninitialized (could be 0, 1, z, or in transition) or don't cares

Bit Vectors

- Multi-bit values are represented by bit vectors (i.e., grouping of 1-bit signals)
  - Right-most bit is always least significant
  - Examples:

```
input a; /* 1-bit input */
input[7:0] a, b, c; /* three 8-bit inputs */
```

- Constants

4’b1001

Base format (b,d,h,o)

Decimal number representing bit width

- Binary Constants
  - 8’b00000000
  - 8’b0xx01xx1
- Decimal Constants
  - 4’d10
  - 32’d65536

Operators

- **Bitwise Boolean operators**

  ~ NOT

  & AND

  ^ Exclusive OR

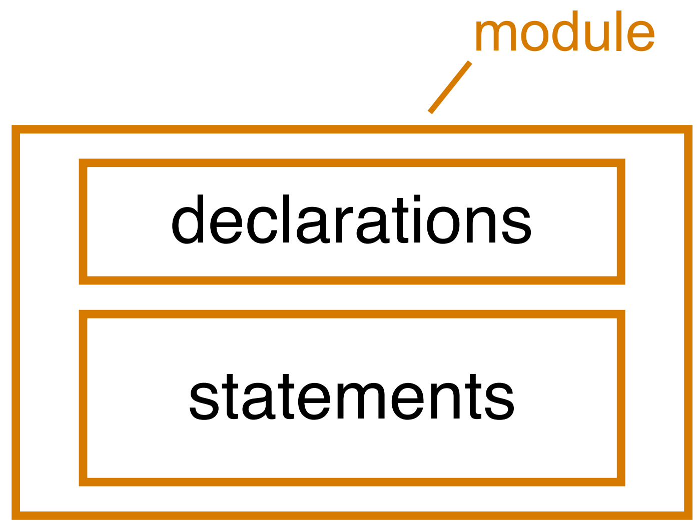
  | OR

- **Arithmetic operators**

+ Addition	/ Division	<< Shift left
– Subtraction	% Modulus	>> Shift right
* Multiplication		

Verilog Program Structure

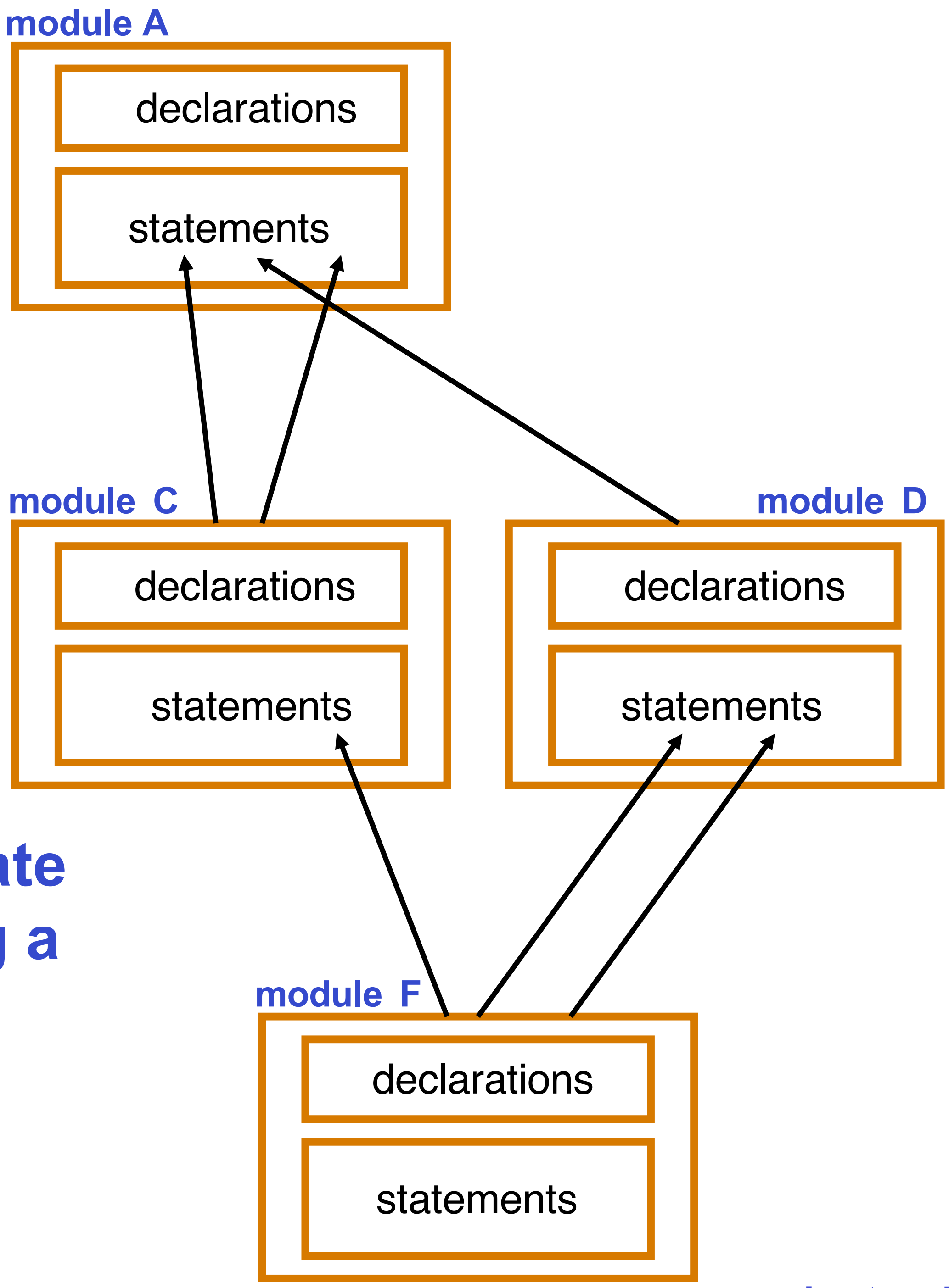
- System is a collection of *modules*
  - Module represents a hardware component or a design unit



- Declarations
  - Describe names and types of inputs and outputs
  - Describe local signals, variables, constants, etc.

- Statements specify what the module does

Verilog Module Hierarchy



A module can instantiate other modules forming a module hierarchy

Example: Verilog Program Structure

```
module M_2_1 (a, b, sel, Y);
  input  a, b;
  input  sel;
  output Y;
  wire ta, tb;

  AND and0 (a, ~sel, ta);
  AND and1 (b,  sel, tb);
  OR  or0  (ta, tb, Y);

endmodule
```

Declarations

Statements
(instantiating two AND gates and one OR gate)

Verilog Programming Styles

- **Structural**
  - Describe how a module is built from other modules and their interconnections via instance statements
  - Textual equivalent of drawing a schematic

- **Behavioral**
  - Specify what a module does in high-level constructs
  - Use continuous assignments and/or procedural code (in always blocks) to indicate what actions to take

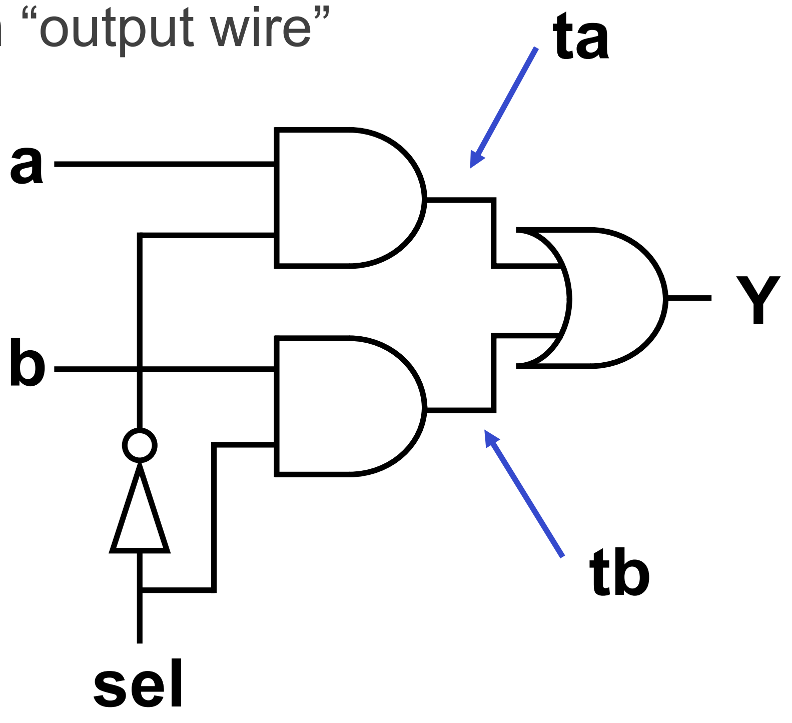
We can mix the structural and behavioral styles in a Verilog design

Net and Variable Types

- We will mainly use two data type classes
  - *wire*: represents a physical connection (also called net) between hardware elements
    - A stateless way of connected two elements
    - Can only be used to model combinational logic
    - Cannot be used in the left-hand side (LHS) in an always block
  - *reg*: similar to wires, but can be used to store information (or state) like registers
    - This is used in the behavioral style only
    - *Can be used to model both combinational & sequential logic*
    - Cannot be used in the LHS of a continuous assignment statement

Structural Style

```
module MUX_2_1 (a, b, sel, Y);
  input  a, b, sel; // here “input” is same with “input wire”
  output Y; // here “output” is same with “output wire”

  wire ta, tb;

  AND and0 (a, ~sel, ta);
  AND and1 (b,  sel, tb);
  OR  or0 (ta, tb, Y);

endmodule
```



The order of the module instantiation does not matter, essentially describing the schematic textually

Behavioral Style with Continuous Assignments

- An *assign* statement represents continuously executing combinational logic

```
module MUX_2_1 (a, b, sel, Y);
  input  a, b, sel;
  output Y;

  assign Y = (~sel & a) | (sel & b);

endmodule
```

continuous assignment

- Multiple continuous assignments happen in parallel; the order does not matter

Always Blocks

```
always @(<sensitivity_list>)
begin
  <procedural assignments>
end
```

- An ***always block*** is a procedural construct that executes whenever there is a change in the specified sensitivity list
  - Can model either combinational or sequential logic
  - Sequential logic can only be modeled using always blocks

- Always blocks execute concurrently with other always blocks, instance statements, and continuous assignment statements in a module

Procedural Statements in Always Block

- **Procedural statements are similar to conventional programming language statements**
  - **begin-end blocks**
    - begin *procedural-statement* … *procedural-statement* end
  - **if**
    - if (*condition*) *procedural-statement* else *procedural-statement*
  - **case**
    - case (*sel-expr*) *choice* : *procedural-statement* … endcase
  - **for**
    - for(initial_assignment; expression; step_assignment) statement;
  - **while**
    - while (*logical-expression*) *procedural-statement*
  - **repeat**
    - repeat (*integer-expression*) *procedural-statement*

Mostly used in test bench
(simulation only, not synthesizable)

(Behavioral Style) Combinational Logic with Always Blocks

```
module MUX_2_1 (a, b, sel, Y);
  input  a, b, sel;
  output reg Y;

  always @(a, b, sel)
  begin
    Y = (~sel & a) | (sel & b);
  end
endmodule
```

sensitivity list

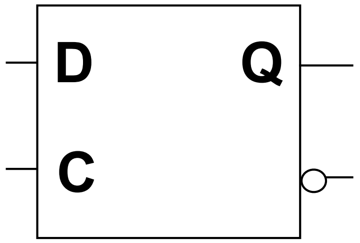
procedural assignment

- **An *always* block is activated whenever a signal in its <u>sensitivity list</u> changes**
  - **Formed by <u>procedural assignment</u> statements**
  - **The left-hand side of a procedural assignment must be declared as a "reg"**

Sequential Logic with Always Blocks

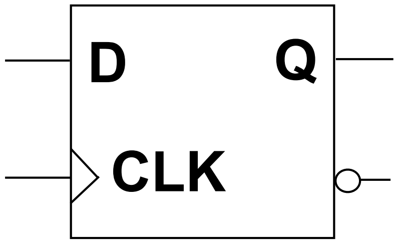
Sequential logic can ONLY be modeled using always blocks

```
reg Q;

always @(clk, D)
begin
  if ( clk )
      Q = D;
end
```



D latch

```
reg Q;

always @(posedge clk)
begin
  Q = D;
end
```



DFF

Q is declared as a "reg" since it appears on the left-hand side of a procedural assignment

Next Class

More Verilog

Finite State Machines

(H&H 3.4)