EECS 151/251A
Spring 2019
Digital Design and Integrated Circuits

Instructors:
Wawrzynek

Lecture 4

Administrativia

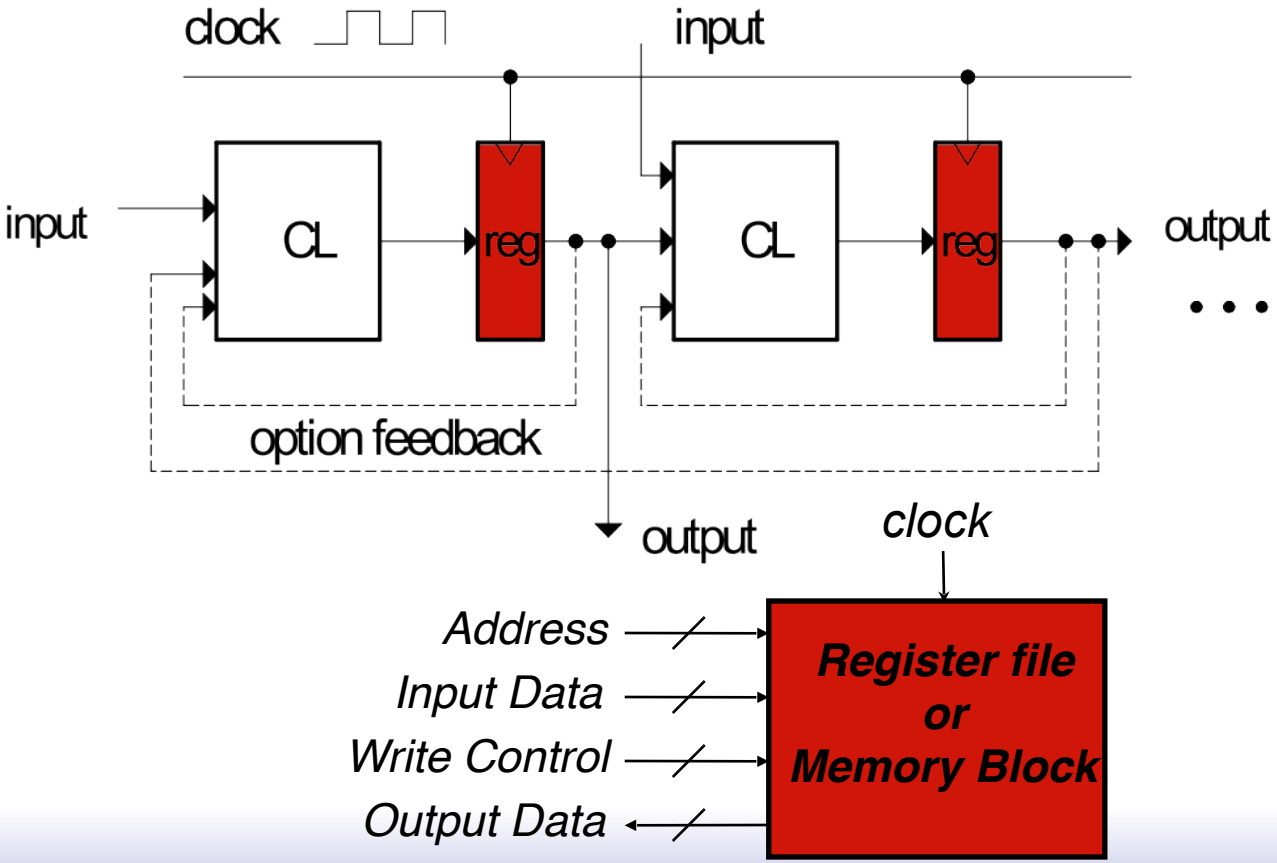
Verilog – So Far

- Combinational Logic Specification
- Two types of description:
  - Structural: design as a composition of blocks (also called a netlist)
    - Maps directly into hardware
  - Behavioral: design as a set of equations
    - Requires “compiler” (synthesis tool) to generate hardware

Sequential Elements

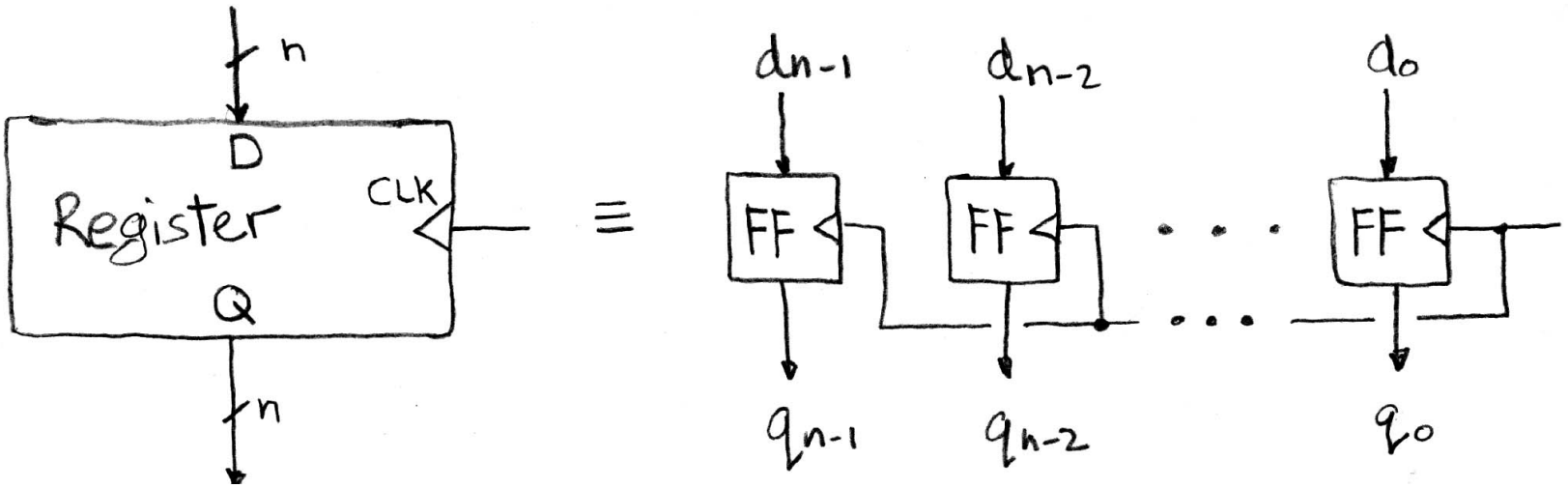
Only Two Types of Circuits Exist

- Combinational Logic Blocks (CL)
- State Elements (registers)



- State elements are mixed in with CL blocks to control the flow of data.
- Sometimes used in large groups by themselves for "long-term" data storage.

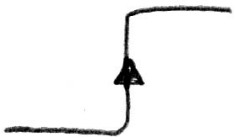
Register Details…What's inside?



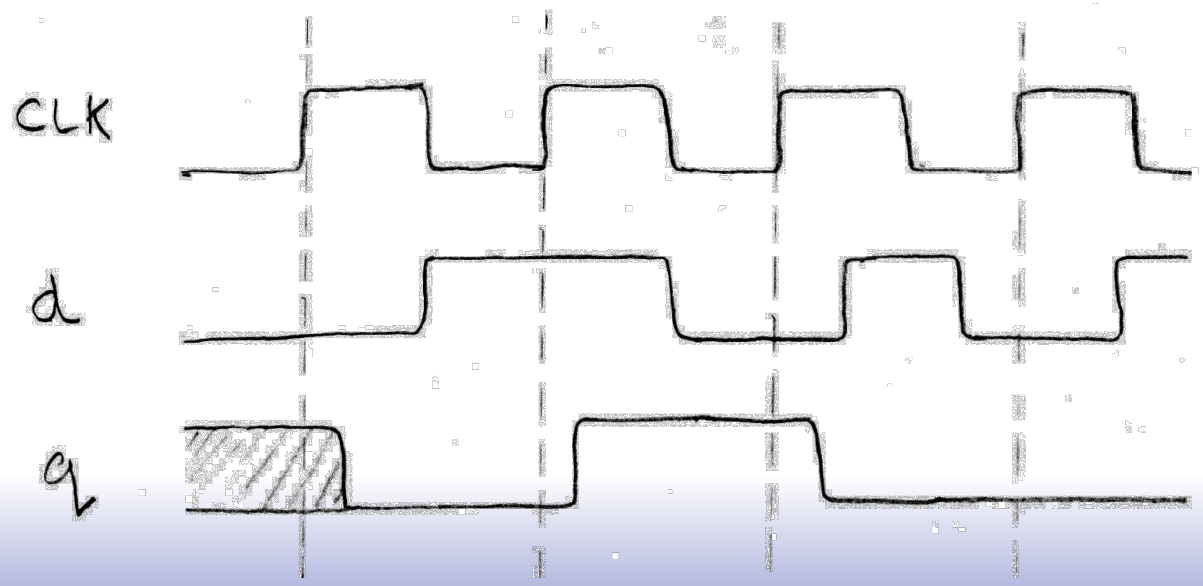
- n instances of a "Flip-Flop"
- Flip-flop name because the output flips and flops between 0 and 1
- D is "data", Q is "output"
- Also called "d-type Flip-Flop"

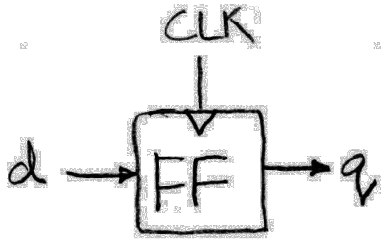
Flip-flop Timing Waveforms

- Edge-triggered d-type flip-flop
  - This one is “positive edge-triggered”



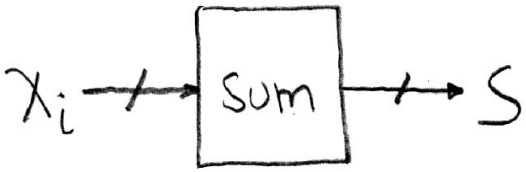
- “On the rising edge of the clock, the input d is sampled and transferred to the output. At all other times, the input d is ignored.”
- Example waveforms:





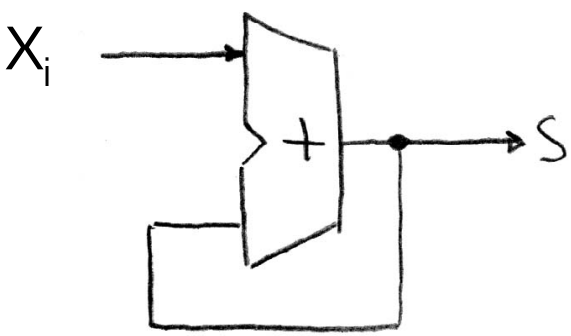
Accumulator Example

Assume X is a vector of N integers, presented to the input of our accumulator circuit one at a time (one per clock cycle), so that after N clock cycles, S hold the sum of all N numbers.



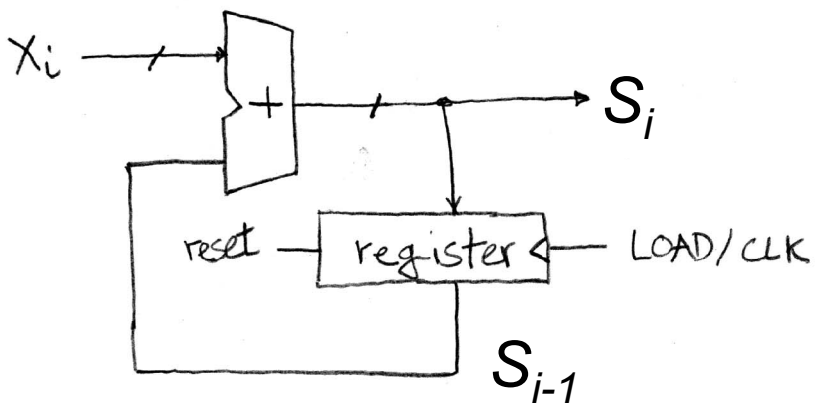
S=0; Repeat N times
S = S + X;

- We need something like this:



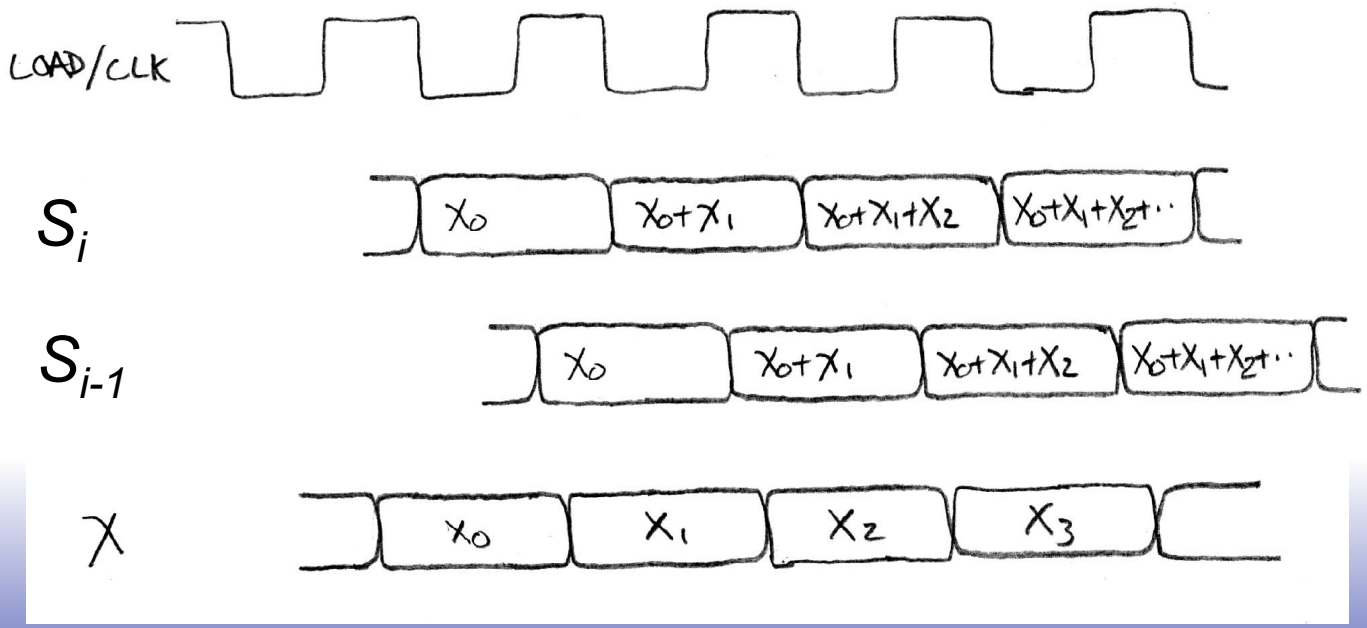
- *But not quite.*
- *Need to use the clock signal to hold up the feedback to match up with the input signal.*

Accumulator

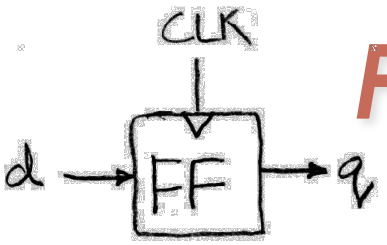


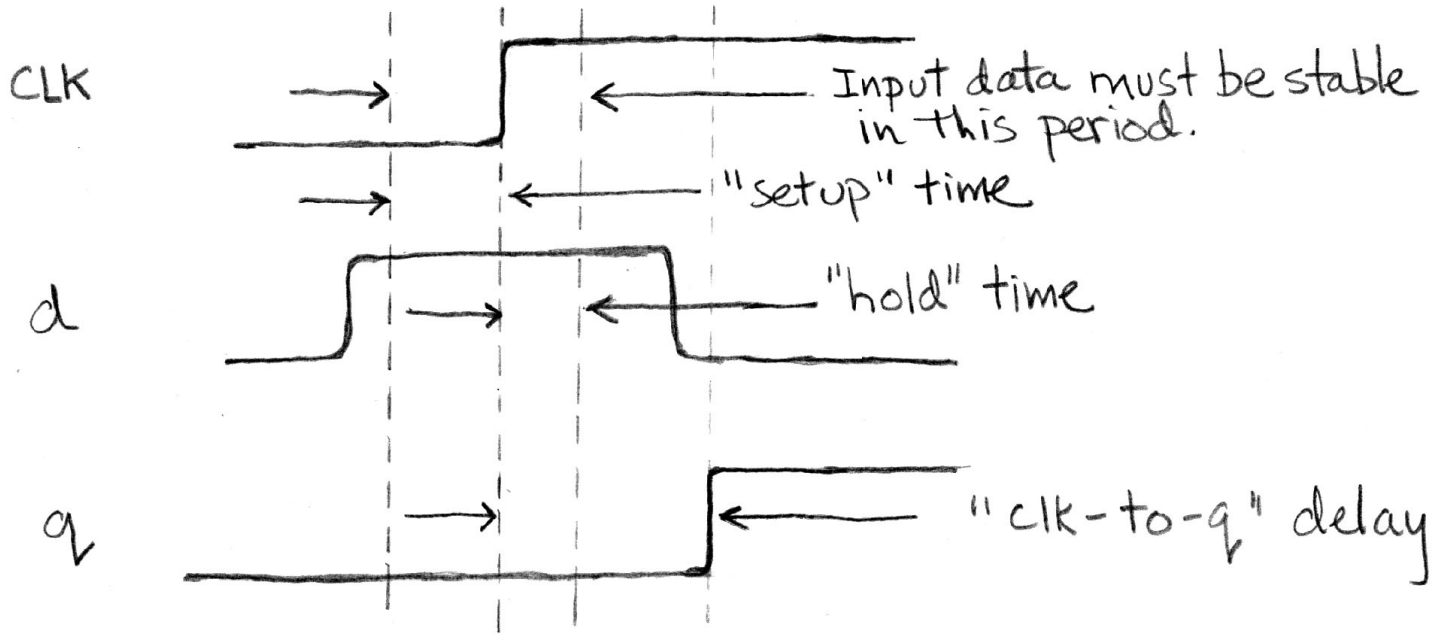
- Put register in feedback path.
- On each clock cycle the register prevents the new value from reaching the input to the adder prematurely. (The new value just waits at the input of the register)

Timing:



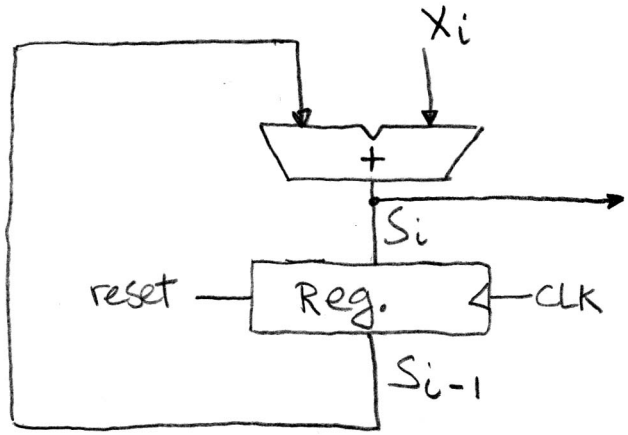
Flip-Flop Timing Details

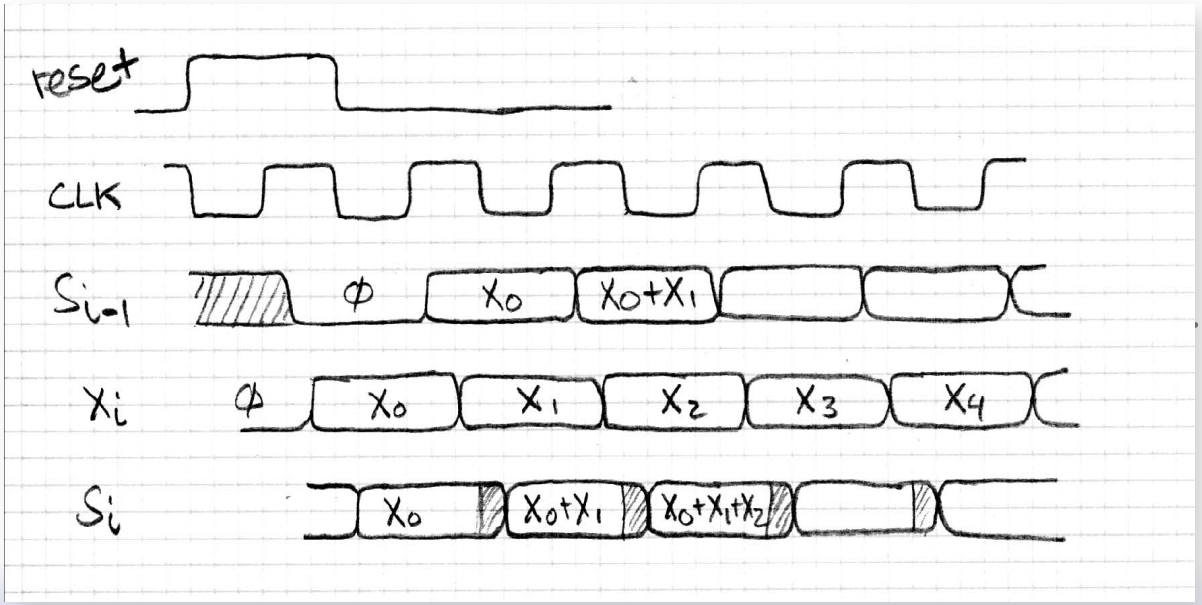




- Three important times associated with flip-flops:
  - Setup time - *How long d must be stable before the rising edge of CLK*
  - Hold time - *How long D must be stable after the rising edge of CLK*
  - Clock-to-q delay – *Propagation delay after rising edge of the CLK*

Accumulator Revisited



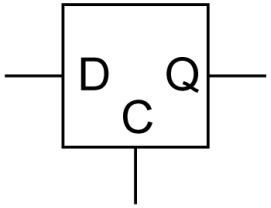


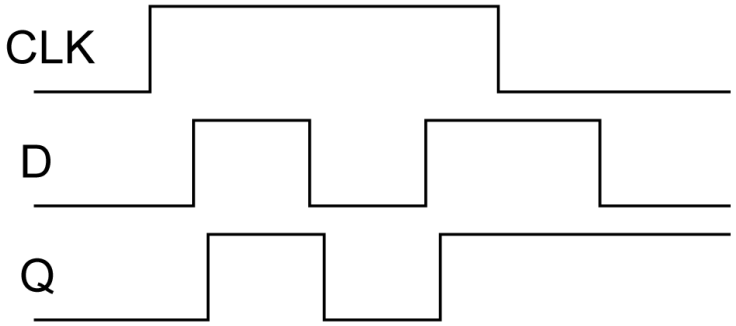
- Note:
  - Reset signal (synchronous)
  - Timing of X signal is not known without investigating the circuit that supplies X. Here we assume it comes just after S_{i-1}.

Observe transient behavior of S_i.

Level-sensitive Latch Inside Flip-flop

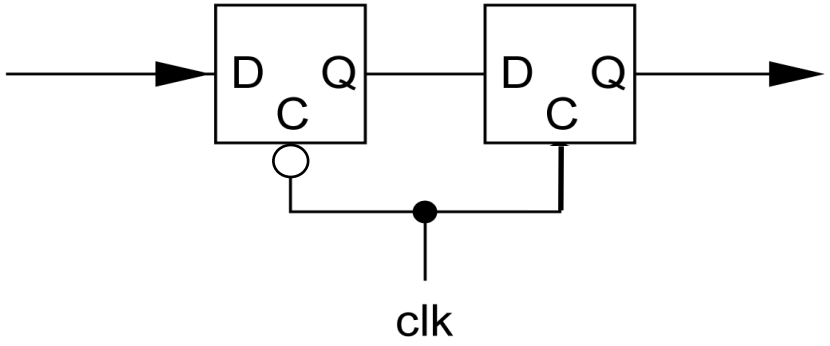
Positive Level-sensitive latch:





When CLK is high, latch is transparent, when clk is low, latch retains previous value.

Positive Edge-triggered flip-flop built from two level-sensitive latches:



Sequential Elements in Verilog

State Elements in Verilog

Always blocks are the only way to specify the "behavior" of state elements. Synthesis tools will turn state element behaviors into state element instances.

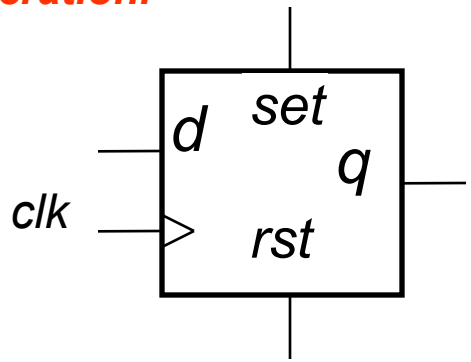
D-flip-flop with synchronous set and reset example:

```
module dff(q, d, clk, set, rst);
  input d, clk, set, rst;
  output q;
  reg q;

  always @(posedge clk)
    if (rst)
      q <= 1'b0;
    else if (set)
      q <= 1'b1;
    else
      q <= d;
endmodule
```

keyword

"always @ (posedge clk)" is key to flip-flop generation.

This gives priority to reset over set and set over d.



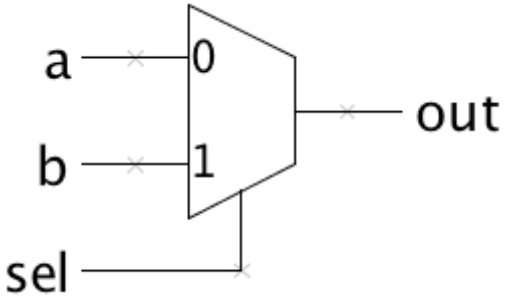
On FPGAs, maps to native flip-flop.

Unlike logic gates, their are no primitive flip-flops in Verilog. Although, it is possible to instantiate FPGA or Standard-cell specific flip-flops.

The Sequential always Block

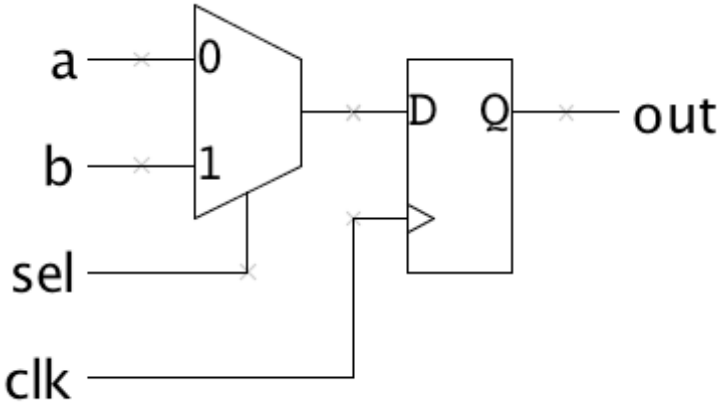
Combinational

```
module comb(input a, b, sel,
            output reg out);
  always @(*) begin
    if (sel) out = b;
    else out = a;
  end
endmodule
```



Sequential

```
module seq(input a, b, sel, clk,
           output reg out);
  always @(posedge clk) begin
    if (sel) out <= b;
    else out <= a;
  end
endmodule
```



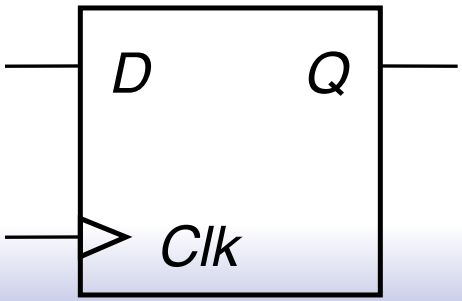
Latches vs. Flip-Flops

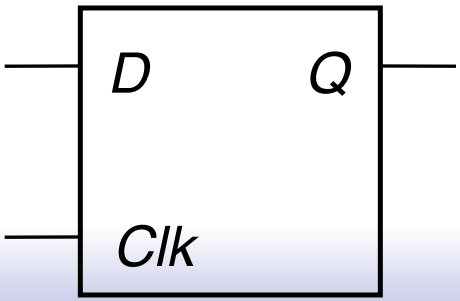
Flip-Flop

```
module flipflop
(
   input clk,
   input d,
   output reg q
);

  always @(posedge clk)
  begin
    q <= d;
  end
endmodule
```

Latch

```
module latch
(
   input clk,
   input d,
   output reg q
);

  always @(clk or d)
  begin
    if ( clk )
      q <= d;
  end
endmodule
```





Importance of the Sensitivity List

- The use of posedge and negedge makes an always block sequential (edge-triggered)

D-Register with synchronous clear

```
module dff_sync_clear(
  input d, clearb, clock,
  output reg q);

  always @(posedge clock)
    begin
      if (!clearb) q <= 1'b0;
      else q <= d;
    end
endmodule
```

always block entered only at each positive clock edge

D-Register with asynchronous clear

```
module dff_async_clear(
  input d, clearb, clock,
  output reg q);

  always @(negedge clearb or posedge clock)
    begin
      if (!clearb) q <= 1'b0;
      else q <= d;
    end
endmodule
```

always block entered immediately when (active-low) clearb is asserted

Note: The following is incorrect syntax: `always @(clear or negedge clock)`

If one signal in the sensitivity list uses posedge/negedge, then all signals must.

- *Assign any signal or variable from only one always block.*

Be wary of race conditions: always blocks with same trigger execute concurrently…

Blocking vs. Nonblocking Assignments

- Verilog supports two types of assignments within always blocks, with subtly different behaviors.
- *Blocking assignment (=):* evaluation and assignment are immediate

```
always @(*) begin
  x = a | b;      // 1. evaluate a|b, assign result to x
  y = a ^ b ^ c;  // 2. evaluate a^b^c, assign result to y
  z = b & ~c;     // 3. evaluate b&(~c), assign result to z
end
```

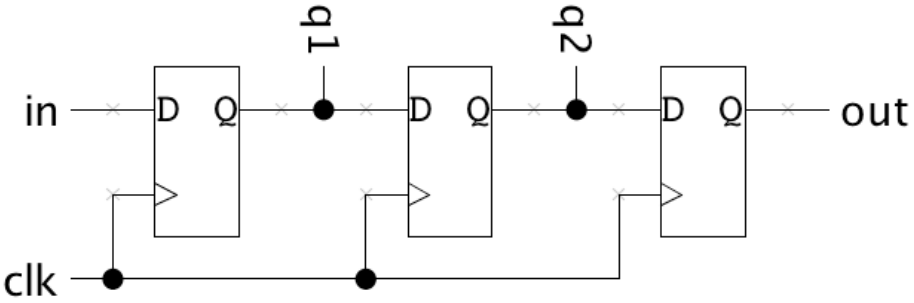
- *Nonblocking assignment (<=):* all assignments deferred to end of simulation time step after all right-hand sides have been evaluated (*even those in other active* always *blocks*)

```
always @(*) begin
  x <= a | b;     // 1. evaluate a|b, but defer assignment to x
  y <= a ^ b ^ c; // 2. evaluate a^b^c, but defer assignment to y
  z <= b & ~c;    // 3. evaluate b&(~c), but defer assignment to z
  // 4. end of time step: assign new values to x, y and z
end
```


Sometimes, as above, both produce the same result. Sometimes, not!

Assignment Styles for Sequential Logic

What we want:
Register Based Digital Delay Line (a.k.a. shift-register)



Will nonblocking and blocking assignments both produce the desired result?

```
module nonblocking(
  input in, clk,
  output reg out
);
  reg q1, q2;
  always @(posedge clk) begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end

endmodule
```

```
module blocking(
  input in, clk,
  output reg out
);
  reg q1, q2;
  always @(posedge clk) begin
    q1 = in;
    q2 = q1;
    out = q2;
  end

endmodule
```

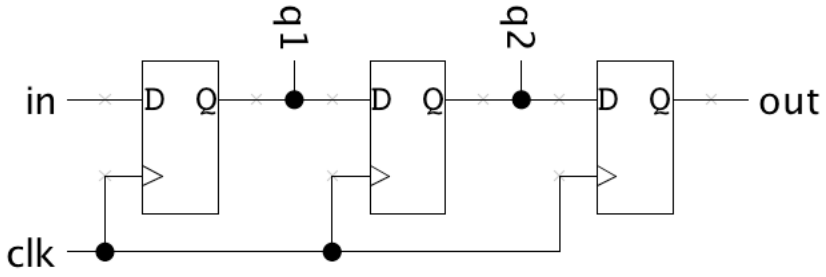
Use Nonblocking for Sequential Logic

```
always @(posedge clk) begin
    q1 <= in;
    q2 <= q1;    // uses old q1
    out <= q2;   // uses old q2
  end
```

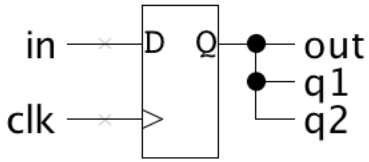
```
always @(posedge clk) begin
    q1 = in;
    q2 = q1;    // uses new q1
    out = q2;   // uses new q2
  end
```

("old" means value before clock edge, "new" means the value after most recent assignment)

"At each rising clock edge, q1, q2, and out simultaneously receive the old values of in, q1, and q2."

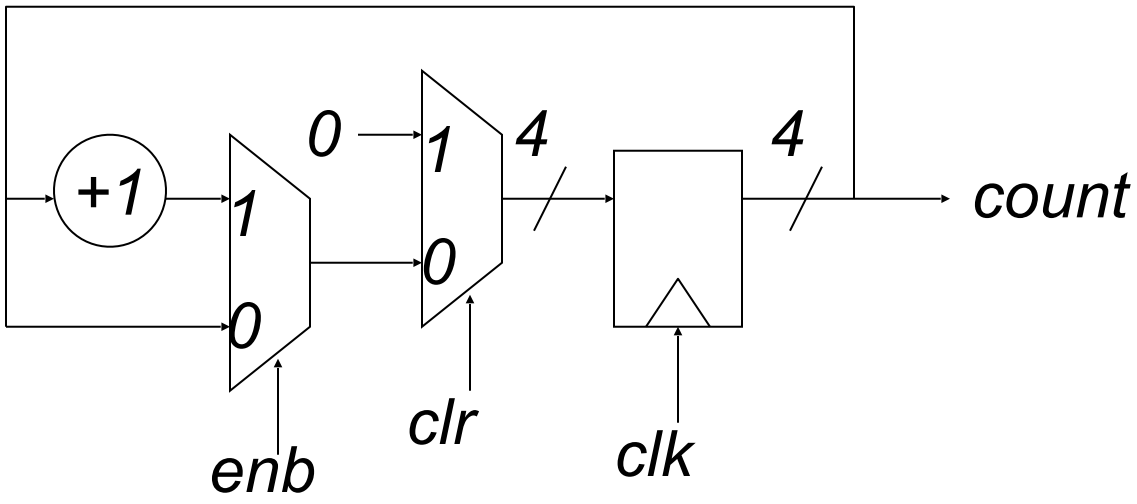


"At each rising clock edge, q1 = in. After that, q2 = q1. After that, out = q2. Therefore out = in."



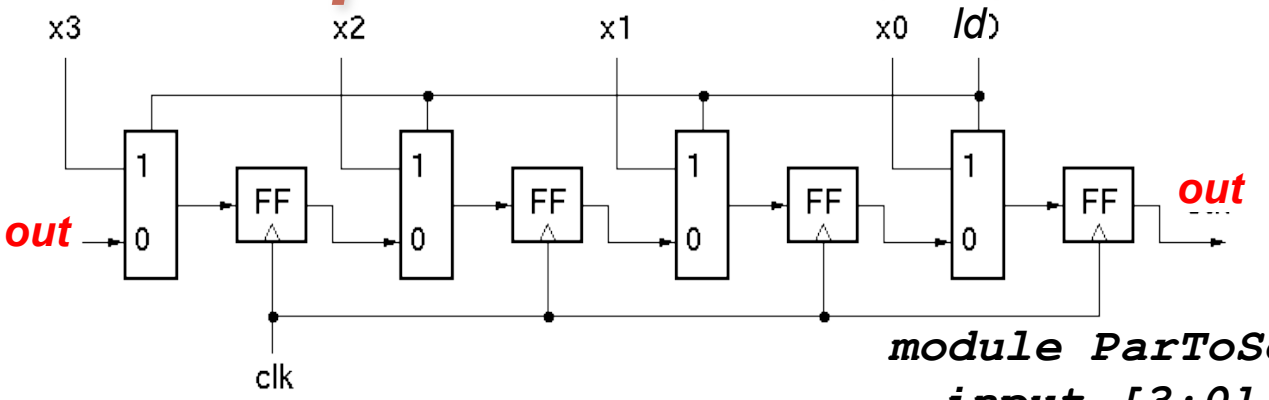
- *Blocking assignments **do not** reflect the intrinsic behavior of multi-stage sequential logic*
- *Guideline: use **nonblocking** assignments for sequential always blocks*

Example: A Simple Counter




```
// 4-bit counter with enable and synchronous clear
module counter(input clk,enb,clr,
               output reg [3:0] count);
  always @(posedge clk) begin
    count <= clr ? 4'b0 : (enb ? count+1 : count);
  end
endmodule
```

Example - Parallel to Serial Converter




```
module ParToSer(ld, X, out, clk);
   input [3:0] X;
   input ld, clk;
   output out;

   reg [3:0] Q;
   wire [3:0] NS;

   assign NS =
     (ld) ? X : {Q[0], Q[3:1]};

   always @ (posedge clk)
      Q <= NS;

   assign out = Q[0];
endmodule
```

Specifies the muxing with "rotation"

forces Q register (flip-flops) to be rewritten every cycle

connect output

Verilog in EECS 151/251A

- We use behavioral modeling at the bottom of the hierarchy
- Use instantiation to 1) build hierarchy and, 2) map to FPGA and ASIC resources not supported by synthesis.
- Favor continuous assign and avoid always blocks unless:
  - no other alternative: ex: state elements, case
  - helps readability and clarity of code: ex: large nested if else
- Use named ports.
- Verilog is a big language. This is only an introduction.
  - Complete IEEE Verilog-Standard document (1364-2005) linked to class website.
  - Harris & Harris book chapter 4 is a good source.
  - ***Be careful of what you read on the web***. Many bad examples out there.
  - We will be introducing more useful constructs throughout the semester. Stay tuned!

Final thoughts on Verilog Examples

Verilog looks like C, but it describes hardware:

Entirely different semantics: multiple physical elements with parallel activities and temporal relationships.

A large part of digital design is knowing how to write Verilog that gets you the desired circuit. First understand the circuit you want then figure out how to code it in Verilog. If you try to write Verilog without a clear idea of the desired circuit, you will struggle.

As you get more practice, you will know how to best write Verilog for a desired result.

Be suspicious of the synthesis tools! Check the output of the tools to make sure you get what you want.