EECS 151/251A
Spring 2021
Digital Design and Integrated Circuits

Instructor: Wawrzynek

Lecture 4: Verilog 2 - Sequencial Circuits

Announcements

- **We will be using remote proctoring for at least one of the two exams this semester.**
- Course drop deadline is January 29 (tomorrow).
- Virtual Front Row, 1/28:
  - Praveen Batra
  - Robin Chu
  - Kaitlyn Chan
  - Victor Ho
  - William Hsu
- *Don't forget to ask questions or offer comments!*

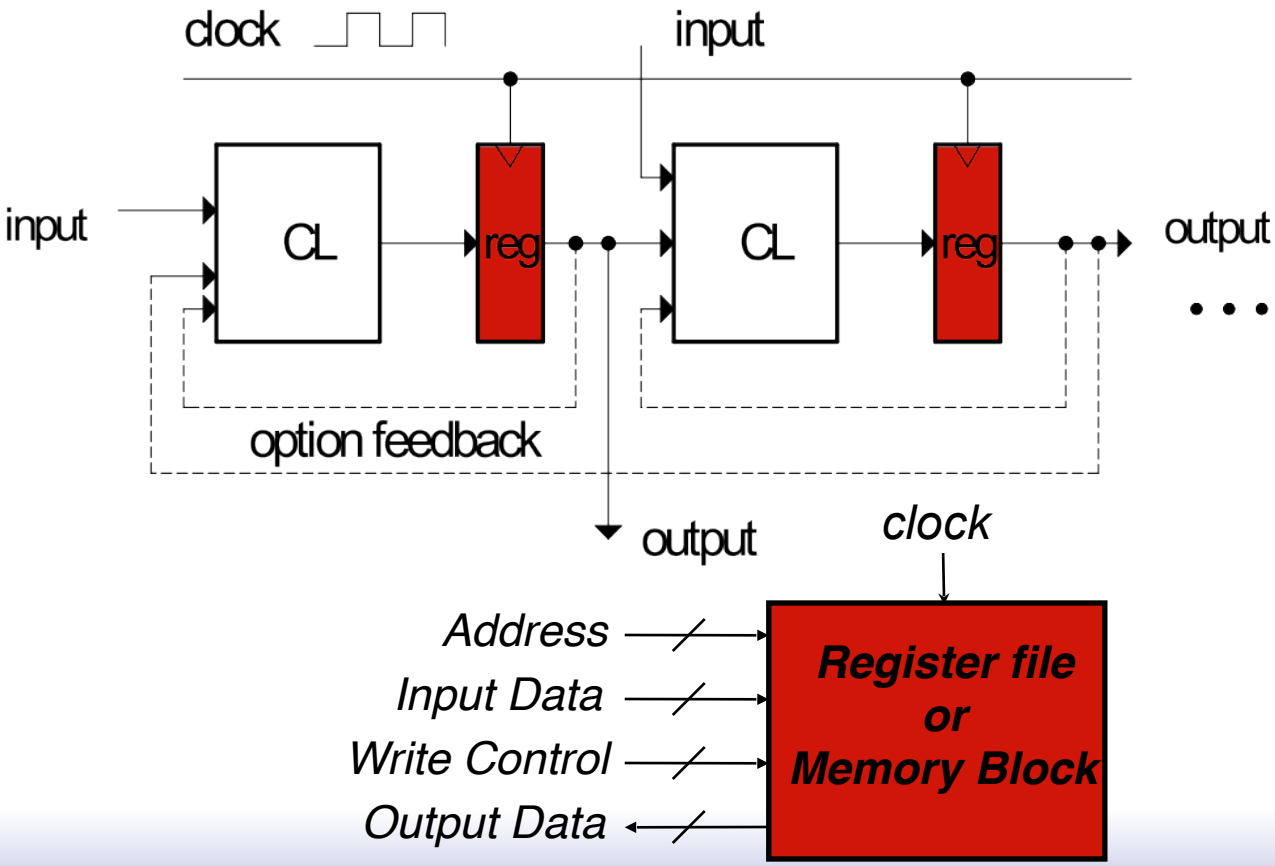
Verilog – So Far

- Combinational Logic Specification
- Two types of description:
  - Structural: design as a composition of components (also called a netlist)
    - Maps directly into hardware
  - Behavioral: design as a set of equations or language constructs
    - Requires “compiler” (synthesis tool) to generate hardware
- With parameters we define “generators”

Sequential Elements

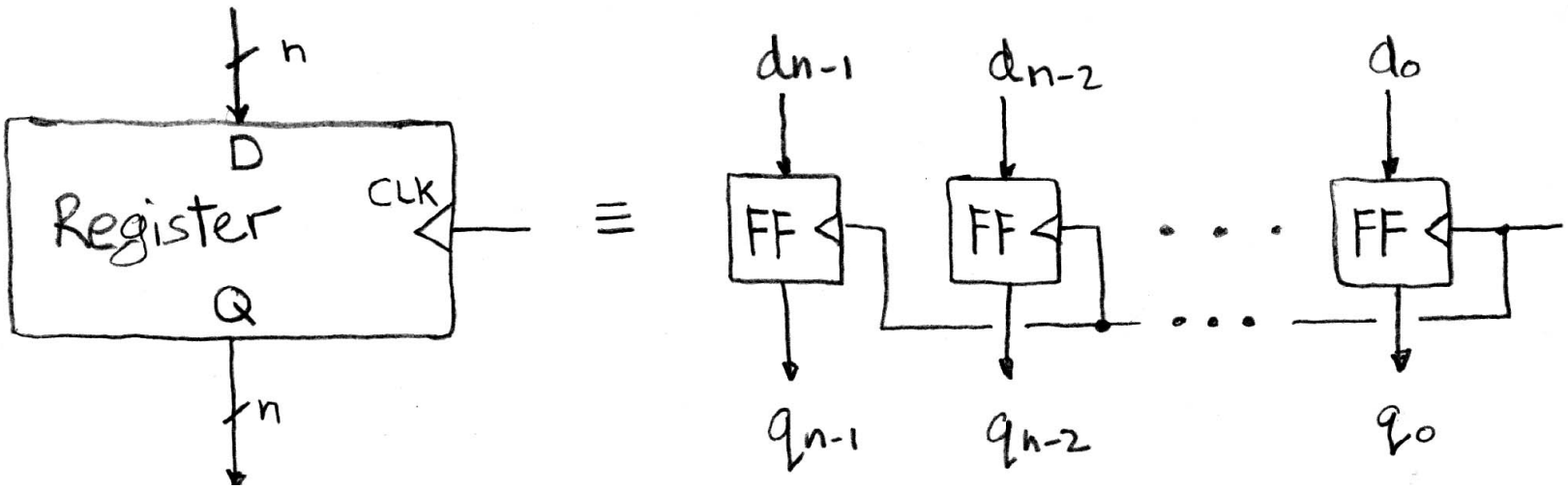
Only Two Types of Circuits Exist

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



- 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.

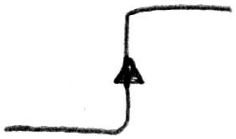
Register Details…What's inside?



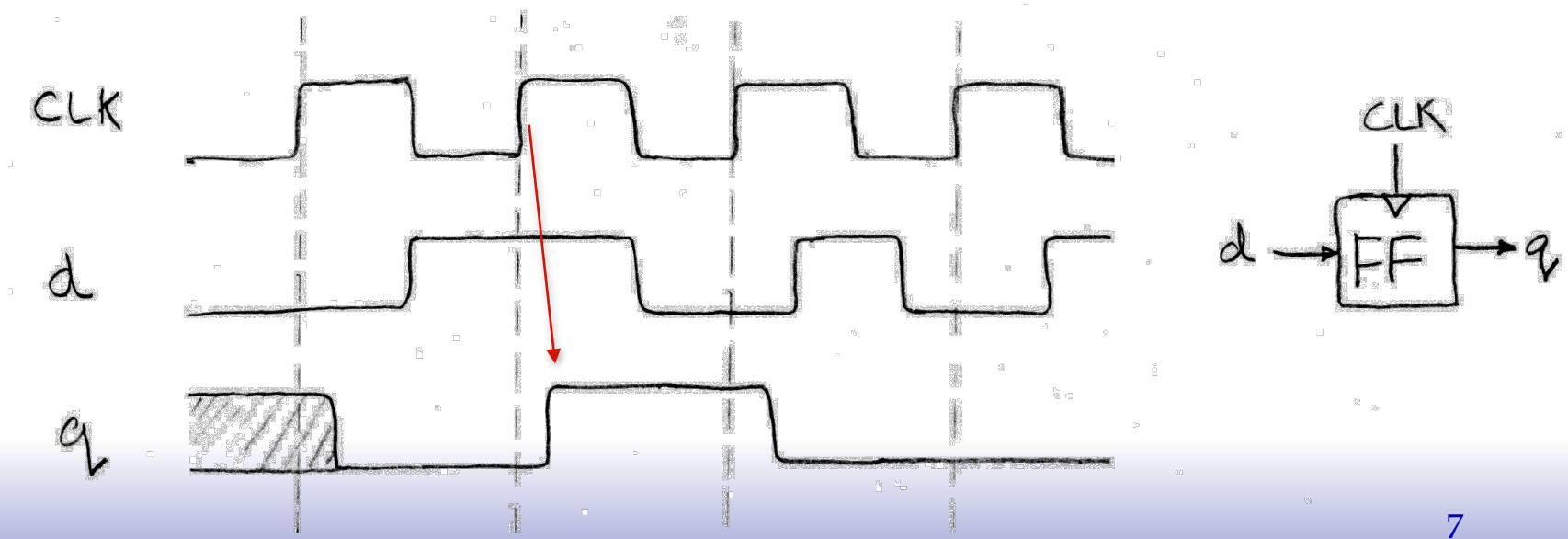
- n instances of a "Flip-Flop"
- Flip-flop name because the output flips and flops between 0 and 1 (the two stable states)
- D is "data" input, Q is "output"
- Commonly used flip-flops are really "d-type edge-triggered flip-flop"

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:

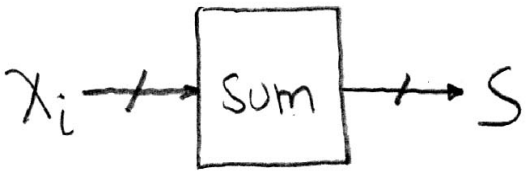


Uses for State Elements

1) As a place to store values for some indeterminate amount of time:
   - Register files (like x1-x31 on the RISC-V)
   - Memory (caches, and main memory)
2) Help control the flow of information between combinational logic blocks.
   - State elements are used to hold up the movement of information at the inputs to combinational logic blocks and allow for orderly passage
   - Used to break long paths of logic to shorten clock period (improve performance)

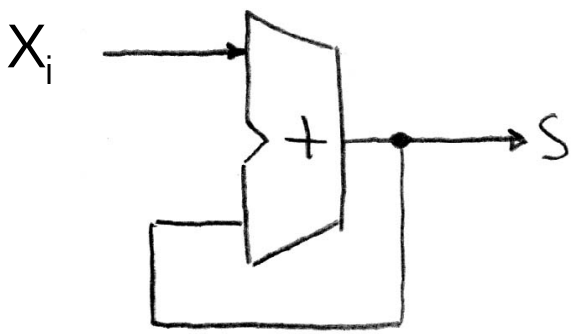
Accumulator Example

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.



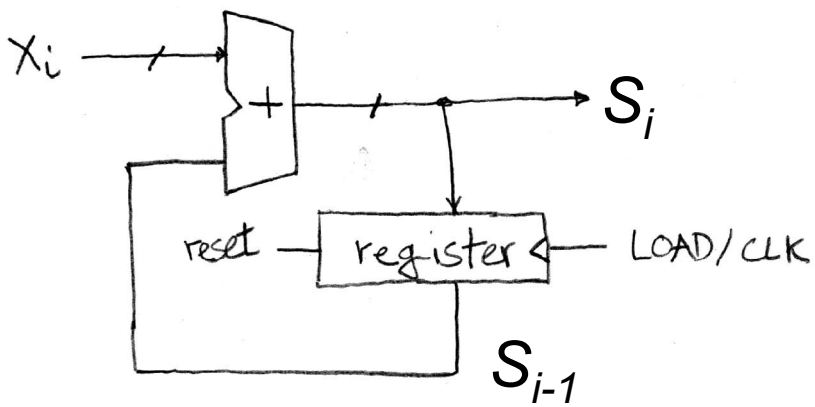
S=0; Repeat N times
S = S + X;

- We need something like this:



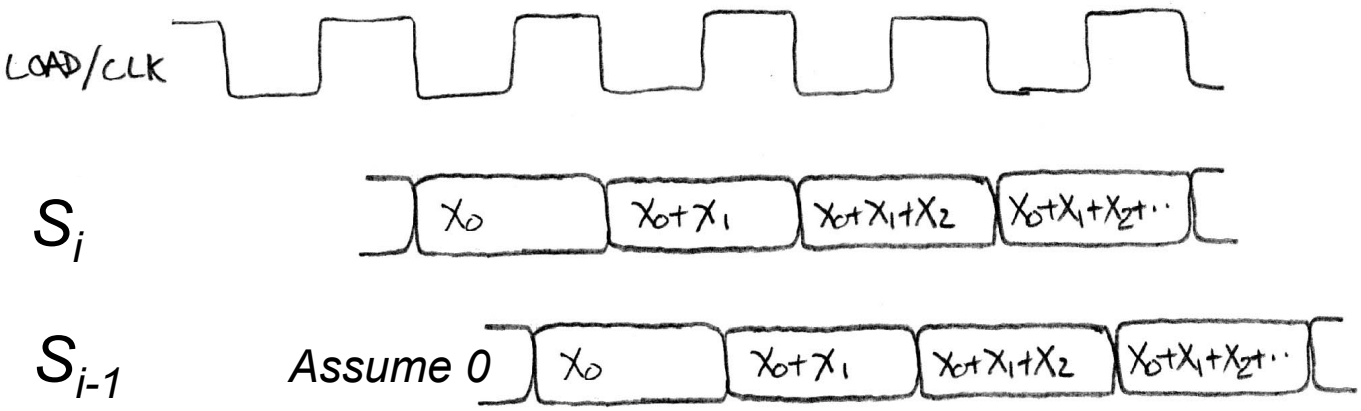
- *But not quite.*
- *Need to use the clock signal to hold up the feedback to match up with the input signal.*

Accumulator

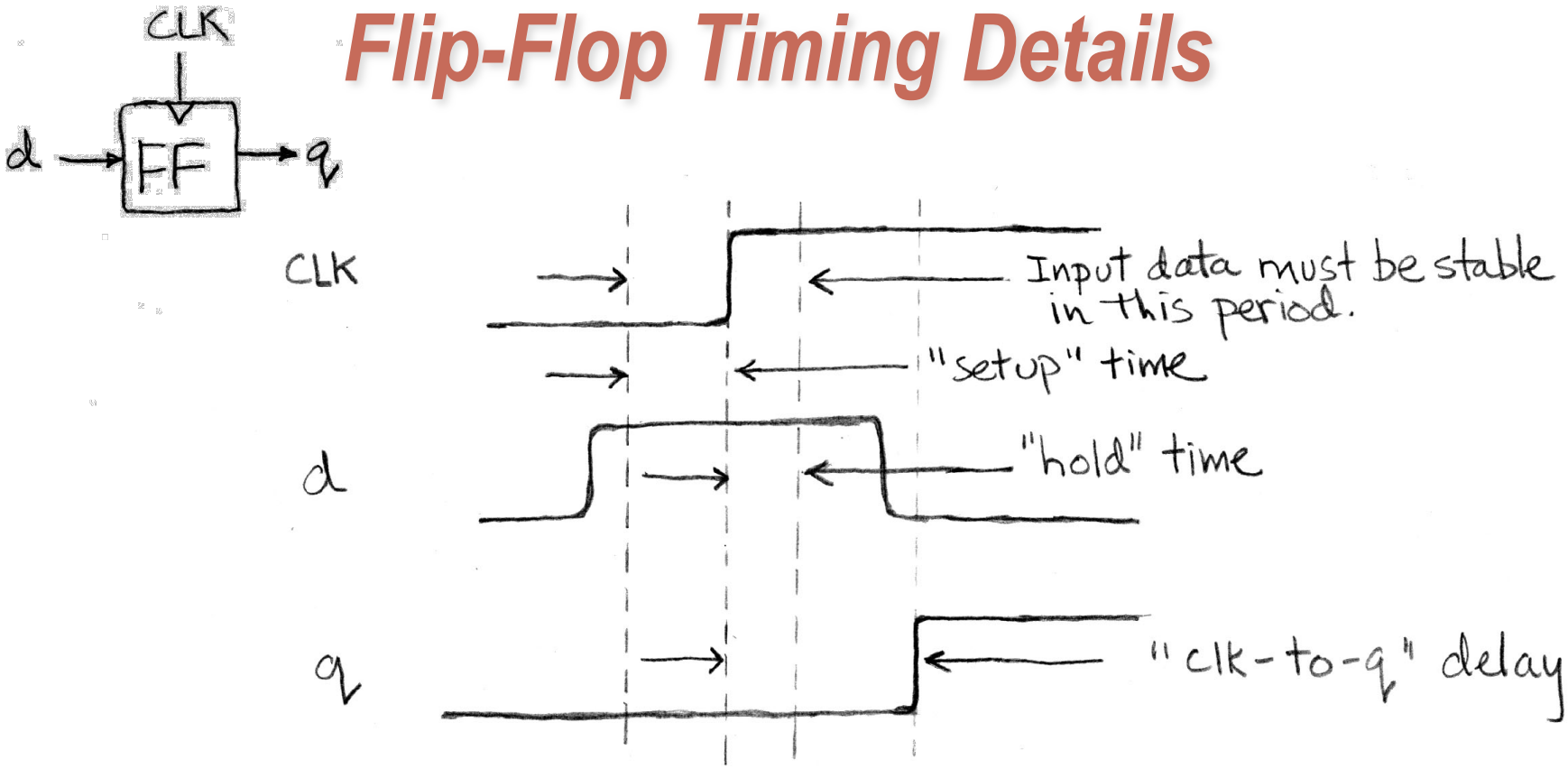


- Put register in feedback path.
- On each clock cycle the register prevents the new value from reaching the input to the adder prematurely. (The new value just waits at the input of the register)

Timing:

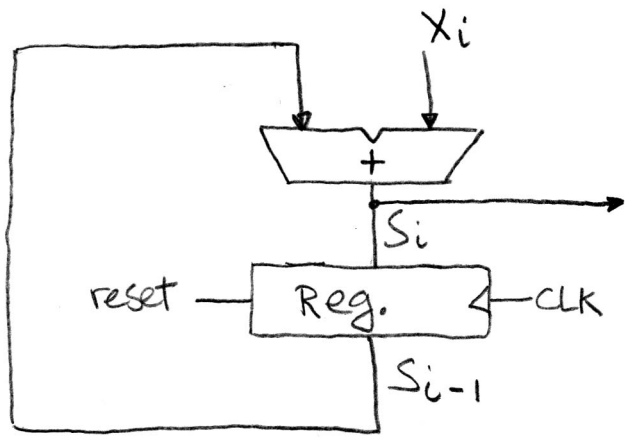


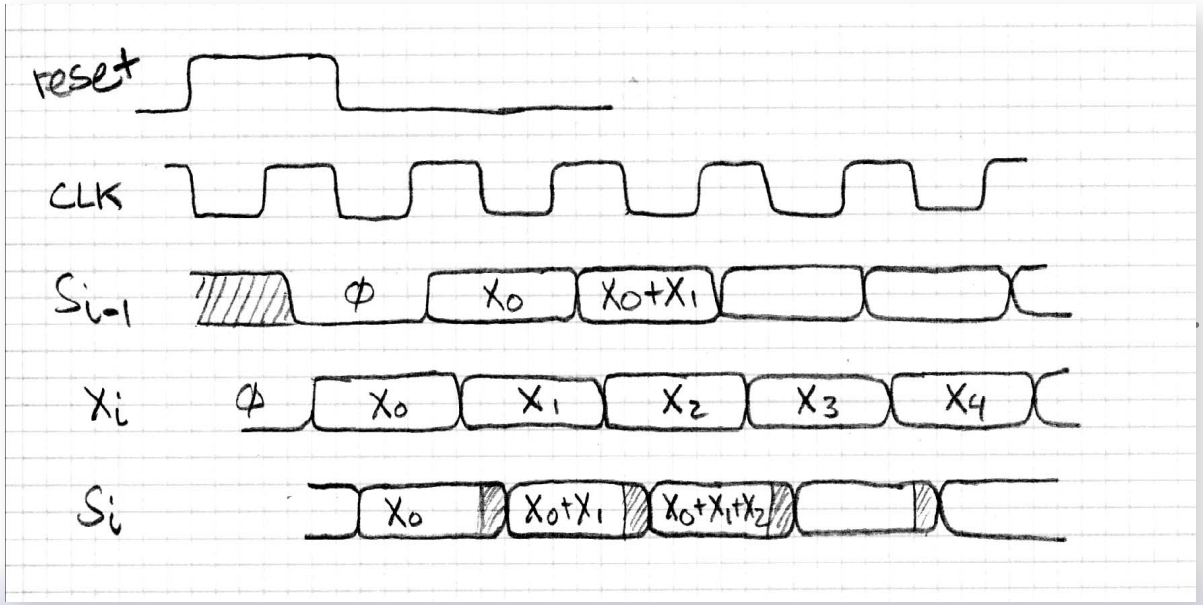
Flip-Flop Timing Details



- 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



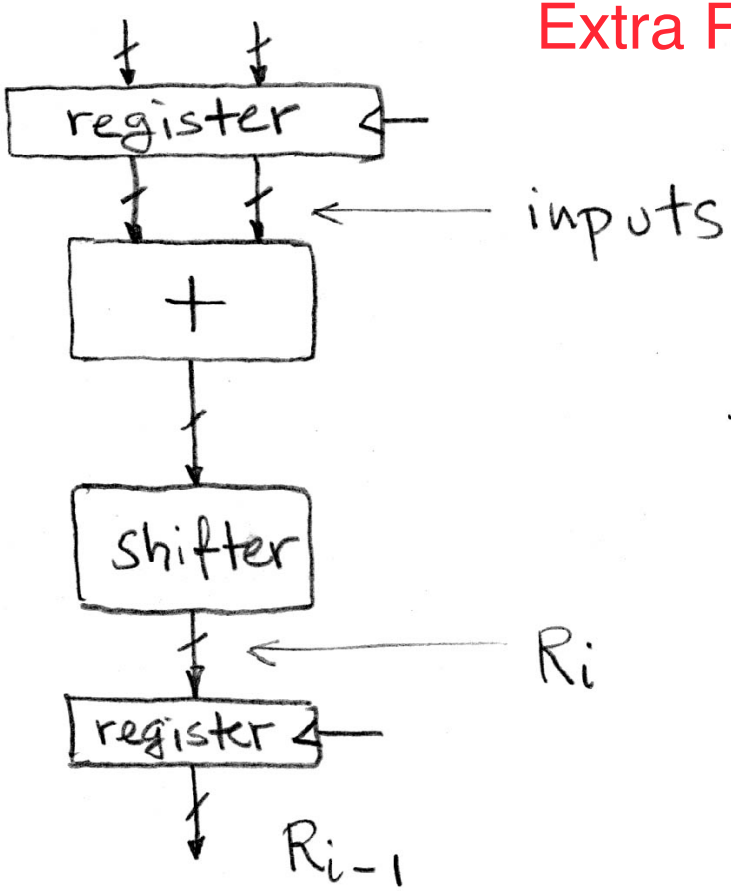


- Note:
  - Reset signal (synchronous)
  - Timing of X signal is not known without investigating the circuit that supplies X. Here we assume it comes just after S_{i-1}.

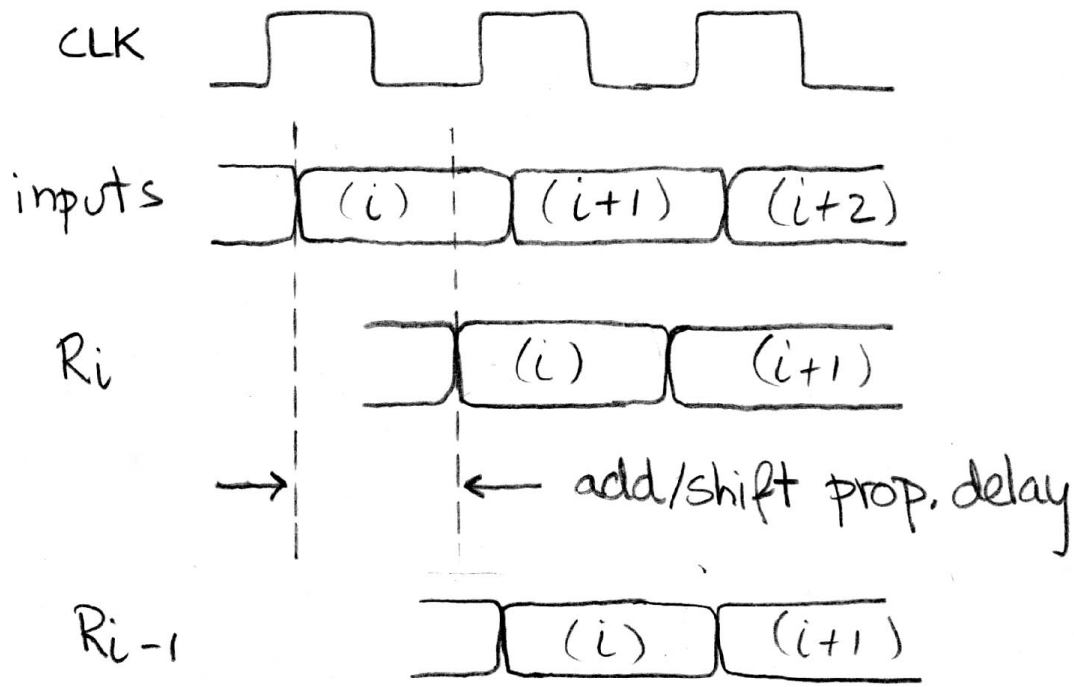
Observe transient behavior of S_i.

Pipelining to improve performance (1/2)

Extra Register are often added to help speed up the clock rate.



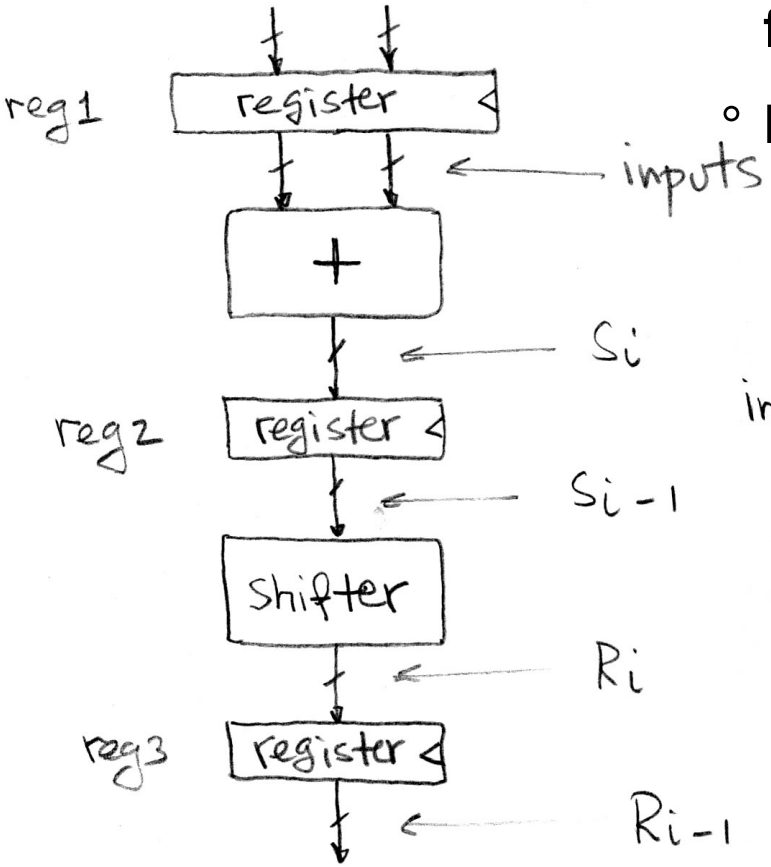
Timing...



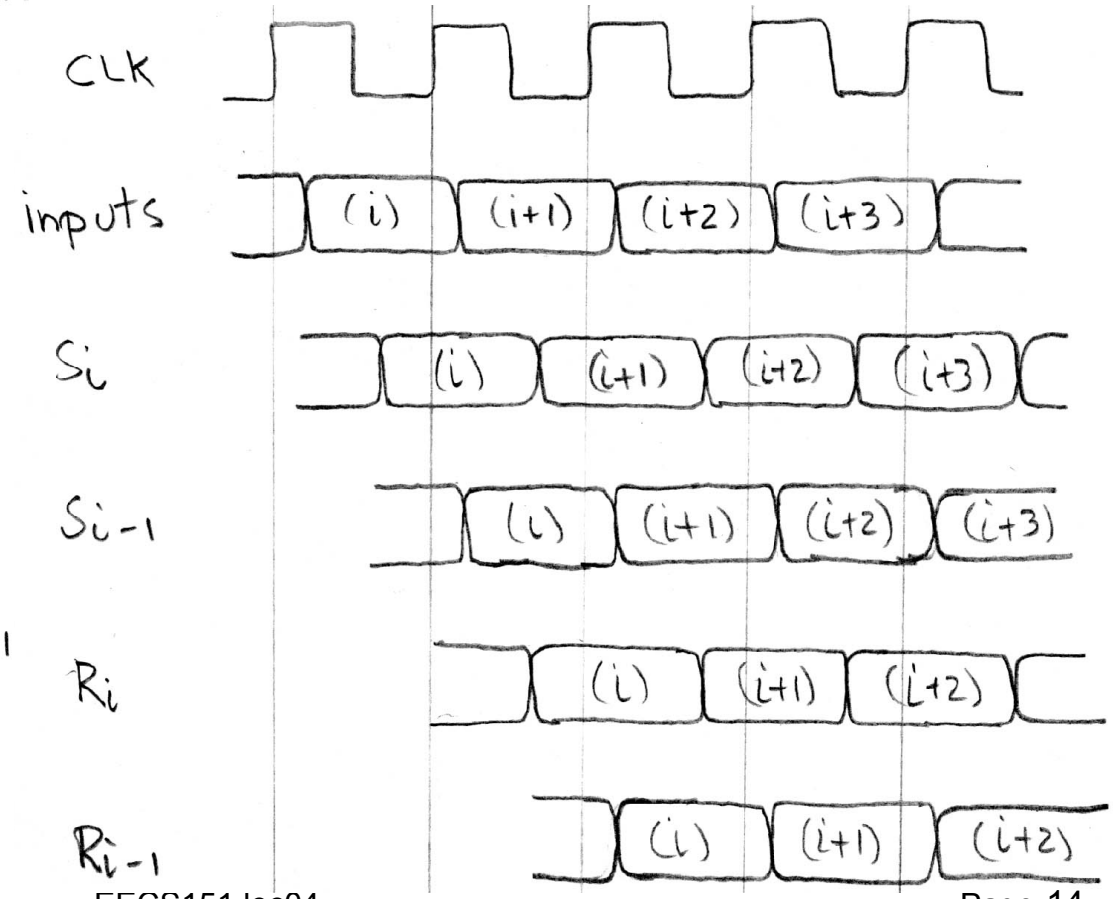
Note: delay of 1 clock cycle from input to output.
Clock period limited by propagation delay of adder/shifter.

Pipelining to improve performance (2/2)

- Insertion of register allows higher clock frequency.
- More outputs per second.



Timing...



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 reset example:

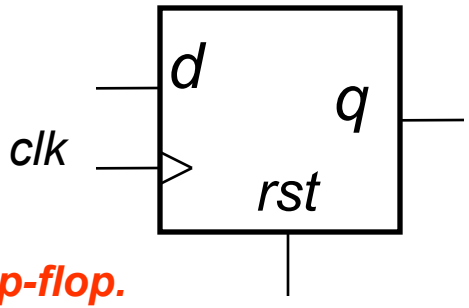
```
module dff(q, d, clk, rst);
  input d, clk, rst;
  output reg q;

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

keyword

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

This gives priority to reset over d.

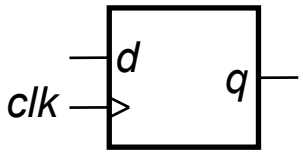


On FPGAs, maps to native flip-flop.

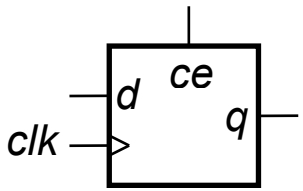
Unlike logic gates, their are no primitive flip-flops in Verilog. Although, <u>we have defined a set for your use.</u>

EECS151 Registers

- All registers are "N" bits wide - the value of N is specified at instantiation
- All positive edge triggered.

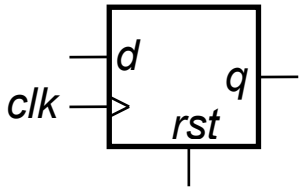



```
module REGISTER(q, d, clk);
    parameter N = 1;
```



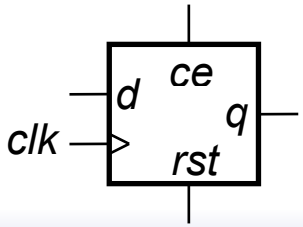

```
module REGISTER_CE(q, d, ce, clk);
    parameter N = 1;
```

On the rising clock edge if clock enable (ce) is 0 then the register is disabled (it's state will not be changed).




```
module REGISTER_R(q, d, rst, clk);
    parameter N = 1;
    parameter INIT = 1b'0;
```

On the rising clock edge if reset (rst) is 1 then the state is set to the value of INIT. Default INIT value is all 0's.




```
module REGISTER_R_CE(q, d, rst, ce, clk);
    parameter N = 1;
    parameter INIT = 1b'0;
```

Reset (rst) has priority over clock enable (ce).

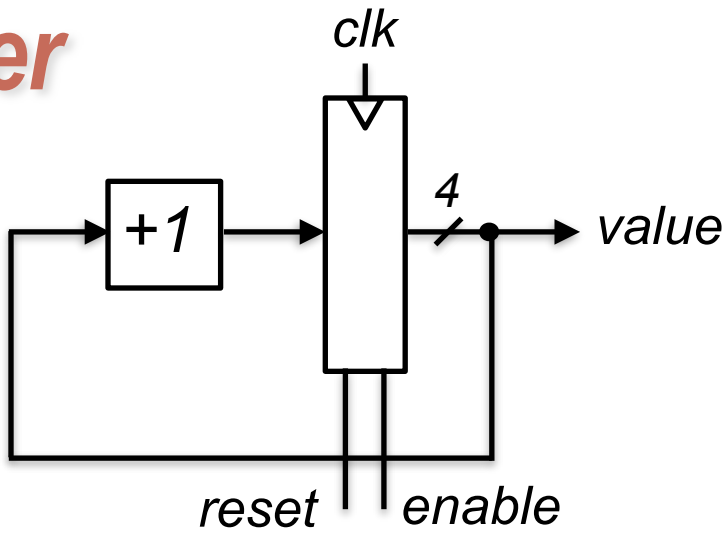
EECS151 "no register inference policy"

- **Instead of using flip-flop and register *inference*, all EECS151/251A Verilog specifications must use explicit instantiation of register modules from the "EECS151 library".**
- Policy applies to lecture, discussion, lab, project, and problem sets.
- For simulation and synthesis use: **`` `include "EECS151.v" ``**

Using predefined register modules allows you to simplify the specification of sequential circuits, and enforces separation of combination logic and state elements (a good design style).

4-bit wrap-around counter

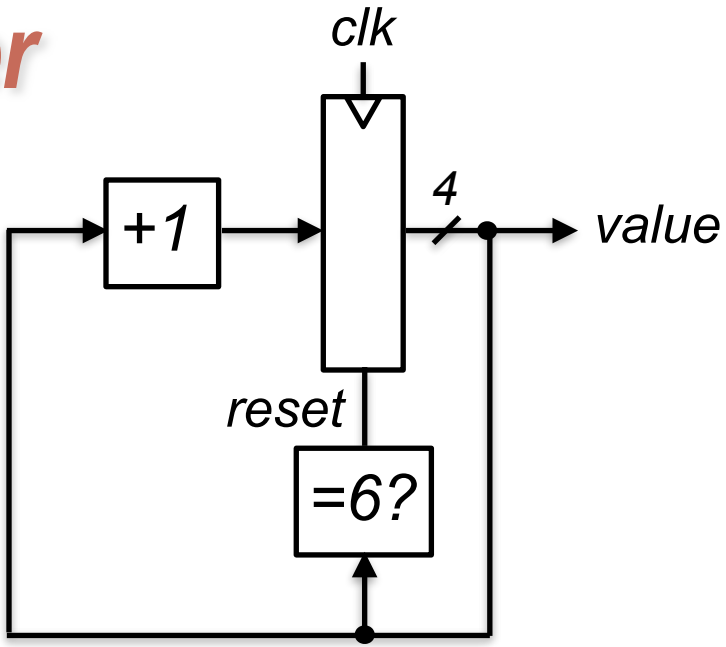
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0, 1, …




```
module counter(value, enable, reset, clk);
   output [3:0] value;
   input        enable, reset, clk;
   wire [3:0]   next;
   REGISTER_R #(4) state (.q(value), .d(next), .rst(reset), .
   assign next = value + 1;
endmodule // counter
```

4-bit "count to 6" counter

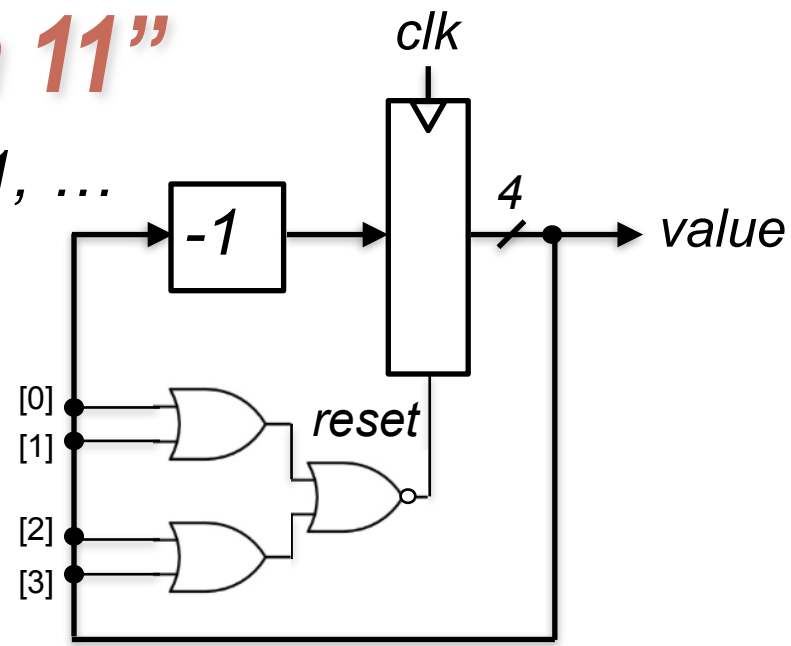
0, 1, 2, 3, 4, 5, 6, 0, 1, …




```
module count_to(value, clk);
   output [3:0] value;
   input        clk;
   wire [3:0]   next;
   wire         reset;
   REGISTER_R #(4) state (.q(value), .d(next), .rst(reset), .clk(clk));
   assign next = value + 1;
   assign reset = (value == 4'h6) ? 1'b1 : 1'b0;
endmodule // count_to
```

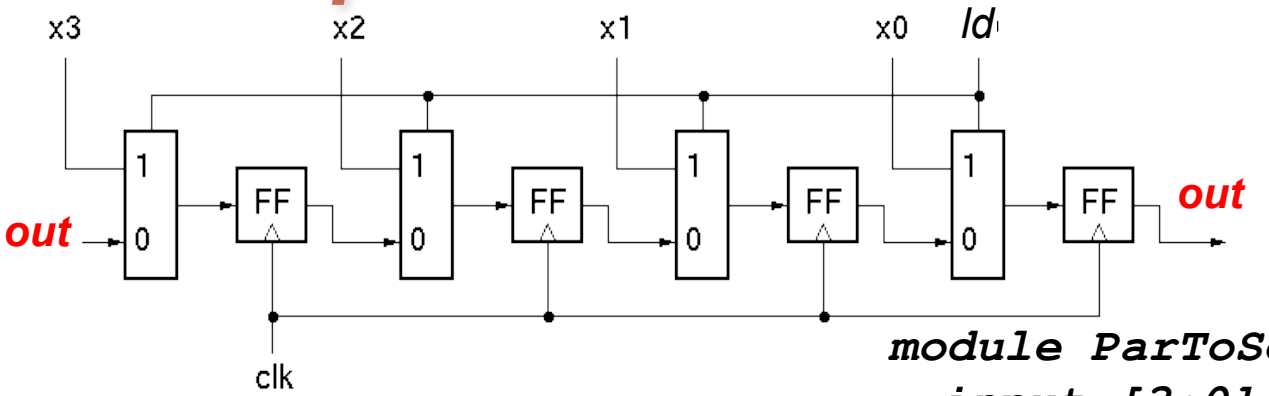
4-bit "count down from 11"

11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 11, …




```
module count_down_from(value, clk);
   output [3:0] value;
   input        clk;
   wire [3:0]   next;
   wire         reset;
   REGISTER_R #(.N(4), .INIT(4'd11))
        state (.q(value), .d(next), .rst(reset), .clk(clk));
   assign next = value - 1;
   assign reset = ~(|value);
endmodule // count_down_from
```

Example - Parallel to Serial Converter




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

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

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

   REGISTER state #(4)
      (.q(Q), .d(NS), .clk(clk));

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

Specifies the muxing with "rotation"

Instantiates a 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: case statement
  - helps readability and clarity of code: ex: large nested if else
- Obey the no register inference policy.
- 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.