Intro to Digital Design
L4: Combinational Building Blocks & Sequential Logic

Instructor: Naomi Alterman

Teaching Assistants:

Derek de Leuw
Isabel Froelich
Kevin Hernandez
Aarjav Jain
Packard Stephenson

Administrivia

- Lab 4 out today
  - Playing with the 7-segment displays

- Quiz 1 is next week in lecture
  - Last 20 minutes, worth 10% of your course grade
  - On Lectures 1-3: CL, K-maps, Waveforms, and Verilog
  - Past Quiz 1 (+ solutions) on website: Course Info → Quizzes
  - **NB**: we'll probably put a **block diagram** question on this one similar to what you've been drawing in your lab reports

Lecture Outline

- **Arithmetic (adders)**
- Sequential Logic in theory
- Sequential Logic in Verilog

Review: Unsigned Binary Integers

- n bits can represent integers 0 to 2^n-1

- When written in **binary**/"base 2“, each digit represents a bit (eg, a signal on a wire)
- How I convert to base 2 in my head:

Binary digits: … 1 1 0 0 1

$\times 16 + \times 8 + \times 4 + \times 2 + \times 1 = 25$

(2^4) (2^3) (2^2) (2^1) (2^0)

Review: Binary Arithmetic

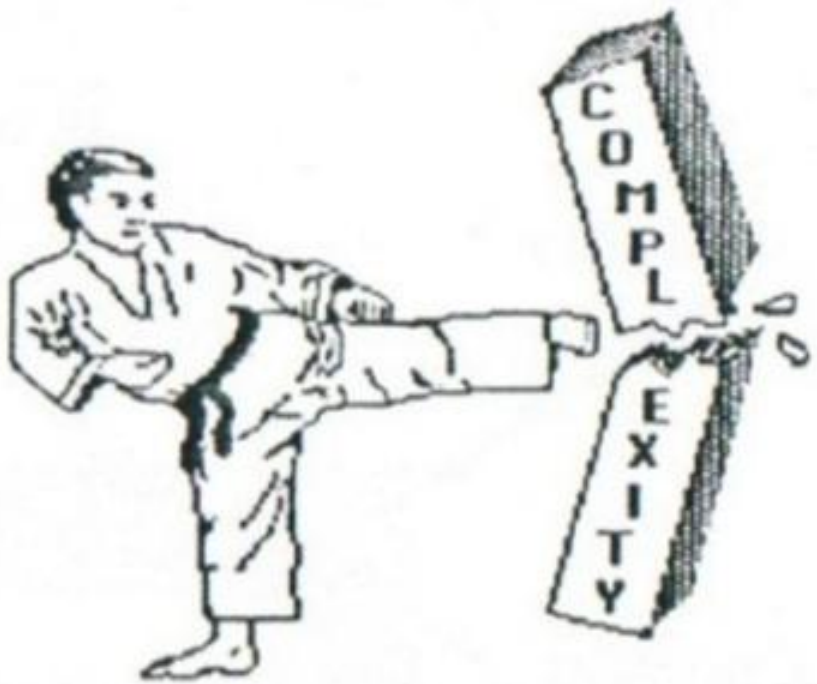
- Add and subtract using the "carry" and "borrow" rules, just in binary

#1

```
  63      0 0 1 1 1 1 1 1
+  8    + 0 0 0 0 1 0 0 0
  71      0 1 0 0 0 1 1 1
```

#2

```
  65      0 1 0 0 0 0 0 1
-  8    - 0 0 0 0 1 0 0 0
  57      0 0 1 1 1 0 0 1
```



http://hartenstein.de/KARL/KARL-folder.pdf

Half Adder (1 bit)

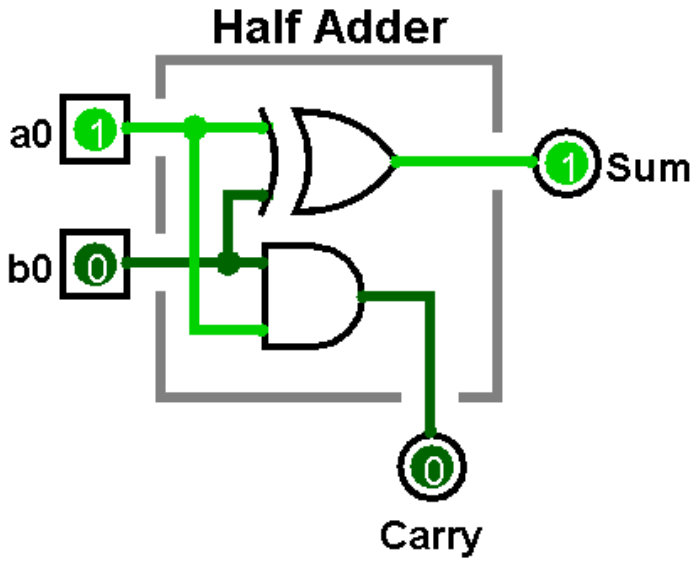
$$\begin{array}{cccc|c} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline & s_3 & s_2 & s_1 & s_0 \end{array}$$

Carry-out bit

a_0	b_0	c_1	s_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$\text{Carry} = a_0 b_0$$

$$\text{Sum} = a_0 \oplus b_0$$



Full Adder (1 bit)

Possible carry-in c_1

$$\begin{array}{cccc|c} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline & s_3 & s_2 & s_1 & s_0 \end{array}$$

Carry-in → c_i | Carry-out → c_{i+1}

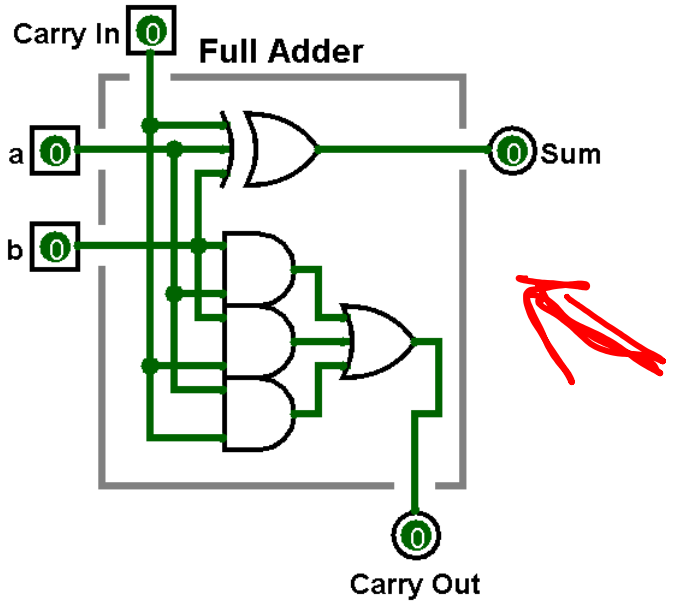
c_i	a_i	b_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$s_i = \text{XOR}(a_i, b_i, c_i)$$

$$c_{i+1} = \text{MAJ}(a_i, b_i, c_i)$$

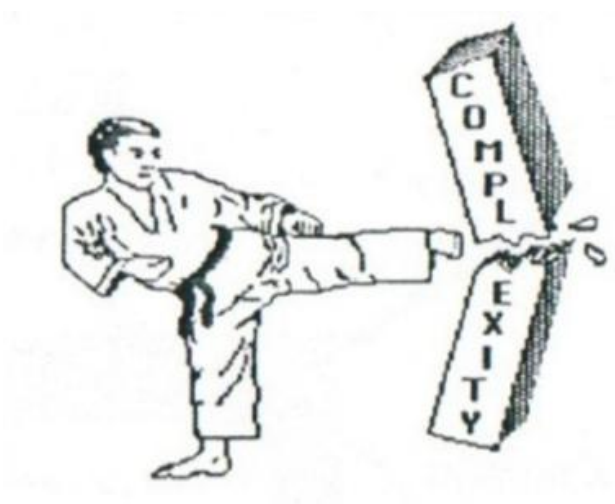
$$= a_i b_i + a_i c_i + b_i c_i$$

if majority of a, b, c are 1 output is 1

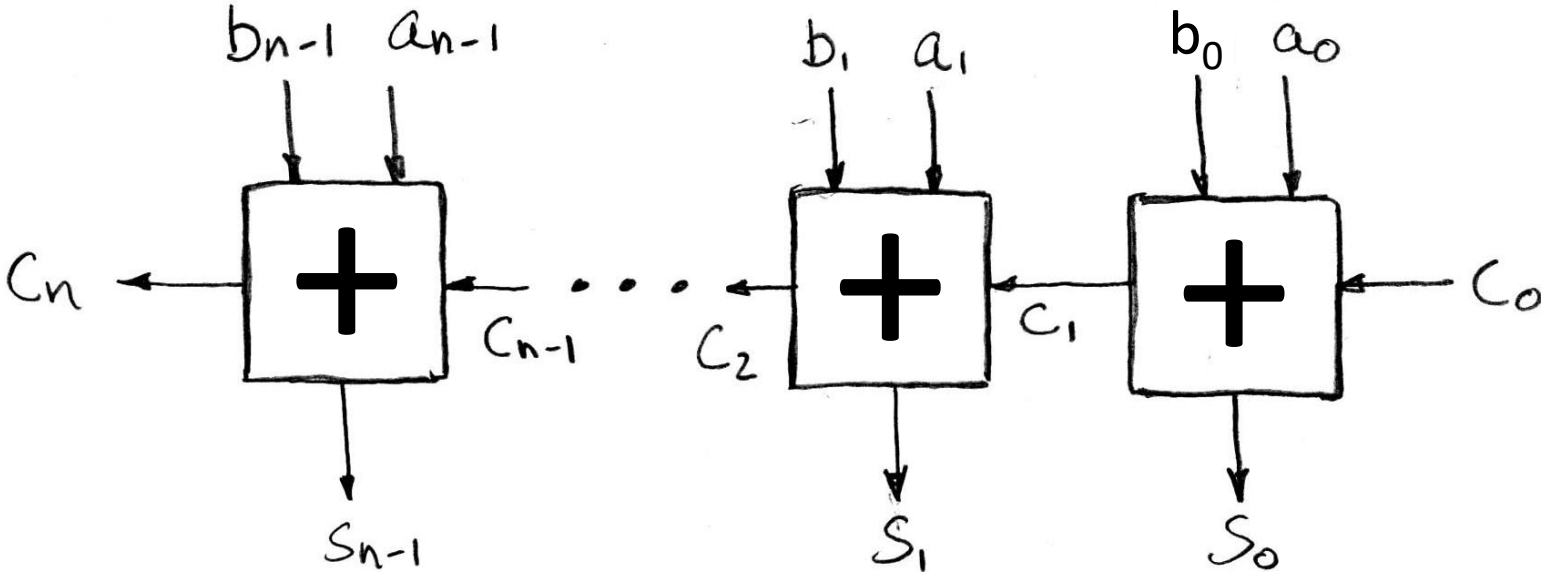


Multi-Bit Adder (N bits)

- Chain 1-bit adders by connecting $CarryOut_i$ to $CarryIn_{i+1}$:



http://hartenstein.de/KARL/KARL-folder.pdf



1-bit Adders in Verilog

- How do we get the carry bit out of a Verilog adder?

```
module sad_add (s, a, b);
  output logic s;
  input  logic a, b;

  always_comb begin
    s = a + b;
  end
endmodule
```

- Use `{sig, …, sig}` for *concatenation*

```
module half_add (c, s, a, b);
  output logic c, s;
  input  logic a, b;

  always_comb begin
    {c, s} = a + b;
  end
endmodule
```

Ripple-Carry Adder in Verilog

```
module fulladd (cout, s, cin, a, b);
  output logic cout, s;
  input  logic cin, a, b;

  always_comb begin
    {cout, s} = cin + a + b;
  end
endmodule
```

- Chain full adders?

```
module add2 (cout, s, cin, a, b);
  output logic cout; output logic [1:0] s;
  input  logic cin;  input  logic [1:0] a, b;
  logic  c1;

  fulladd b1 (cout, s[1], c1,  a[1], b[1]);
  fulladd b0 (c1,   s[0], cin, a[0], b[0]);
endmodule
```

instantiating N copies of our adder circ.

Feeling negative

- Ok, but how do we build logic to handle negative numbers?

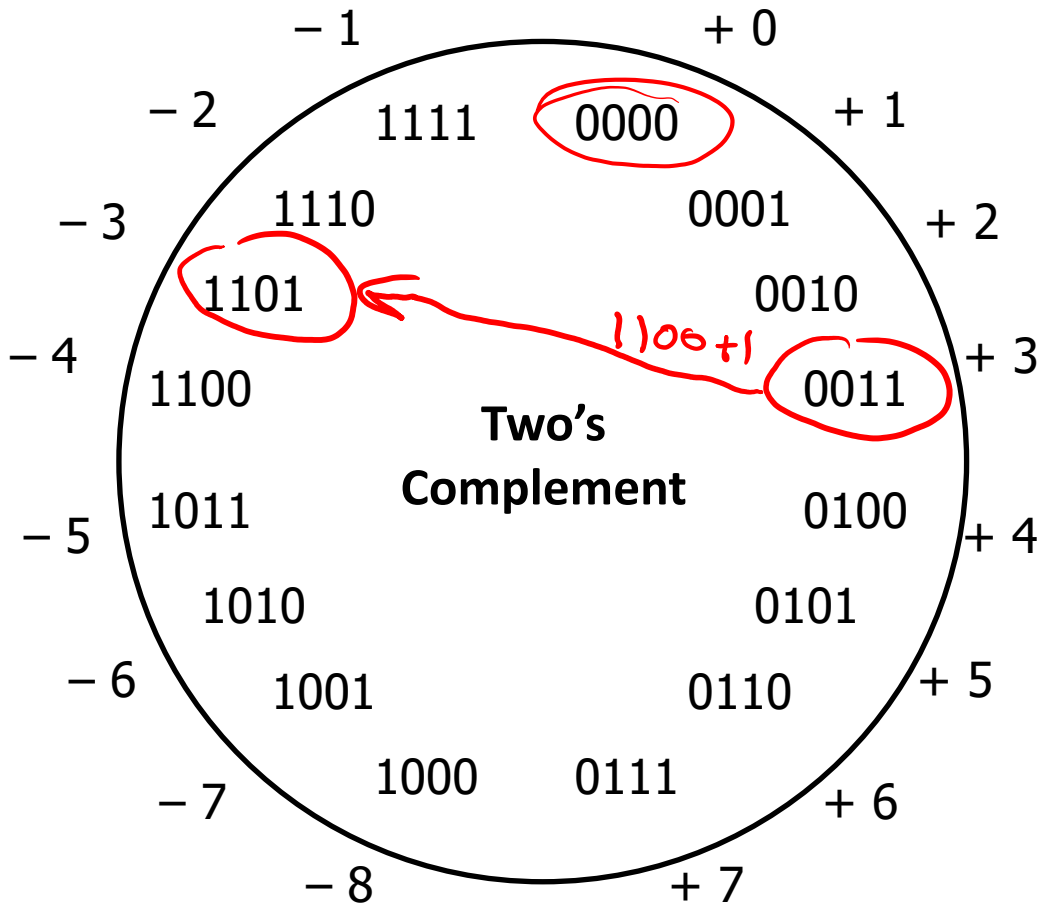
Review: Two's Complement

- MSB's weight becomes -2^{n-1}
  - Can think of MSB as "sign bit"

Review: Two's Complement (cont'd)

- **Properties:**
  - In n bits, represent integers -2^{n-1} to $2^{n-1} - 1$
  - Positive number encodings match unsigned numbers
  - Single zero (encoding = all zeros)
- **Negation procedure:**
  - Take the bitwise complement and then add one

```
( ~x + 1 == -x )
```



Subtraction in binary

- Negate the 2nd operand and then add like usual:
  $A - B = A + (-B) = A + (\sim B + 1)$
  negate B
- **Note**: we "throw away" the last carry-out
- 4-bit examples:

#3

	s4	U4
0 0 1 0	+2	2
+ 1 1 0 0	−4	12
1 1 1 0	−2	

#4

	s4	u4
1 0 0 0	−8	8
+ 0 1 0 0	+4	4
1 1 0 0	−4	

#5

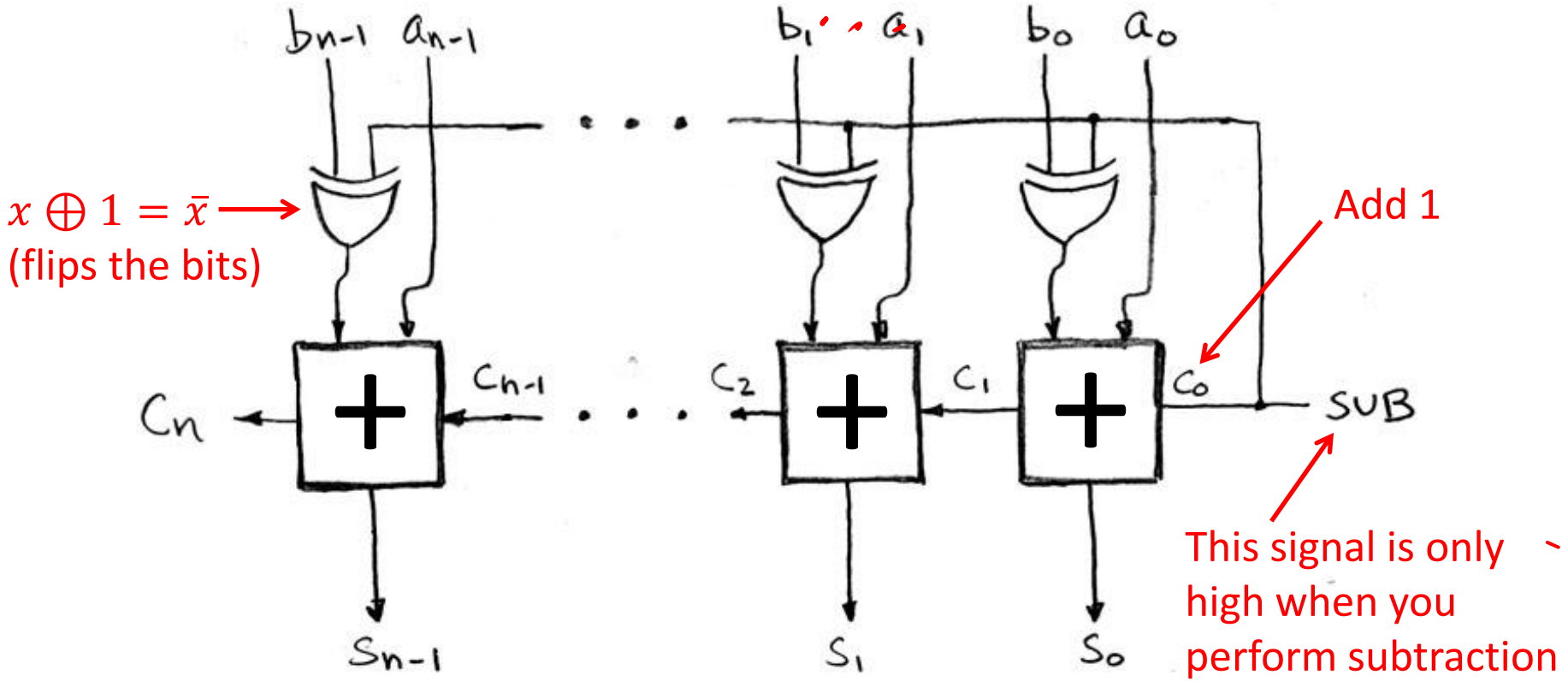
0 1 1 0	+6	6
− 0 0 1 0	+2	2
+ 1 1 1 0		
0 1 0 0	4	

#6

1 1 1 1	−1	15
− 1 1 1 0	−2	14

Subtraction in circuitry

- How do we modify our circuit to subtract numbers too?
  - Flip the bits and add 1!



Detecting Arithmetic Overflow

- **Overflow:** When a calculation produces a result that can't be represented in the current encoding scheme
  - Integer range limited by fixed width
  - Can occur in both the positive and negative directions

- **Unsigned Overflow**
  - Result of add/sub is > U_{max} or < U_{min}

- **Signed Overflow**
  - Result of add/sub is > T_{max} or < T_{min}
  - (+) + (+) = (−) or (−) + (−) = (+)

Signed Overflow Examples

#7

	s4	u4
0 1 0 1	+5	5
+ 0 0 1 1	+3	3
1 0 0 0	-8	8

+ + → −

#8

	s4	
1 0 0 1	-7	
+ 1 1 1 0	-2	
0 1 1 1		−7

(4)(2)(1)

+ + → −

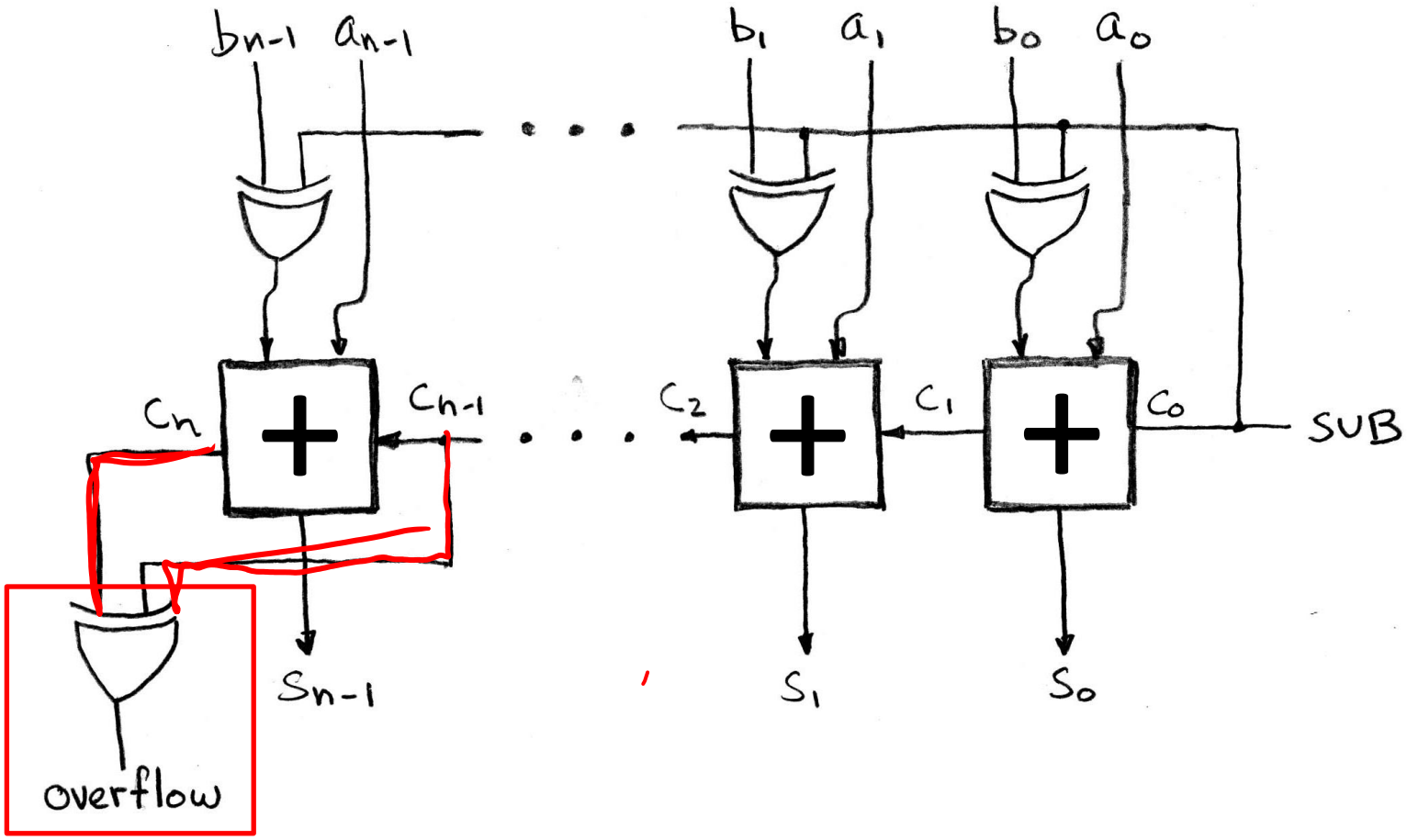
#9

	s4
0 1 0 1	+5
+ 0 0 1 0	+2

#10

	s4
1 1 0 0	-4
+ 0 1 0 0	4

Two's Complement Adder/Subtractor with Overflow



Add/Sub in Verilog (parameterized)

#define
templates < >

- Variable-width add/sub (with overflow, carry)

```
module addN #(parameter N=32) (overflow, carry_out, out, subtract, A, B);
  output logic           overflow, carry_out;
  output logic [N-1:0] out;
  input  logic           subtract;
  input  logic [N-1:0] A, B;

endmodule  // addN
```

carry-in

add up to the last col of bits

Add/Sub in Verilog (parameterized)

- Variable-width add/sub (with overflow, carry)

```
module addN #(parameter N=32) (overflow, carry_out, out, subtract, A, B);
  output logic          overflow, carry_out;
  output logic [N-1:0] out;
  input  logic          subtract;
  input  logic [N-1:0] A, B;
  logic  [N-1:0]        B_prime;    // possibly bit-flipped B
  logic                 carry_in;   // eg, second-to-last carry-out

  always_comb begin
    B_prime = B ^ {N{subtract}};  // replication operator
    {C2, out[N-2:0]} = A[N-2:0] + B_prime[N-2:0] + subtract;
    {carry_out, out[N-1]} = A[N-1] + B_prime[N-1] + carry_in;
    overflow = carry_out ^ carry_in;
  end
endmodule  // addN
```

- "overflow" is the x86-64 CPU's **OF** flag, "carry_out" is **CF** flag

Add/Sub in Verilog (parameterized)

```
module addN_tb ();
  logic          subtract;
  logic [N-1:0] A, B;
  logic          overflow, carry_out;
  logic [N-1:0] out;

  addN #(.N(4)) dut (.overflow, .carry_out, .out, .subtract, .A, .B);
  initial begin
    #100;  subtract = 0;  A = 4'b0101;  B = 4'b0010;  //  5 +  2
    #100;  subtract = 0;  A = 4'b1101;  B = 4'b1011;  // -3 + -5
    #100;  subtract = 0;  A = 4'b0101;  B = 4'b0011;  //  5 +  3
    #100;  subtract = 0;  A = 4'b1001;  B = 4'b1110;  // -7 + -2
    #100;  subtract = 1;  A = 4'b0101;  B = 4'b1110;  //  5 -(-2)
    #100;  subtract = 1;  A = 4'b1101;  B = 4'b0101;  // -3 -  5
    #100;  subtract = 1;  A = 4'b0101;  B = 4'b1101;  //  5 -(-3)
    #100;  subtract = 1;  A = 4'b1001;  B = 4'b0010;  // -7 -  2
    #100;
  end
endmodule  // addN_tb
```

params

port list

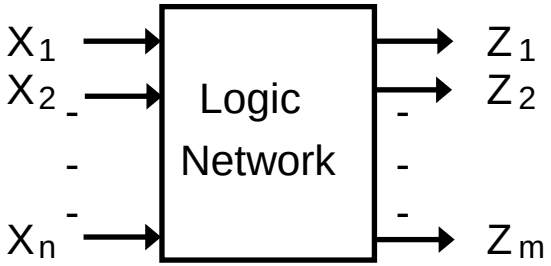
Miso Moment

Lecture Outline

- Arithmetic (adders)
- **Sequential Logic in theory**
- Sequential Logic in Verilog

Synchronous Digital Systems (SDS)

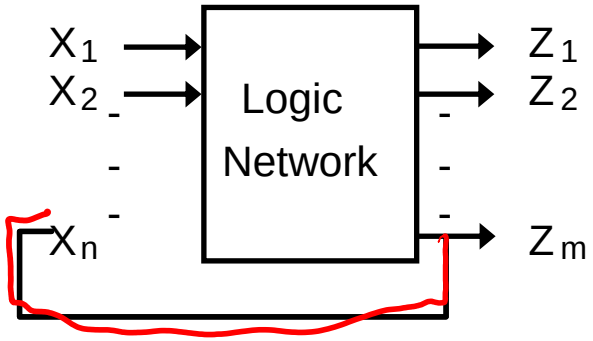
Combinational Logic (CL)



Network of logic gates without feedback.

Outputs are functions only of inputs.

Sequential Logic (SL)



The presence of feedback introduces the notion of "state."

Circuits can "remember" or store information.

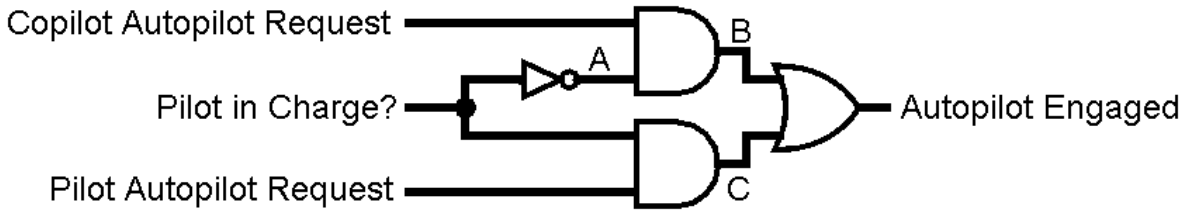
Uses for Sequential Logic

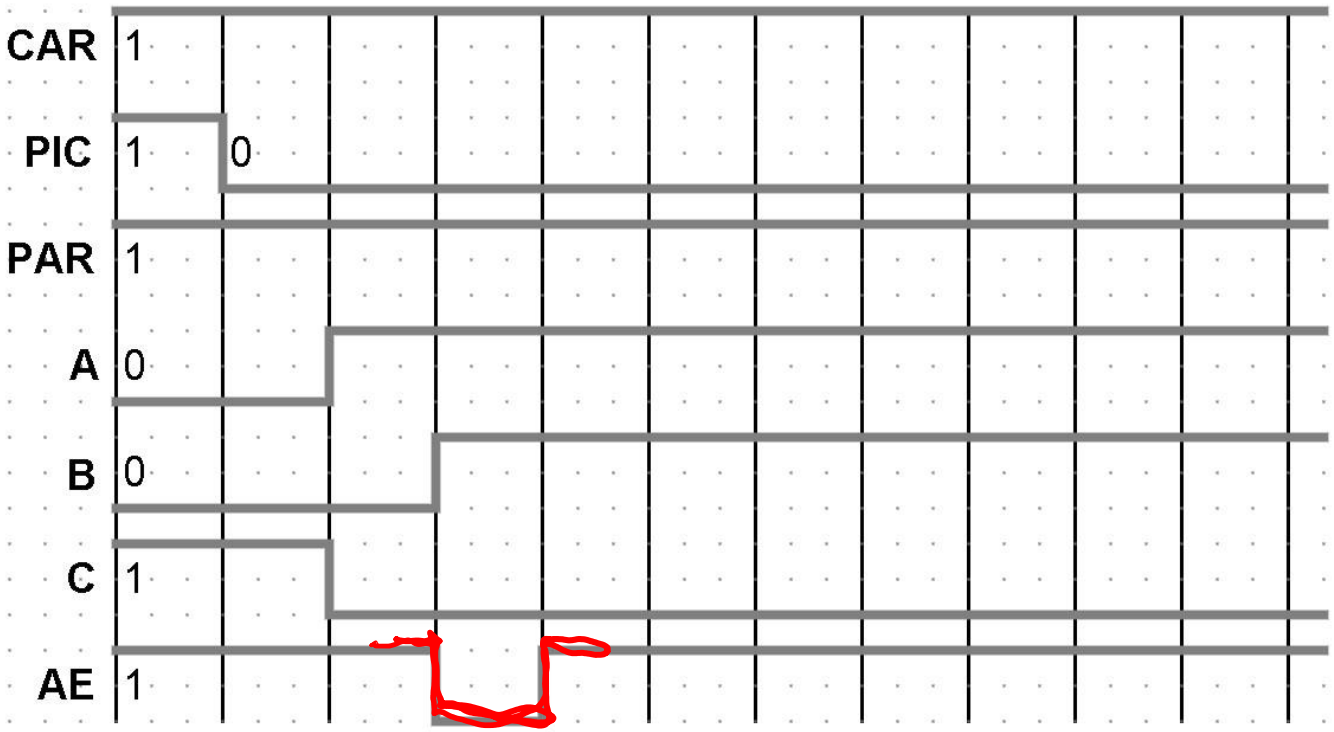
- Place to store values for some amount of time:
  - Registers
  - Memory

- *Help control flow of information between combinational logic blocks*
  - Hold up the movement of information to allow for orderly passage through CL

Control Flow of Information

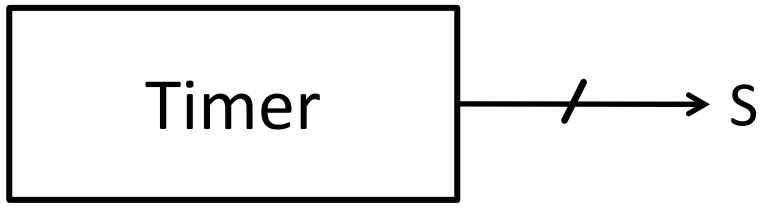
- Remember the hazards?





Design example: Perpetual Timer

- A circuit that counts up from 0 over time
  - When time is up, stops counting and beeps incessantly
  - Needs to "remember" previous value to calculate next value

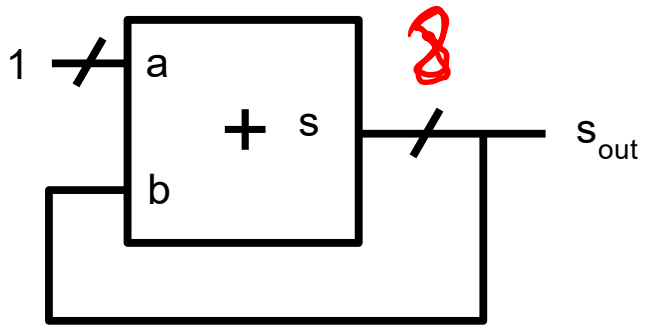


- Want:

```
s = 0;
while (true){
      s = s + 1;
}
```

Timer: First Try

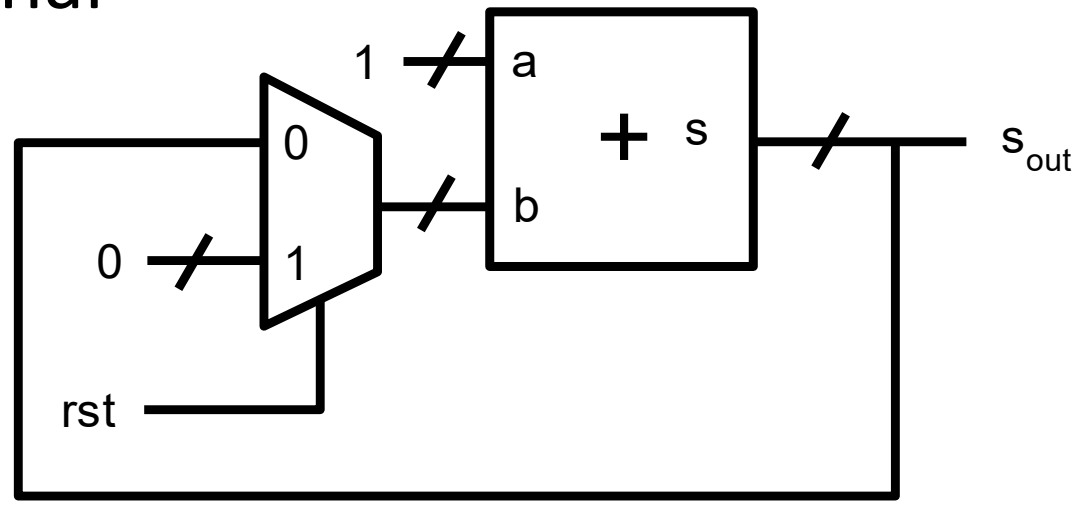
Does this work?



1) How do we say: ‘S=0’?
2) How to control the next iteration of the ‘for’ loop?

Timer: Second Try

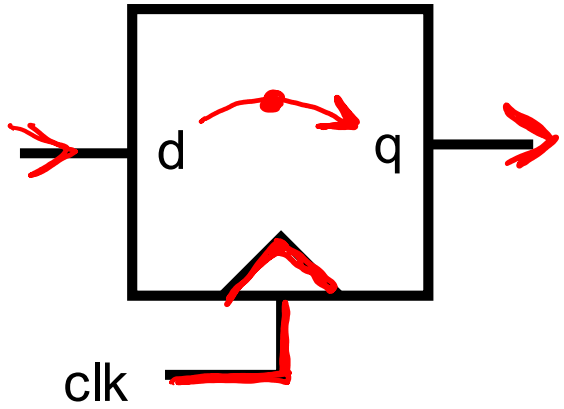
We'll add a "reset" signal
Does this work?

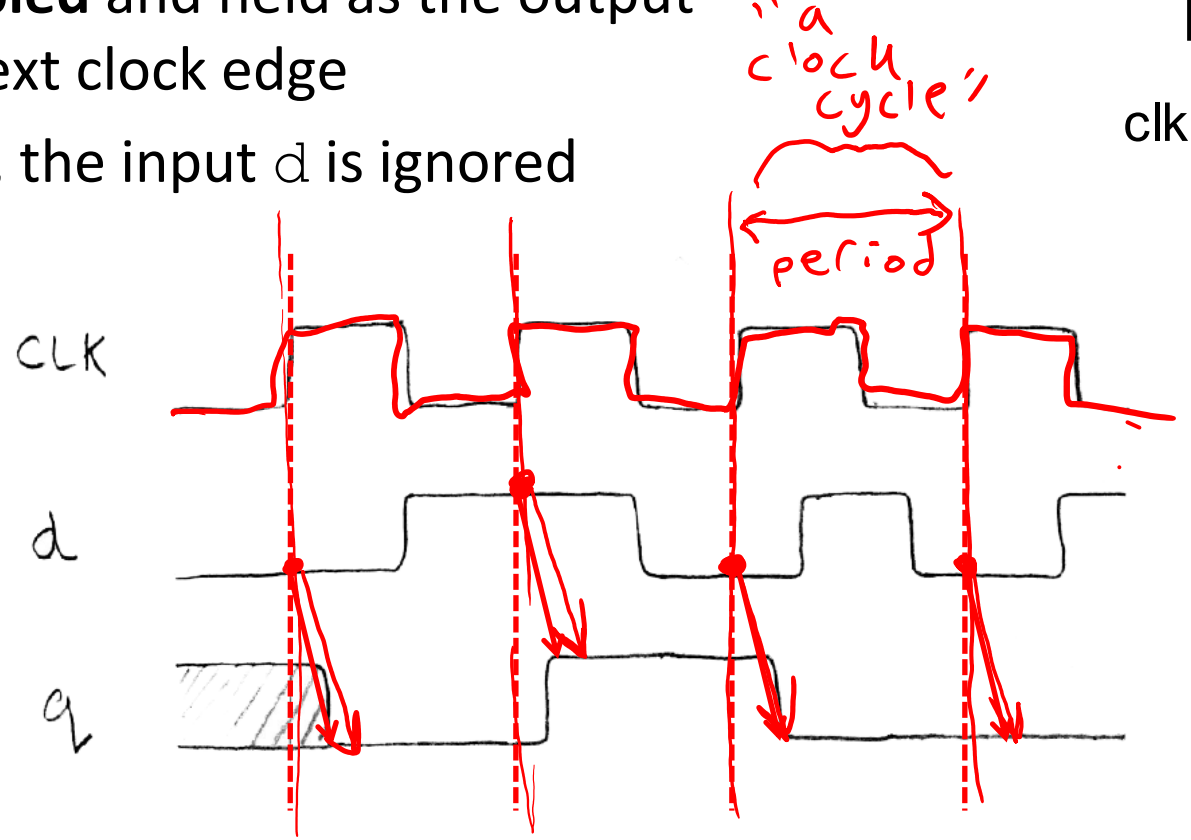


How to control the next iteration of the 'for' loop?

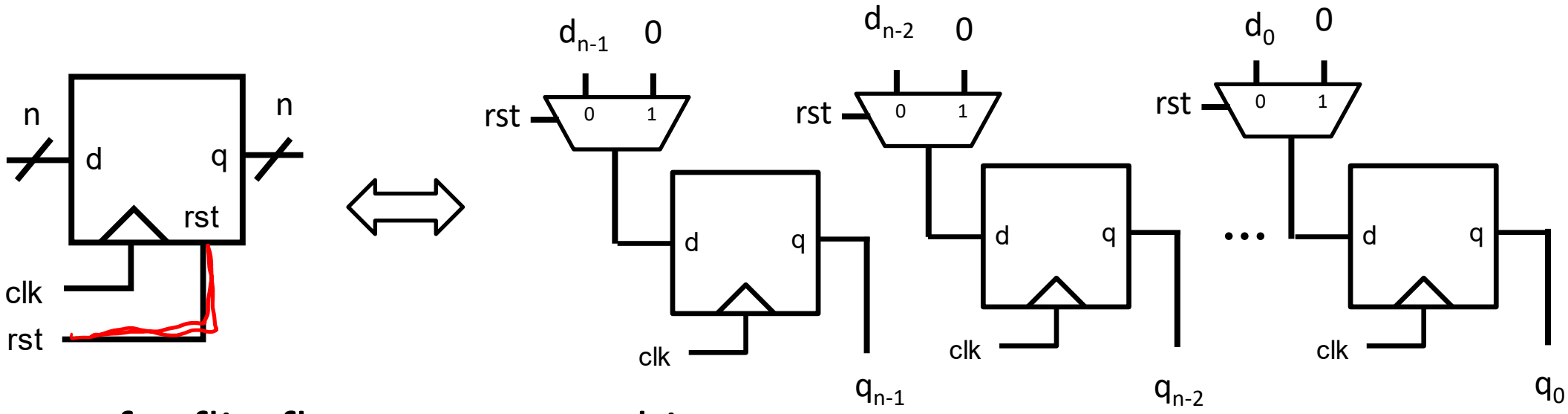
State Element: Flip-Flop

- Positive edge-triggered D-type flip flop
  - On the rising edge of the clock ( ), input `d` is **sampled** and held as the output "`q`" until the next clock edge
  - All other times, the input `d` is ignored





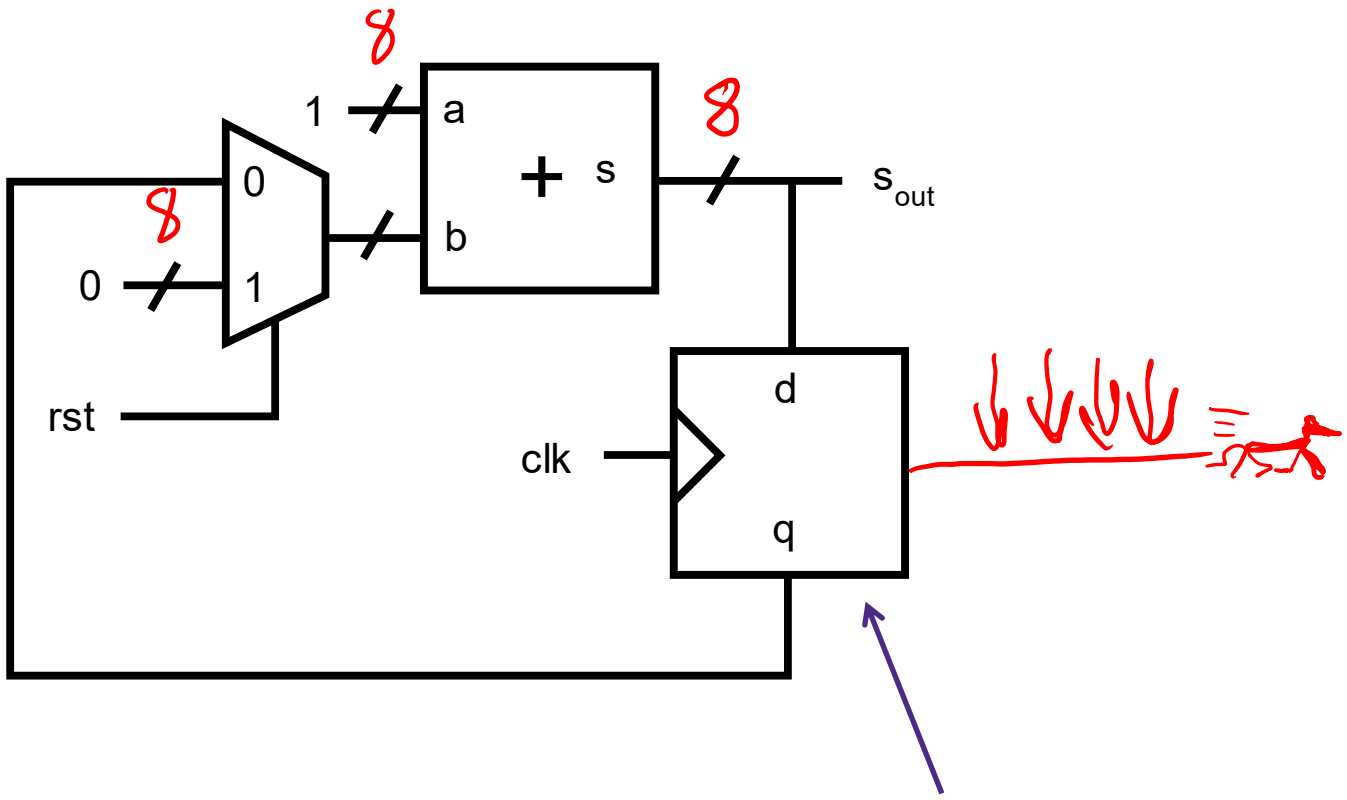
State Element: Register



- A row of n flip-flops to store n bits
  - One for every bit in input/output bus width
- Optional `RESET` input
  - Forces `Q` to 0 when asserted
  - "Synchronous": only detected if high on clock edge
  - "Asynchronous": happens immediately

Timer: Third try

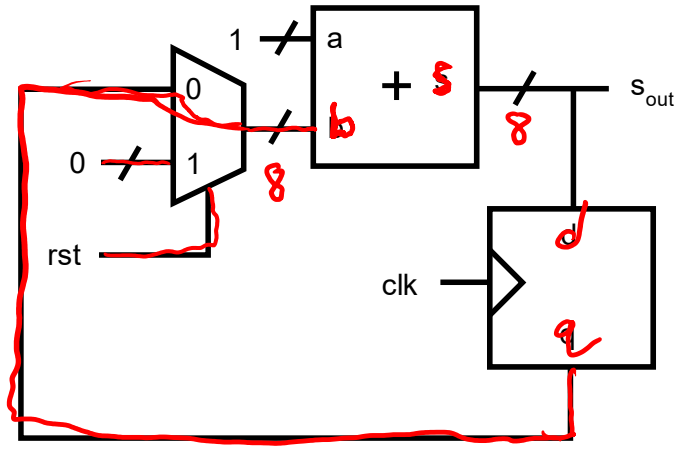
We happy?

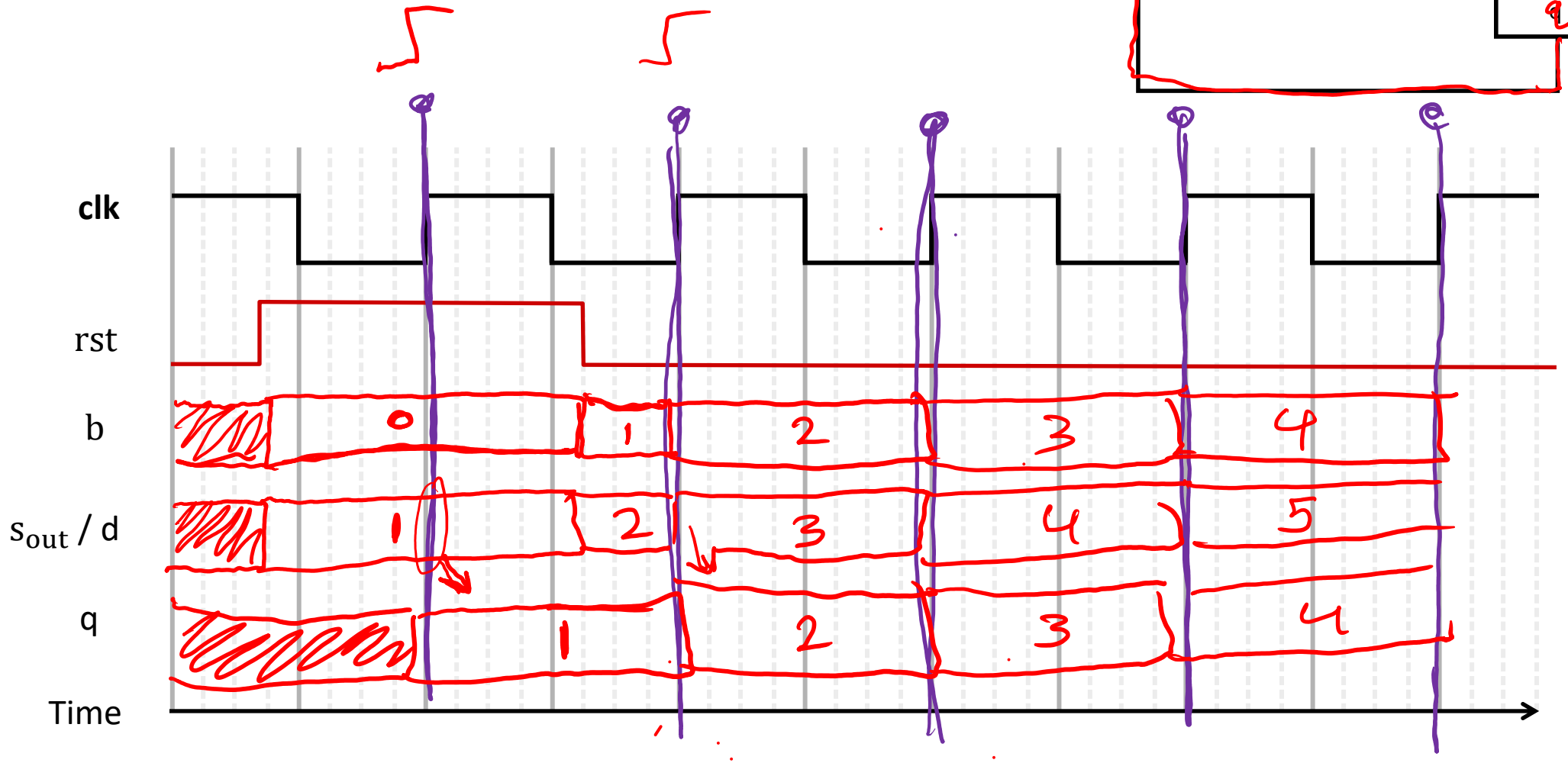


Register holds up the transfer of data to adder

Synchronous waveforms

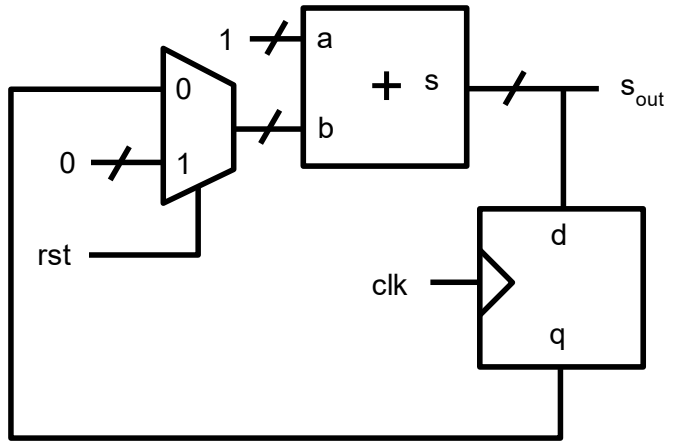
Start by assuming no propagation delays

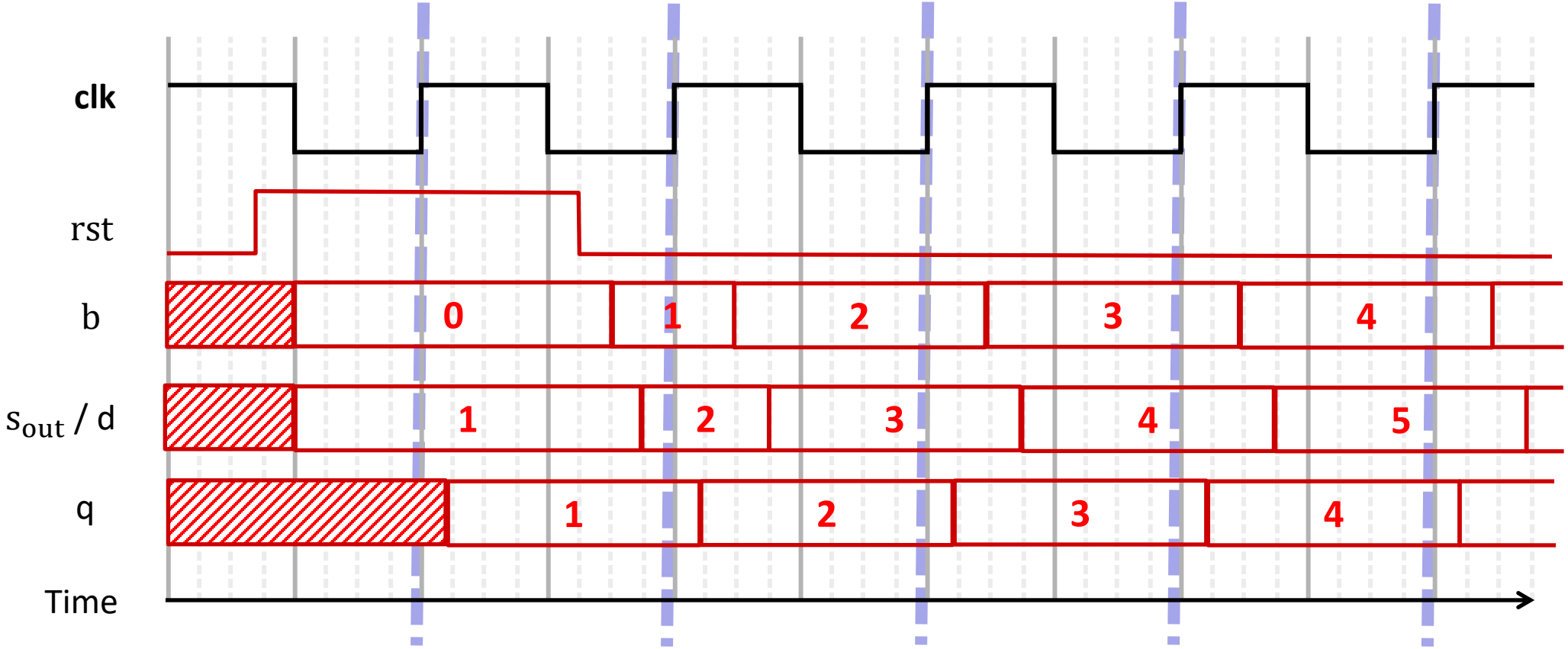




Synchronous waveforms

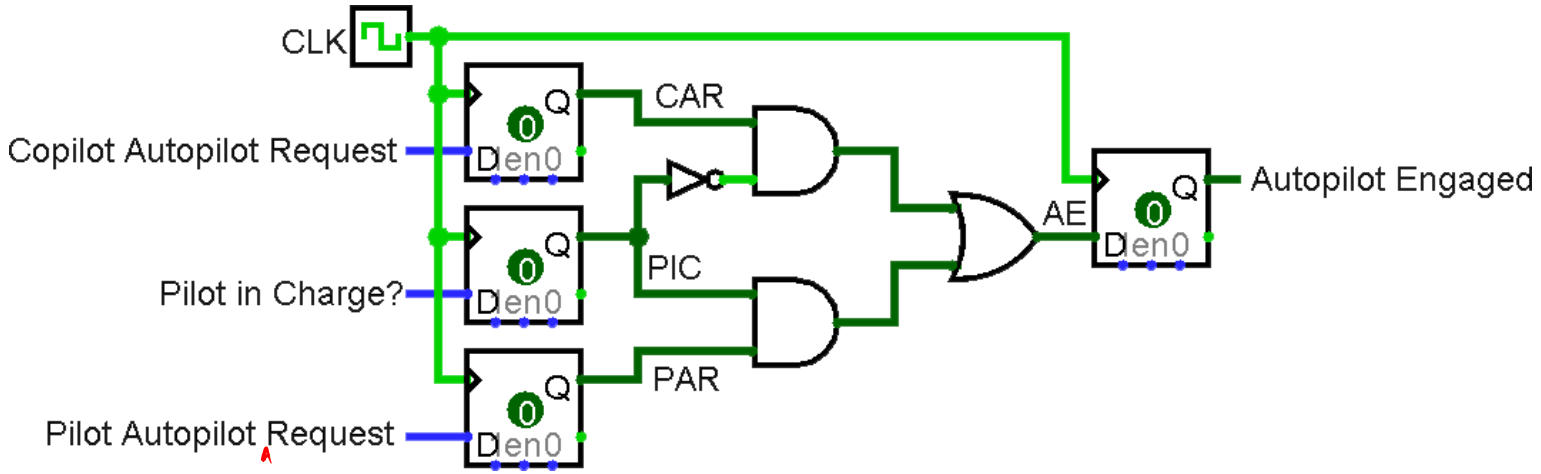
Now a propagation delay of 3ns (1 tick) per block

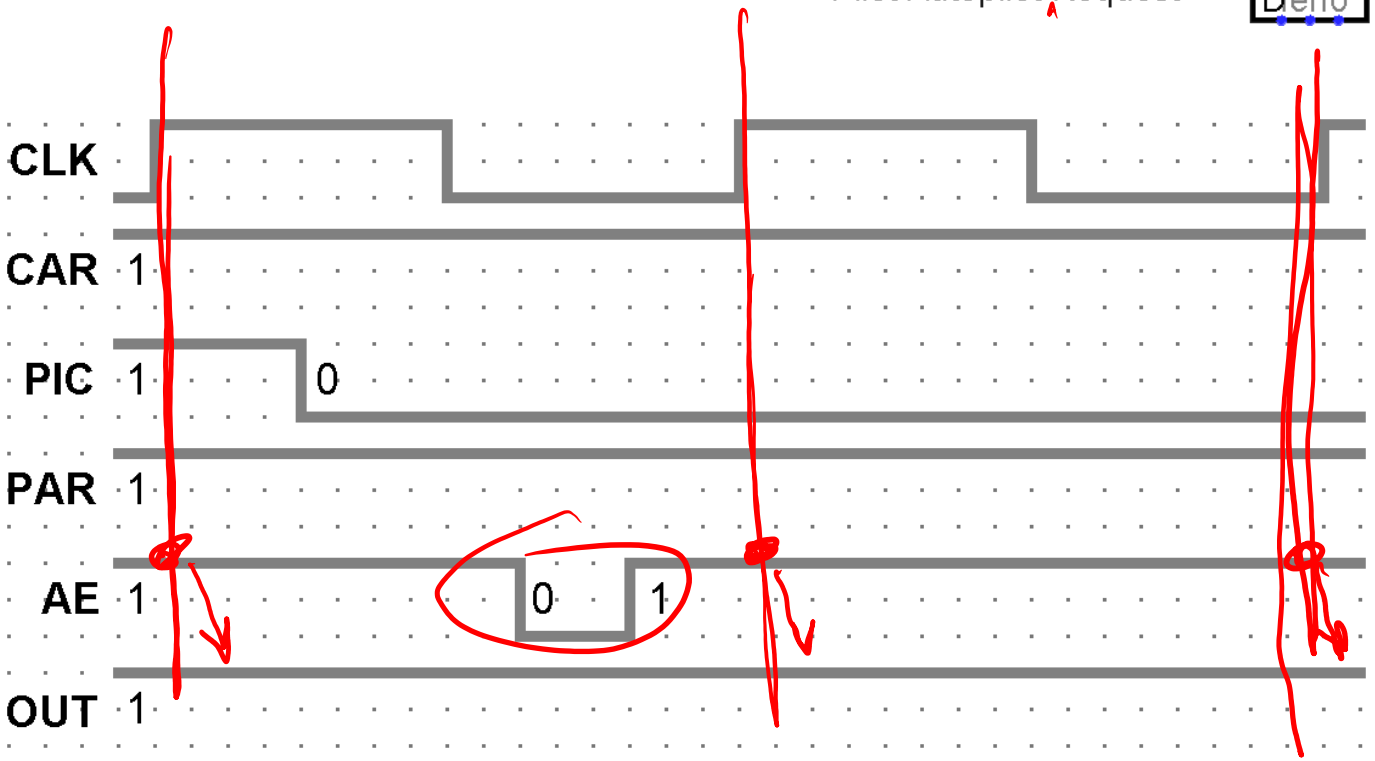




Autopilot Revisited

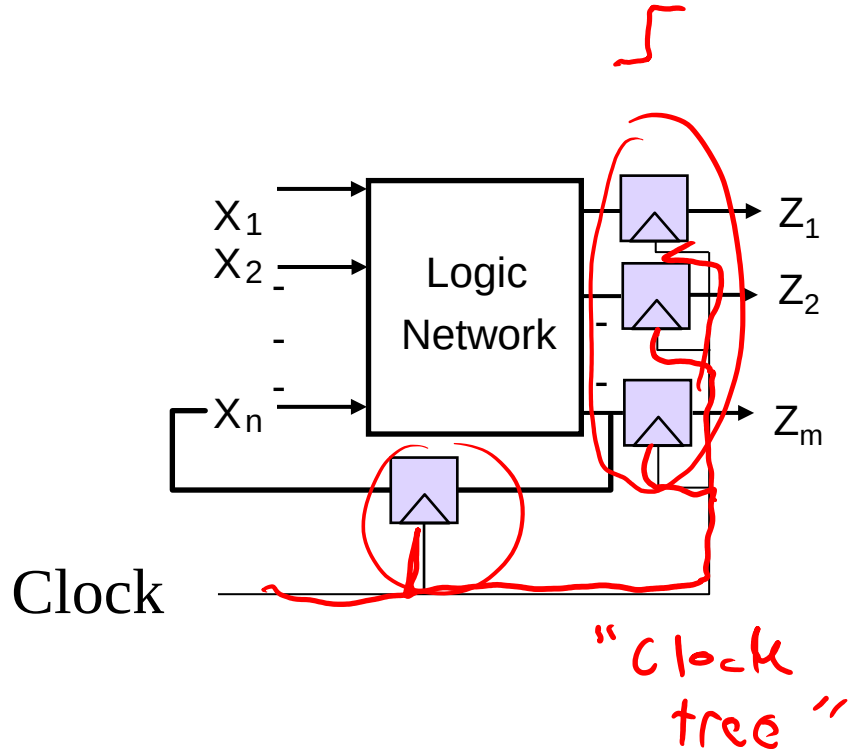
- Flip-flops "filter out" circuit hazards!

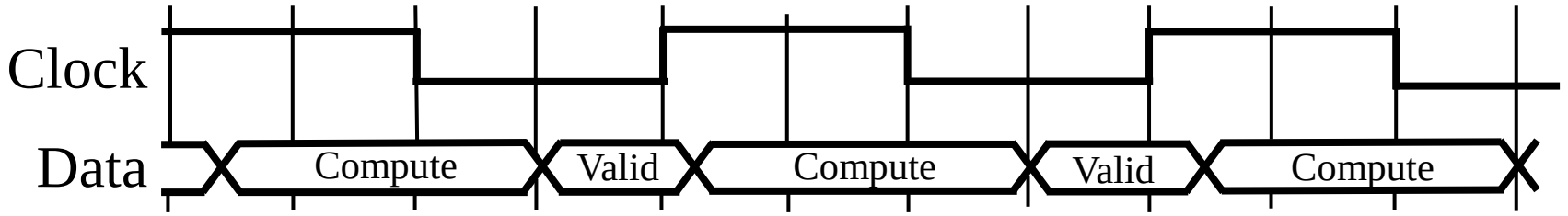




Safe Sequential Circuits

- Clocked elements on feedback, perhaps outputs
  - Clock signal synchronizes operation
  - Clocked elements hide glitches/hazards
  - Output can wiggle with hazards as much as it wants as long as it's **stable around the positive clock edge**
    - More on this in a few weeks ;)





Lecture Outline

- Arithmetic (adders)
- Sequential Logic in theory
- **Sequential Logic in Verilog**

Reminder: "always_comb" blocks

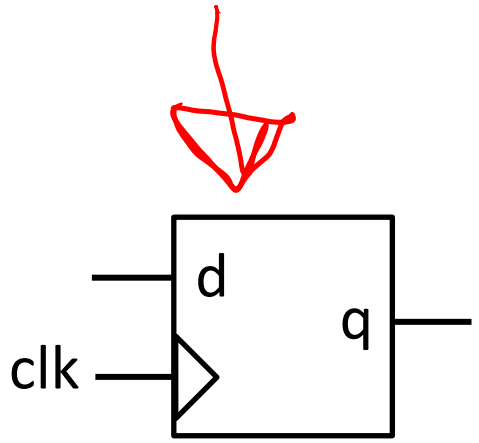
- Verilog requires us to wrap control flow statements in an **`always_comb` block**
  - Block defines the full set of circuits that *may* drive the value on a `logic` variable
  - Idea: the last assignment in an always block to a given variable is the result that gets used

- But I promised there were more species of "`always`" block…

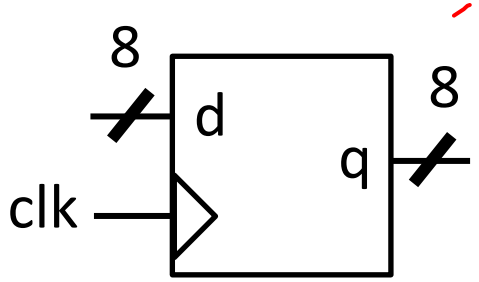
Verilog: Basic D Flip-Flop, Register

```
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
```

sensitivity list




```
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
```



Sequential "always" blocks

- General `always` blocks:
  - "Always" running, but its output only gets sampled when signals in the ***sensitivity list*** change*:*

```
always @(posedge clk)
```

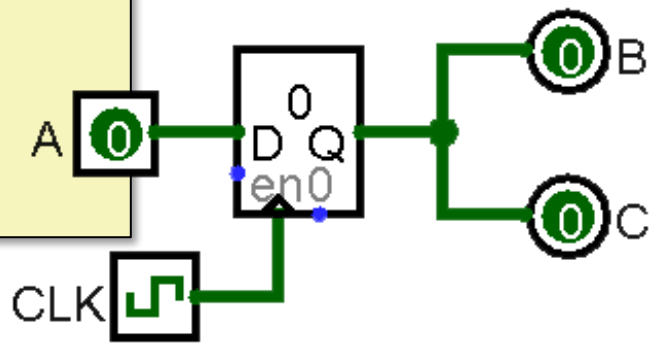
- `always_ff`:
  - Forces SystemVerilog to use flip flops as the stateful element in the circuit
  - **Only** use `always_ff` for sequential logic in this class, never `always`

```
always_ff @ (posedge clk)
```

Blocking vs. Nonblocking

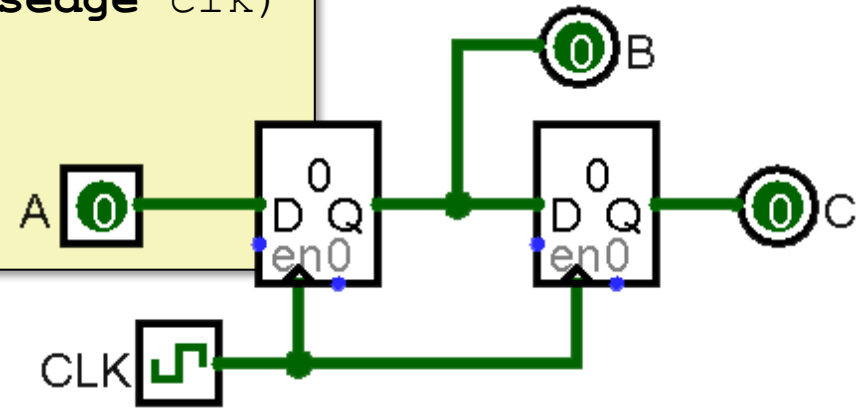
- **Blocking** statement (=): statements executed sequentially
  - Resembles programming languages

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



- **Nonblocking** statement (<=): statements executed "in parallel"
  - Resembles hardware

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



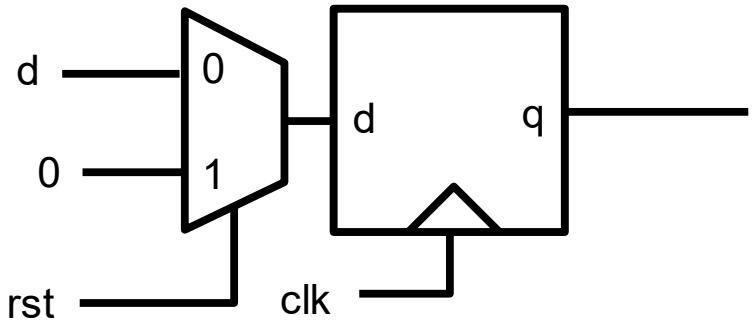
SystemVerilog Coding Guidelines

1) When modeling sequential logic with an `always_ff` block, use ***nonblocking*** assignments (**<=**)
2) When modeling combinational logic with an `always_comb` block, use *blocking* assignments (**=**)
3) When modeling both sequential and combinational logic within the same `always_ff` block, use *nonblocking* assignments
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: Reset Functionality

- Option 1: synchronous reset

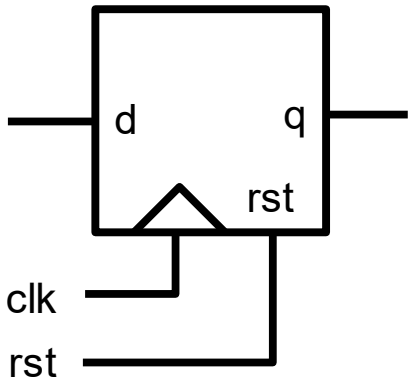
```
module DFFR (q, d, reset, clk);
  output logic q;  // q is state-holding
  input  logic d, reset, clk;

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

endmodule
```



Verilog: Reset Functionality

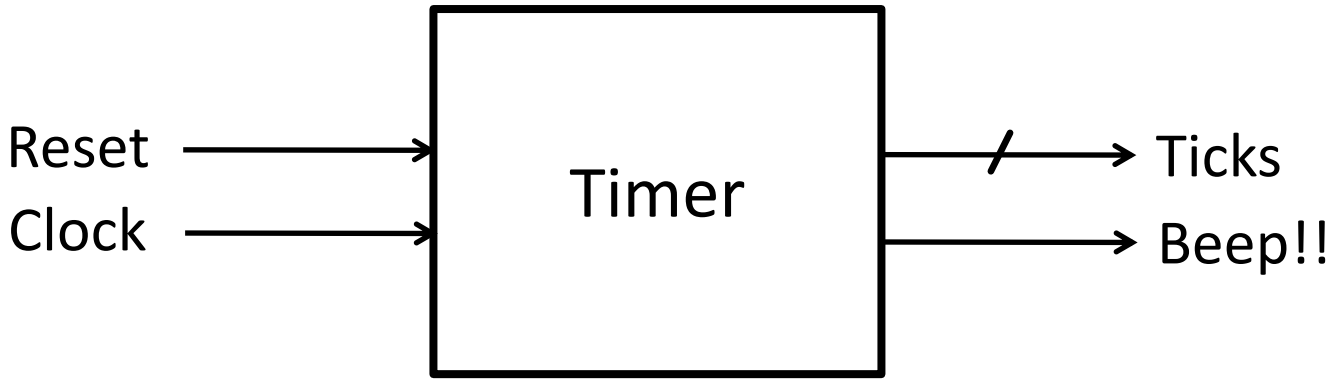
- Option 2: asynchronous reset

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

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

endmodule
```



Exercise for the reader: Advanced Timer

- Draw a circuit diagram for a block that counts up from 0 to parameter N
  - Very similar to our "perpetual timer" example, but it'll need another mux and a block to compare if two numbers are equal
    - Can use a black box for the comparator
    - (but you know enough to design that too, if you wanted to 😉)



Summary

- Binary addition and subtraction can be performed with chained full adders
  - Two's complement allows us to use the same hardware
  - We can detect signed overflow by XORing the carry-in and carry-out of the sign bit
- State elements: 1-bit "flip-flops" and multi-bit "registers"
  - Can be initialized with "reset"
  - Used to store data and control the flow of information between stages of combinational logic
  - Triggered by rising edge of clock signal
- State in Verilog implemented with always_ff blocks
  - Clock indicated through sensitivity list `@(posedge clock)`
  - Non-blocking assignment with <=