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

Instructor: Naomi Alterman

Teaching Assistants:

Derek de Leuw
Isabel Froelich
Kevin Hernandez
Sathvik Kanuri
Aadithya Manoj

Administrivia

- Lab 3 Demos due during your assigned demo slots
  - Don't forget to submit your lab materials *before* Wednesday at 2:30 pm, regardless of your demo time
  - Come to lab with your reports open and bitfiles ready to load up
- Lab 4 – 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

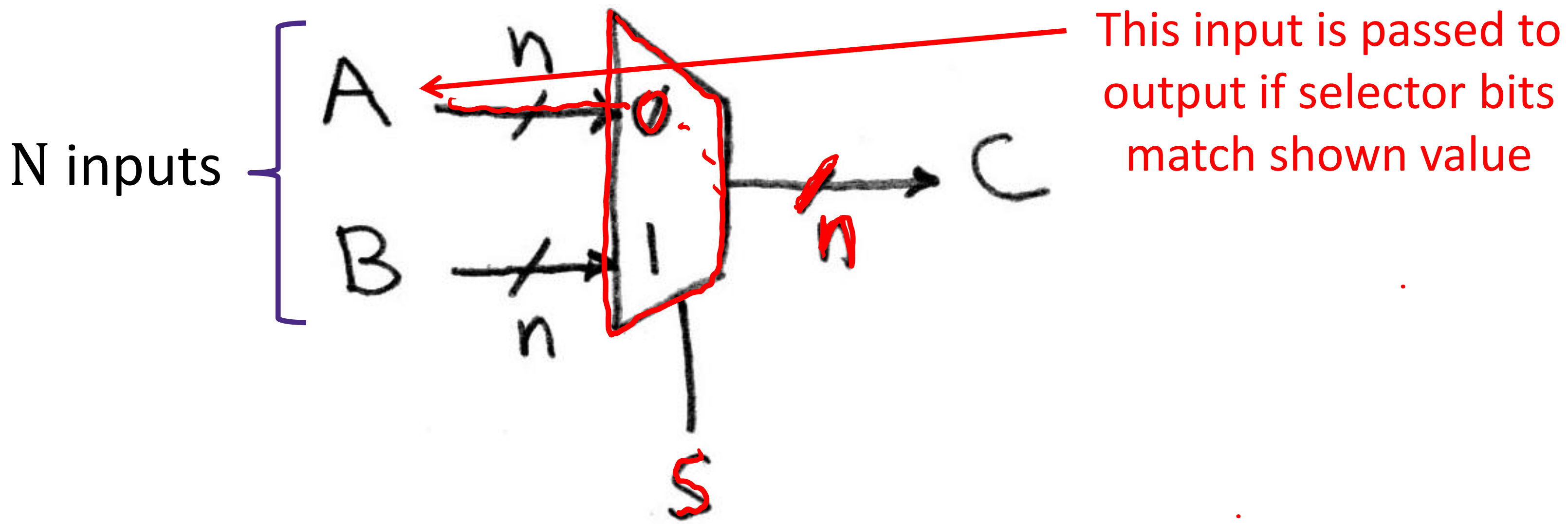
Lecture Outline

- **Multiplexors**
- Adders
- Sequential Logic in theory
- Sequential Logic in Verilog

Data Multiplexor

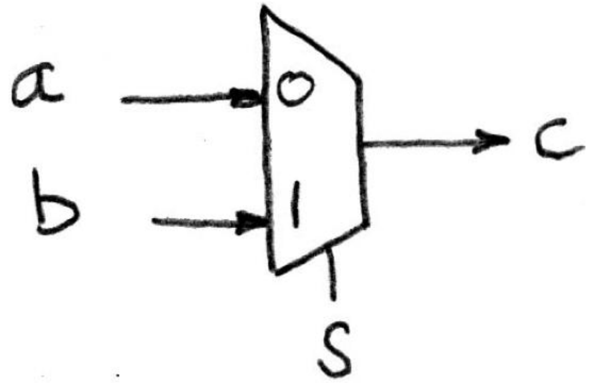
- Multiplexor ("MUX") is a *selector*
  - Use an s-bit "select signal" to direct one of 2^s n-bit wide inputs to output
  - Called a n-bit, N-to-1 MUX

- Example: n-bit 2-to-1 MUX
  - Input S (s bits wide) selects between two inputs of n bits each



Review: Implementing a 1-bit 2-to-1 MUX

- **Schematic:**



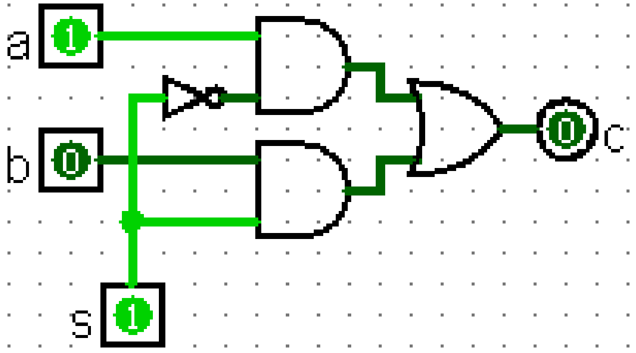
- **Boolean Algebra:**

$$\begin{aligned} c &= \bar{s}a\bar{b} + \bar{s}ab + s\bar{a}b + sab \\ &= \bar{s}(a\bar{b} + ab) + s(\bar{a}b + ab) \\ &= \bar{s}(a(\bar{b} + b)) + s((\bar{a} + a)b) \\ &= \bar{s}(a(1) + s((1)b) \\ &= \bar{s}a + sb \end{aligned}$$

- **Truth Table:**

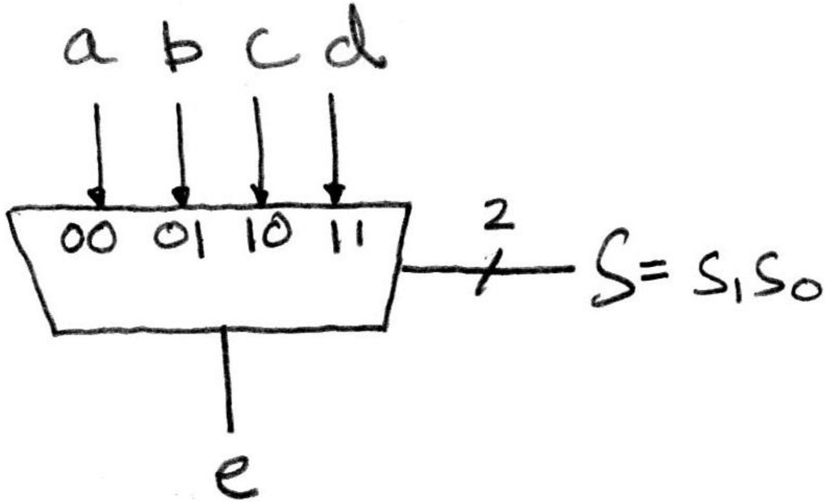
s	a	b	c
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

- **Circuit Diagram:**



1-bit 4-to-1 MUX

- **Schematic:**



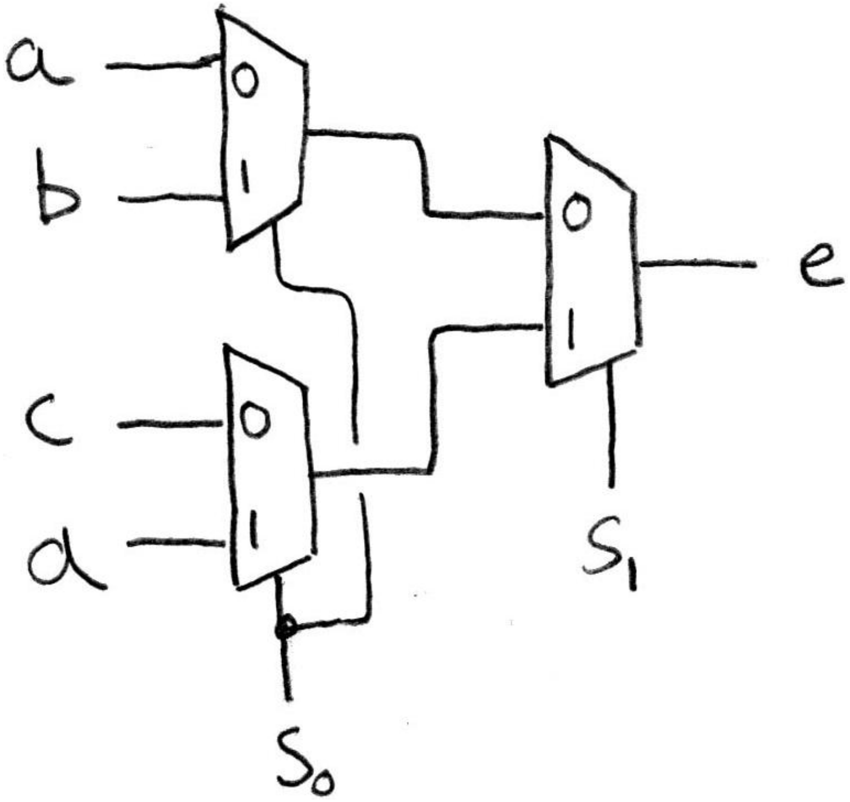
- **Truth Table:** How many rows?

- **Boolean Expression:**

$$e = \overline{s_1}\,\overline{s_0}a + \overline{s_1}s_0 b + s_1\overline{s_0}c + s_1 s_0 d$$

1-bit 4-to-1 MUX

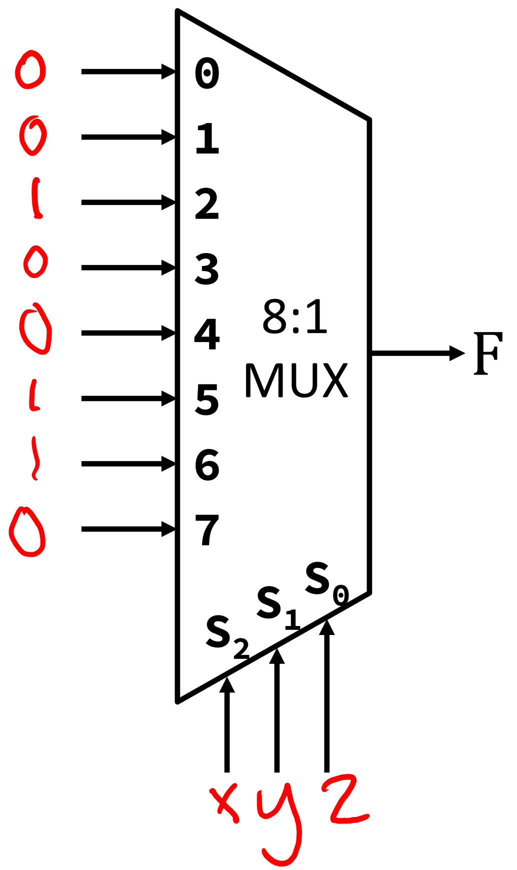
- Can we leverage what we've previously built?
  - Alternative hierarchical approach:



Multiplexers in General Logic

- Implement $F = X\bar{Y}Z + Y\bar{Z}$ with a 8:1 MUX

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1		1	1
1	1	0	1
1	1	1	0



Lecture Outline

- Multiplexors
- **Adders**
- Sequential Logic in theory
- Sequential Logic in Verilog

Review: Unsigned Integers

- Unsigned values follow the standard base 2 system
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_7 2^7 + b_6 2^6 + \cdots + b_1 2^1 + b_0 2^0$
- In n bits, represent integers 0 to 2^n-1
- Add and subtract using the “carry” and “borrow” rules, just in binary

1+1=2
b10

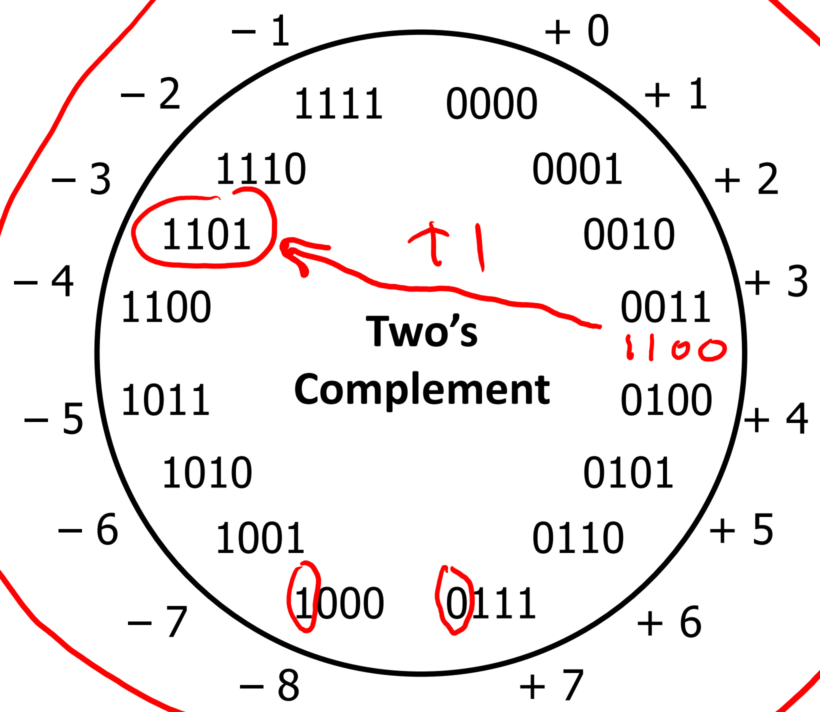
```
  63     00111111       64     01000000
+  8    +00001000     -  8    -00001000
  71     01000111       56     00111000
```

Review: Two's Complement (Signed)

b_{w-1} has weight -2^{w-1}, other bits have usual weights $+2^{i}$

- **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 )`



Addition and Subtraction in Hardware

- The same bit manipulations work for both unsigned and two's complement numbers!
  - Perform subtraction via adding the negated 2nd operand:
    $A - B = A + (-B) = A + (\sim B) + 1$

- 4-bit examples:

	Two's	Un
0010	+2	2
+ 1100	−4	12

	Two's	Un
1000	−8	8
+ 0100	+4	4

	Two's	Un
0110	+6	6
− 0010	+2	2

	Two's	Un
1111	−1	15
− 1110	−2	14

Half Adder (1 bit)

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

0/1
0/1
0/1/2

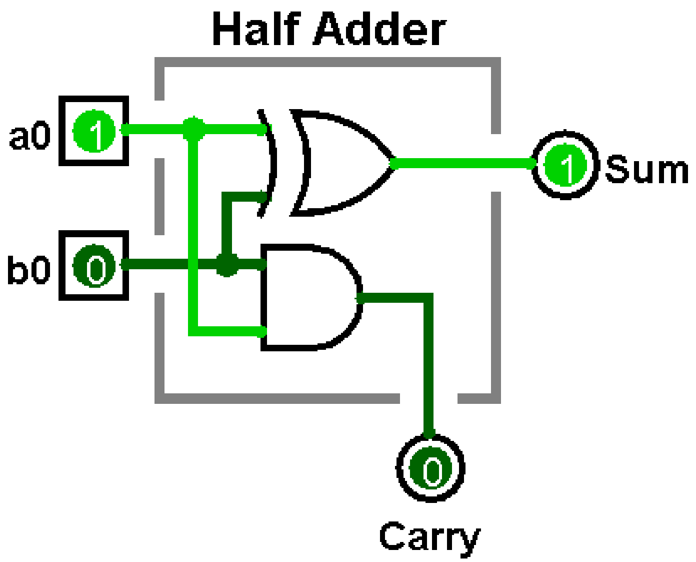
Carry-out bit

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

$\text{Carry} = a_0 b_0$

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

XOR



Full Adder (1 bit)

0/1

Possible carry-in c_1

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

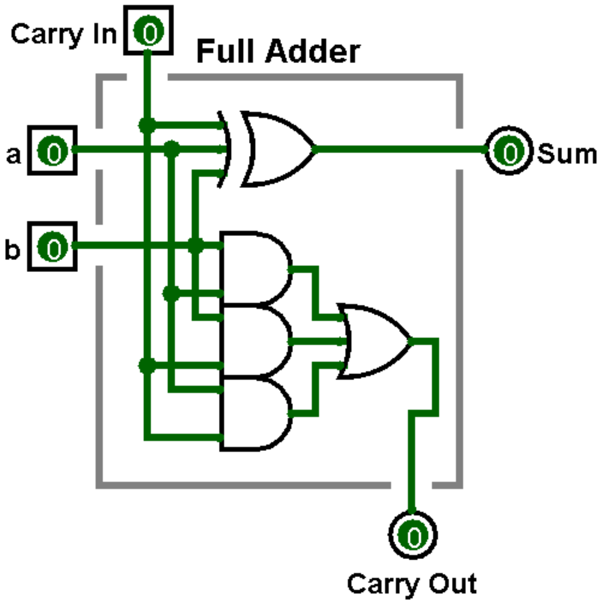
Carry-in ↘ Carry-out ↙

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

1 if Majority of inputs are 1

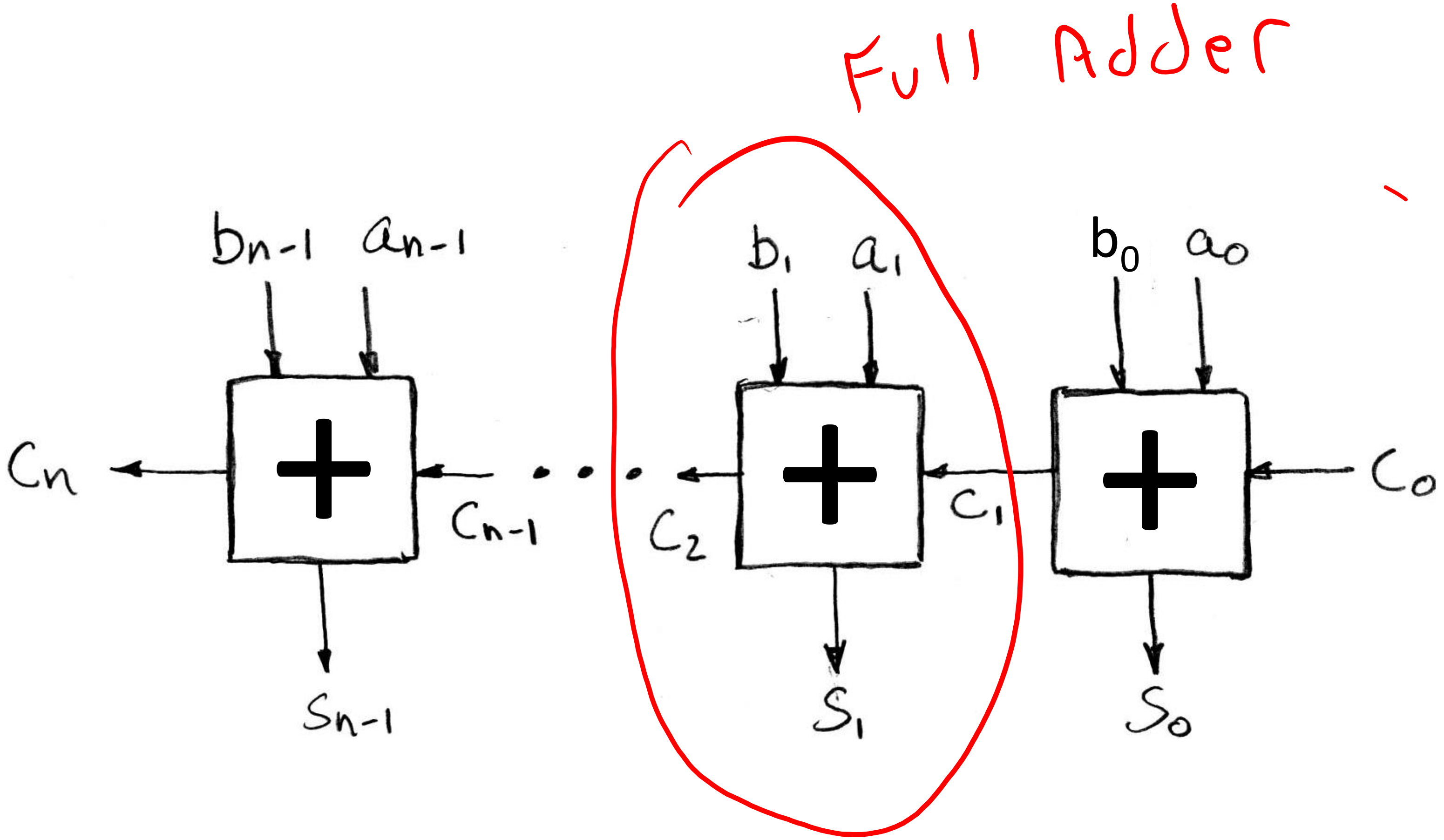
$$s_i = \mathrm{XOR}(a_i, b_i, c_i)$$

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



Multi-Bit Adder (N bits)

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



1-bit Adders in Verilog

- What's wrong with this?
  - Truncation!

```
module halfadd1 (s, a, b);
  output logic s;
  input  logic a, b;

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

- Fixed:
  - Use of `{sig, …, sig}` for *concatenation*

```
module halfadd2 (c, s, a, b);
  output logic c, s;
  input  logic a, b;

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

2 bit signal
MSB | LSB

Ripple-Carry Adder in Verilog

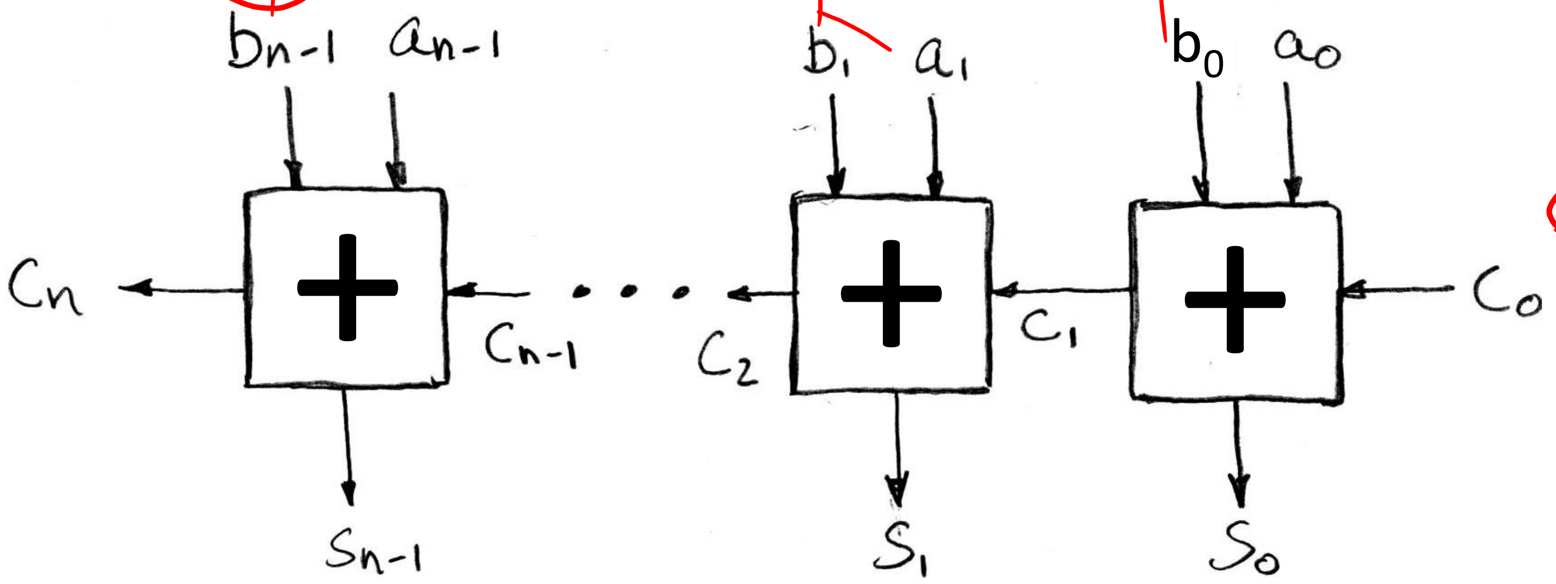
```
module fulladd (cout, s, cin, a, b);
  output logic cout, s;
  input  logic cin, a, b;

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

- Chain full adders?

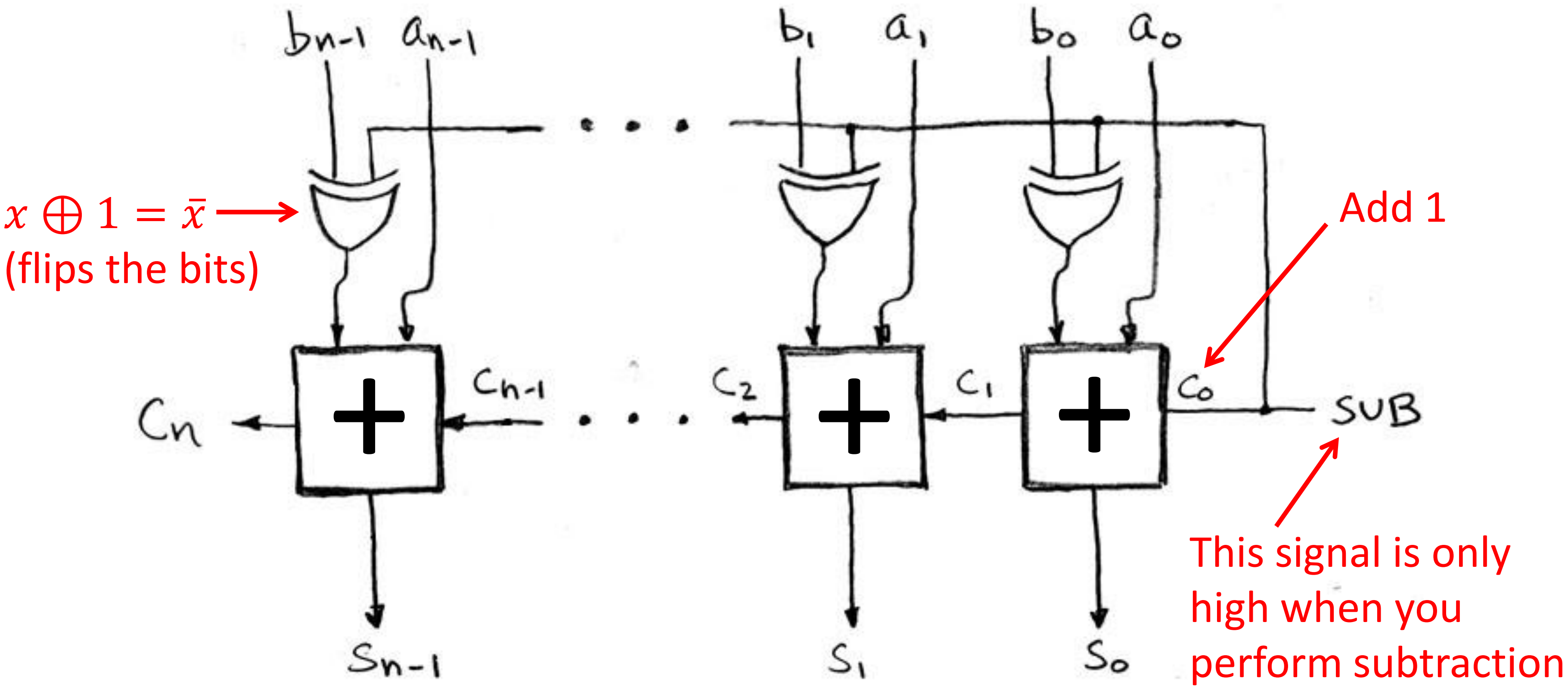
```
module add2 (cout, s, cin, a, b);
  output logic cout; output logic [1:0] s;
  input  logic cin;  input  logic [1:0] a, b;
  logic  c1;

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

Subtraction?

- Can we use our multi-bit adder to do subtraction?
  - Flip the bits and add 1?
    - $X \oplus 1 = \overline{X}$
    - $CarryIn_0$ (using full adder in all positions)



Multi-bit Adder/Subtractor



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 > UMax or < Umin

0b11..1 0b00..0

- **Signed Overflow**
  - Result of add/sub is > TMax or < TMin
  - (+) + (+) = (–) or (–) + (–) = (+)

0b011..1 0b100..0

Signed Overflow Examples

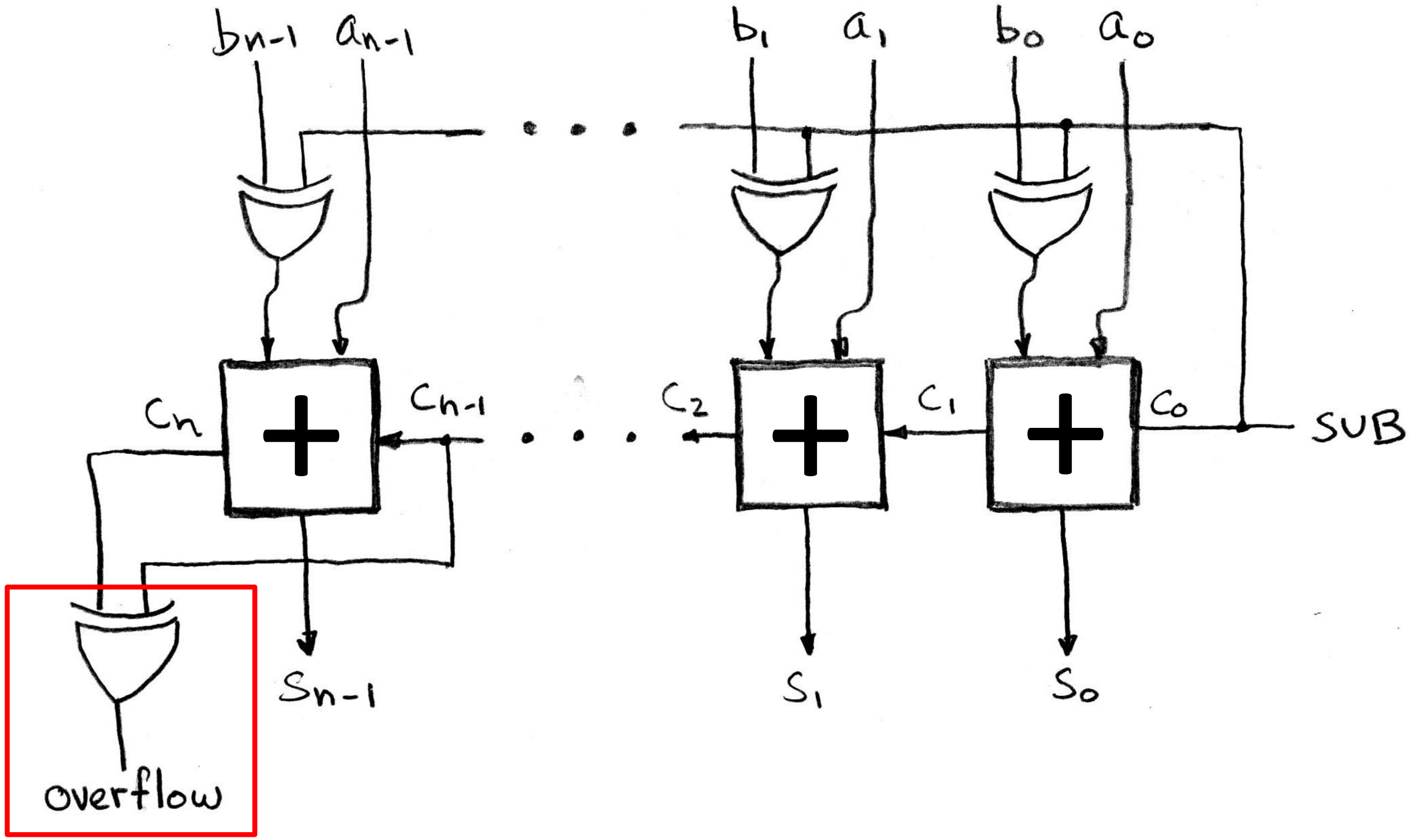
```
                Two's
  0 1 0 1        +5
+ 0 0 1 1        +3
---------
```

```
                Two's
  1 0 0 1        -7
+ 1 1 1 0        -2
---------
```

```
                Two's
  0 1 0 1        +5
+ 0 0 1 0        +2
---------
```

```
                Two's
  1 1 0 0        -4
+ 0 1 0 0         4
---------
```

Multi-bit Adder/Subtractor with Overflow



Add/Sub in Verilog (parameterized)

#define
template< >

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

```
module addN #(parameter N=32) (OF, CF, S, sub, A, B);
  output logic         OF, CF;
  output logic [N-1:0] S;
  input  logic         sub;
  input  logic [N-1:0] A, B;
  logic  [N-1:0] D;    // possibly flipped B
  logic          C2;   // second-to-last carry-out

  always_comb begin
    D = B ^ {N{sub}};  // replication operator
    {C2, S[N-2:0]} = A[N-2:0] + D[N-2:0] + sub;
    {CF, S[N-1]} = A[N-1] + D[N-1] + C2;
    OF = CF ^ C2;
  end
endmodule  // addN
```

  - Here using OF = overflow flag, CF = carry flag (from condition flags in x86-64 CPUs)

Add/Sub in Verilog (parameterized)

```
module addN_tb ();
  logic          sub;
  logic [N-1:0] A, B;
  logic          OF, CF;
  logic [N-1:0] S;

  addN #(.N(4)) dut (.OF, .CF, .S, .sub, .A, .B);

  initial begin
    #100;  sub = 0;  A = 4'b0101;  B = 4'b0010;  //  5 +  2
    #100;  sub = 0;  A = 4'b1101;  B = 4'b1011;  // -3 + -5
    #100;  sub = 0;  A = 4'b0101;  B = 4'b0011;  //  5 +  3
    #100;  sub = 0;  A = 4'b1001;  B = 4'b1110;  // -7 + -2
    #100;  sub = 1;  A = 4'b0101;  B = 4'b1110;  //  5 -(-2)
    #100;  sub = 1;  A = 4'b1101;  B = 4'b0101;  // -3 -  5
    #100;  sub = 1;  A = 4'b0101;  B = 4'b1101;  //  5 -(-3)
    #100;  sub = 1;  A = 4'b1001;  B = 4'b0010;  // -7 -  2
    #100;
  end
endmodule  // addN_tb
```

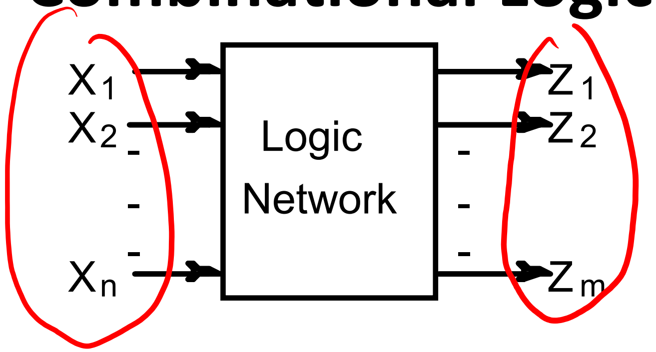
Miso Moment

Lecture Outline

- Multiplexors
- Adders
- **Sequential Logic in theory**
- Sequential Logic in Verilog

Synchronous Digital Systems (SDS)

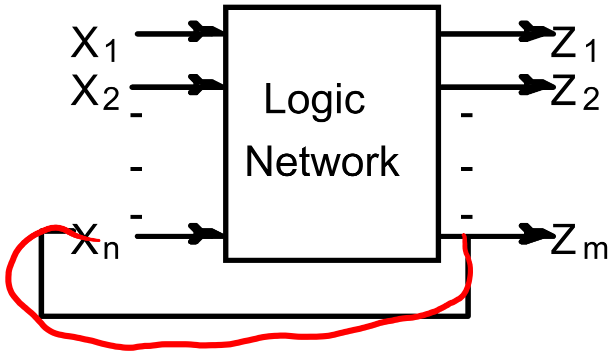
- **Combinational Logic (CL)**



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.

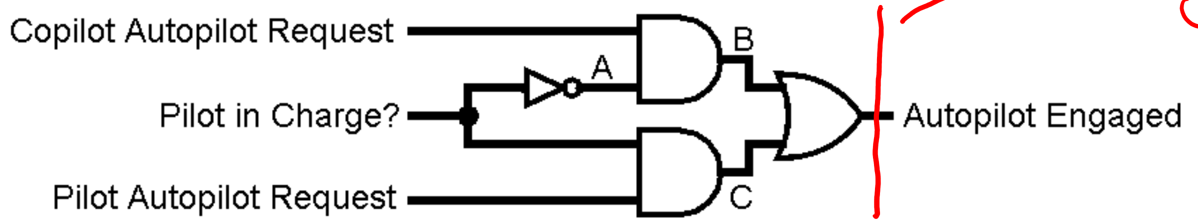
Uses for Sequential Logic

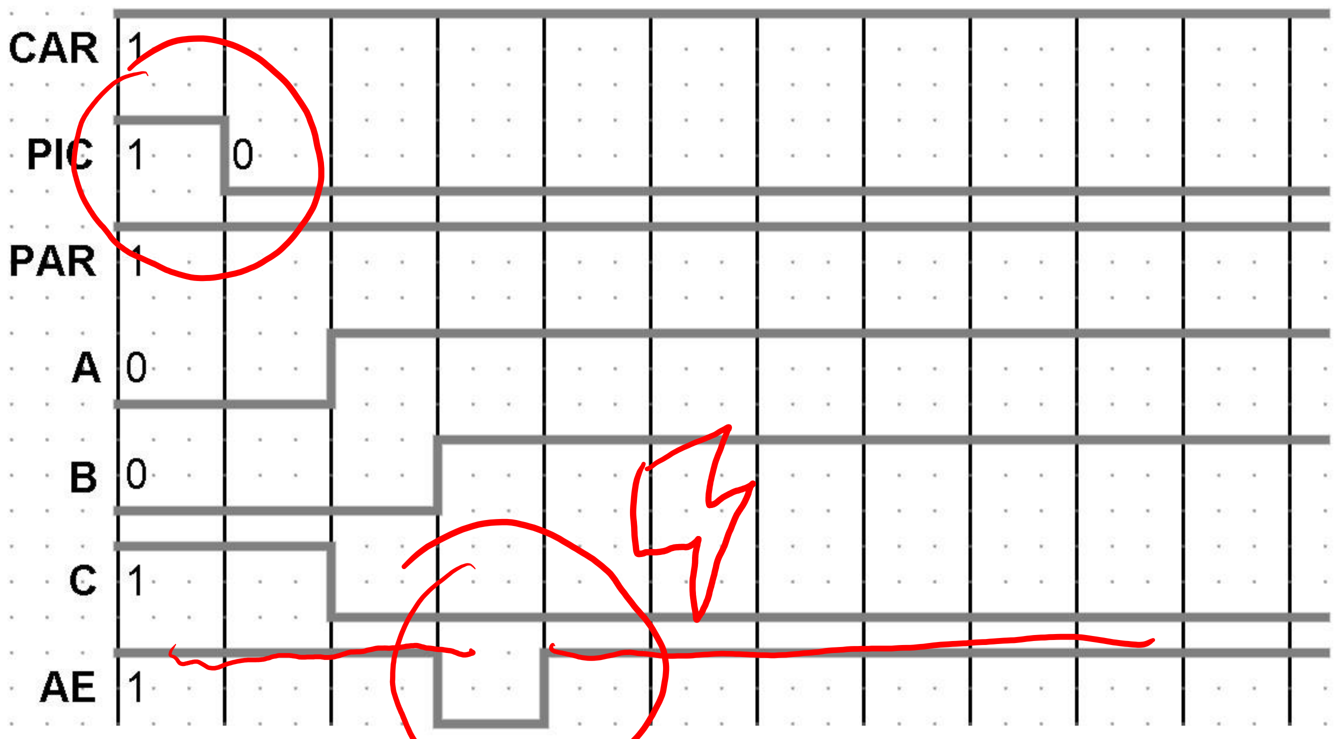
- Place to store values for some amount of time:
  - Registers
  - Memory
- *Help control flow of information between combinational logic blocks*
  - Hold up the movement of information to allow for orderly passage through CL

Control Flow of Information?

- Circuits can temporarily go to incorrect states!

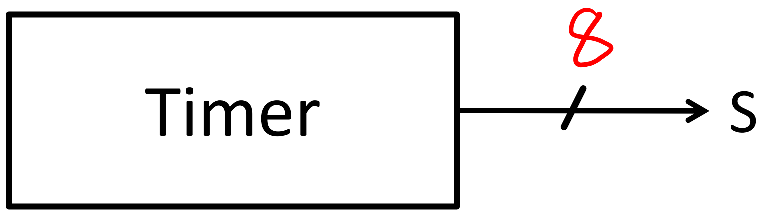
wait until out stabilizes

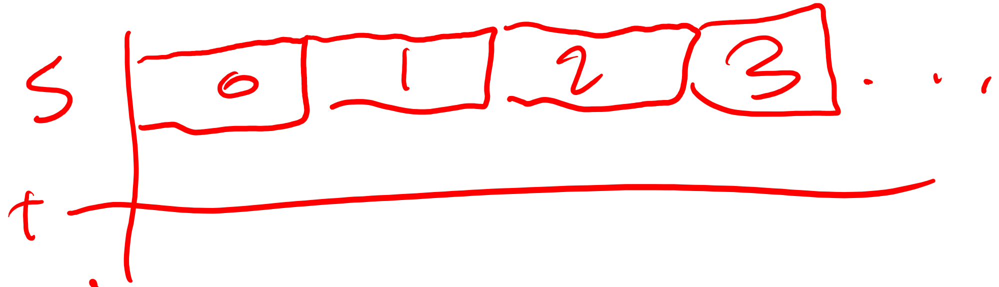




Design example: Perpetual Timer

- 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



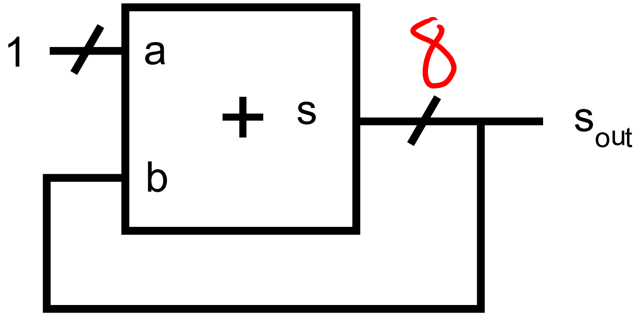


- Want:

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

Timer: First Try

Does this work?

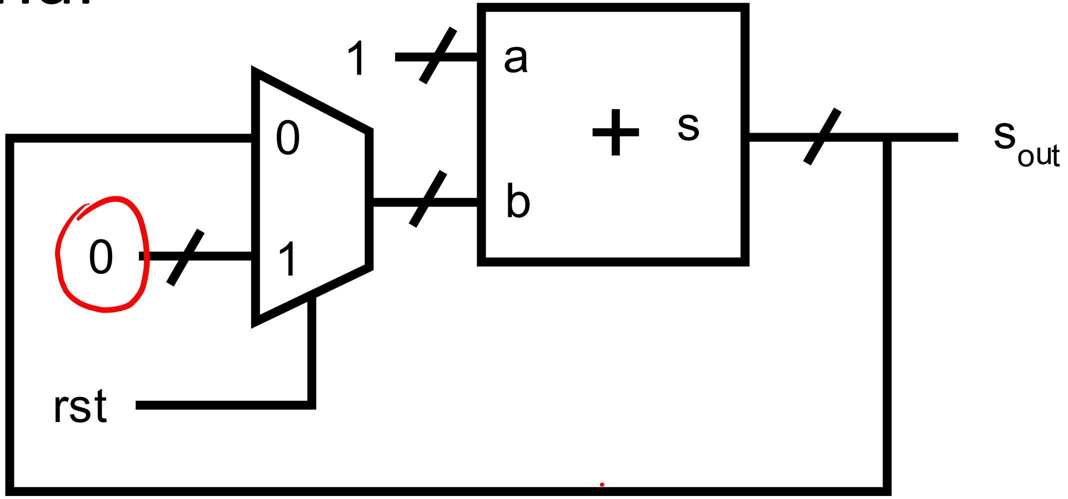


1) How do we say: ‘S=0’?
2) How to control the next iteration of the ‘for’ loop?

Timer: Second Try

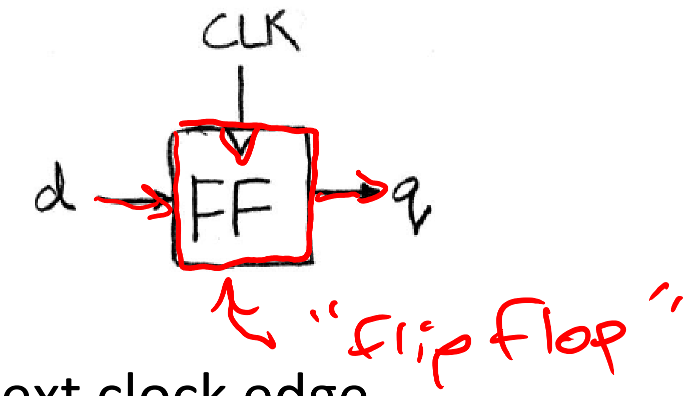
We'll add a "reset" signal

Does this work?

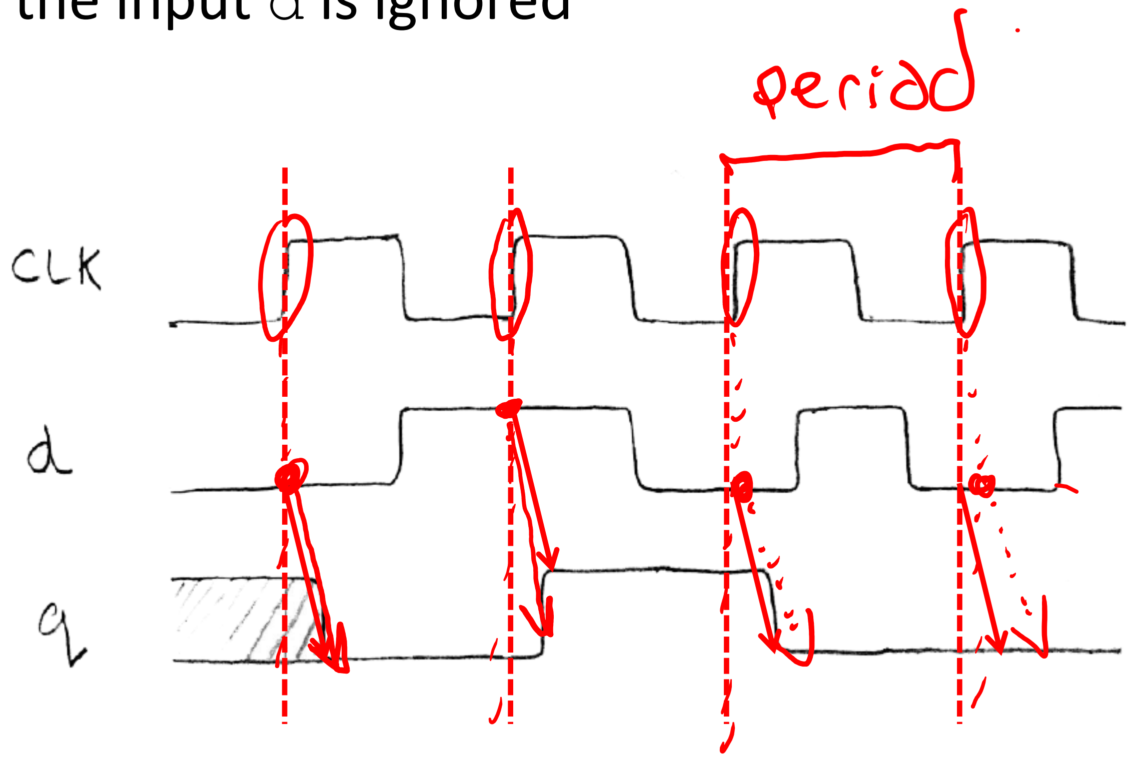


How to control the next iteration of the 'for' loop?

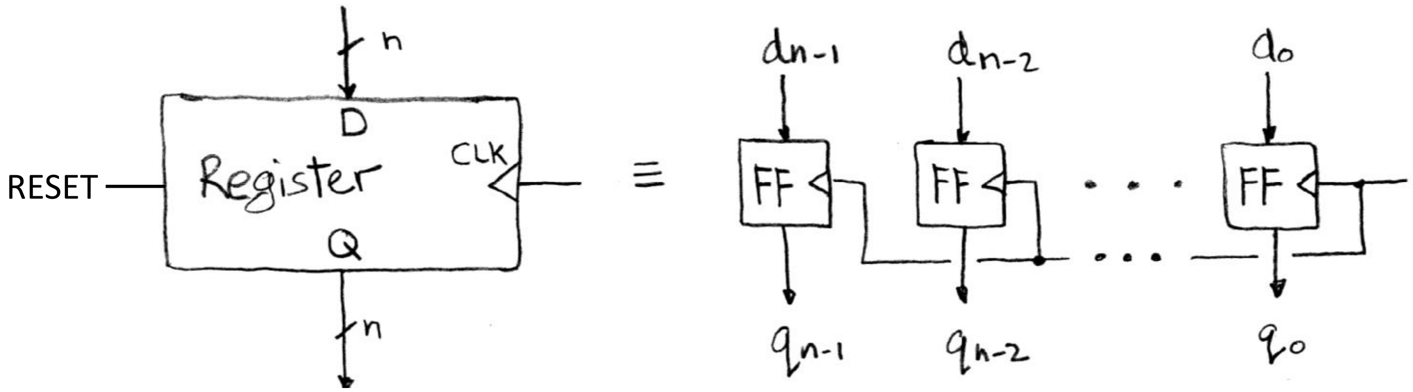
State Element: Flip-Flop



- Positive edge-triggered D-type flip flop
  - On the rising edge of the clock (0 ↑ 1), input `d` is **sampled** and held as the output "`q`" until the next clock edge
  - All other times, the input `d` is ignored



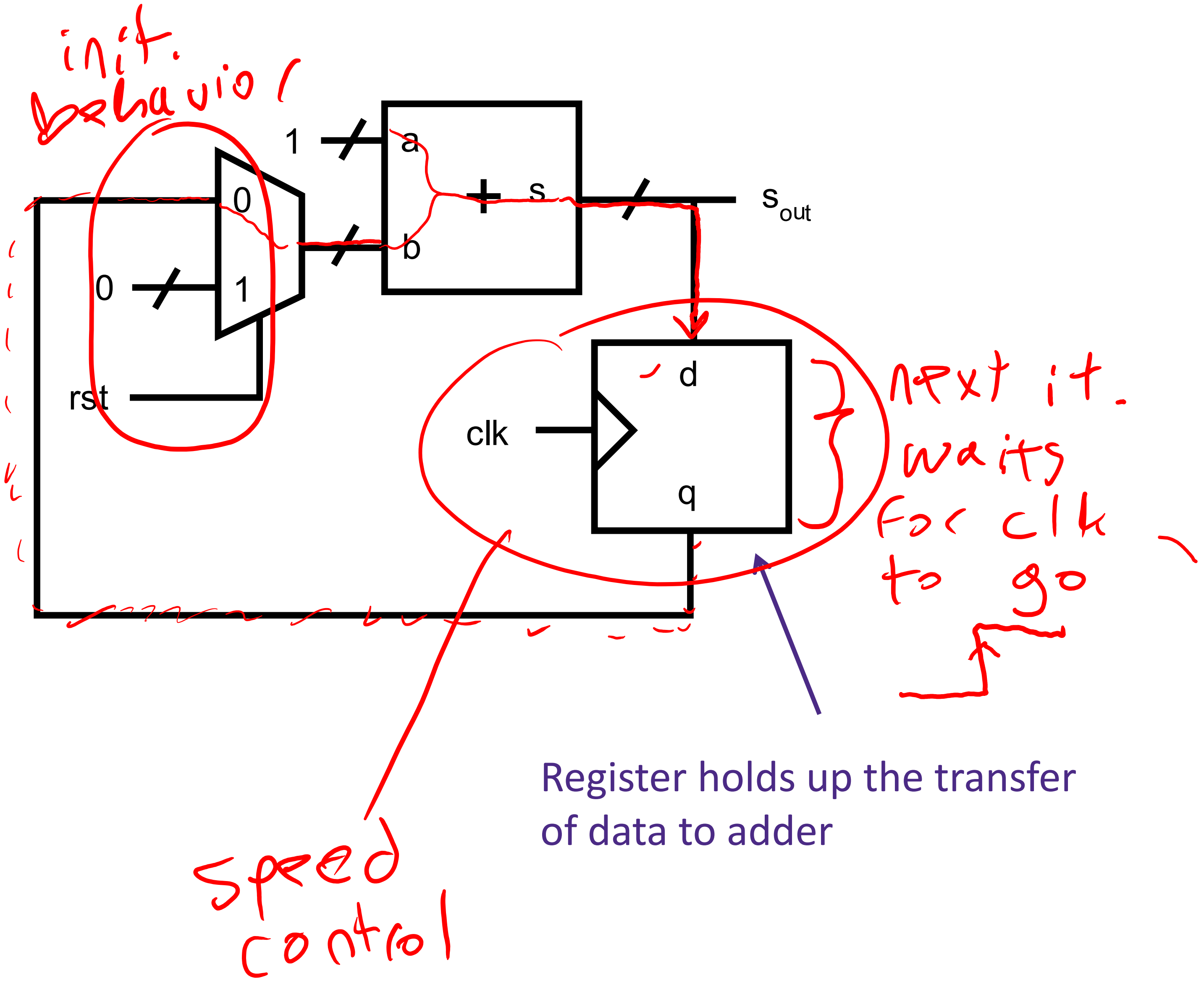
State Element: Register



- n instances of flip-flops together
  - One for every bit in input/output bus width
- Optional synchronous `RESET` input
  - Forces `Q` to 0 when asserted
  - Just shorthand for adding a mux to the FF's input

Timer: Third try

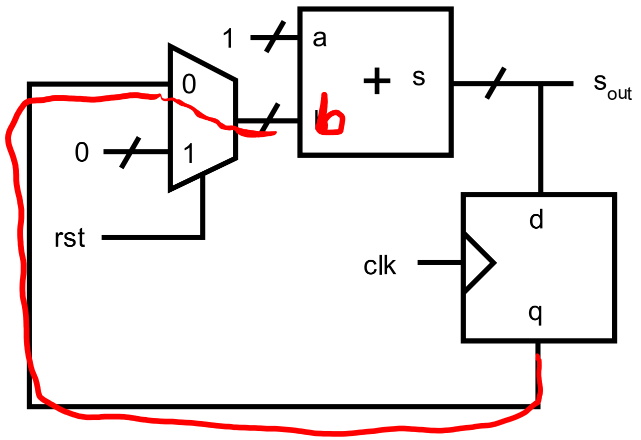
We happy?

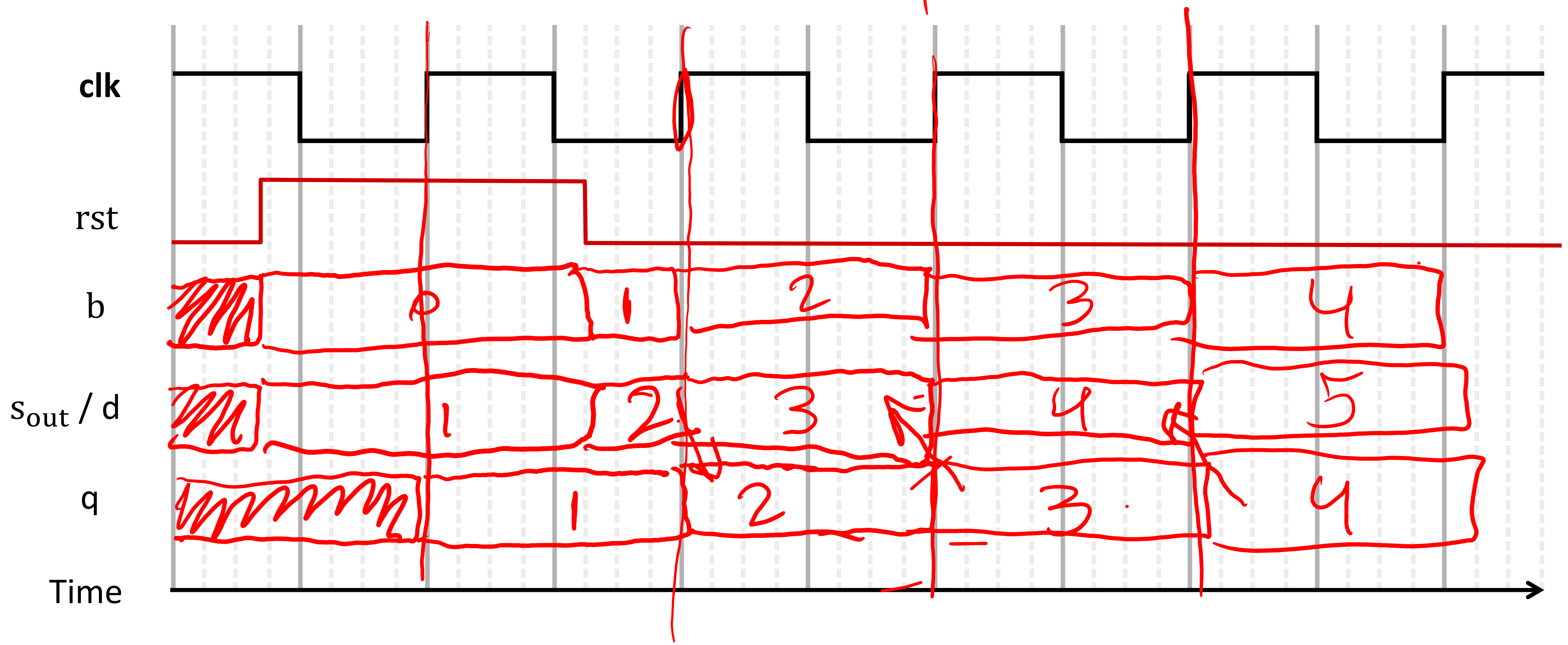


Register holds up the transfer of data to adder

Synchronous waveforms

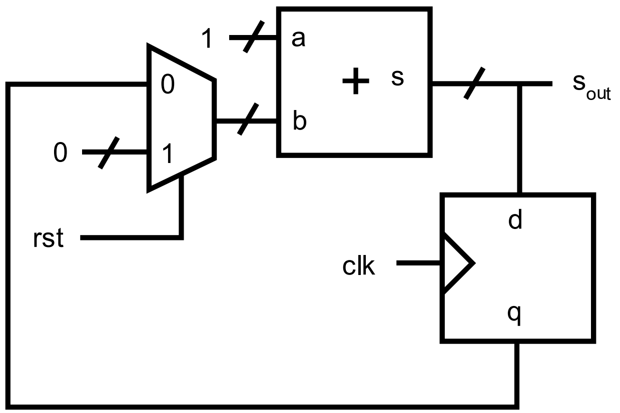
Start by assuming no propagation delays

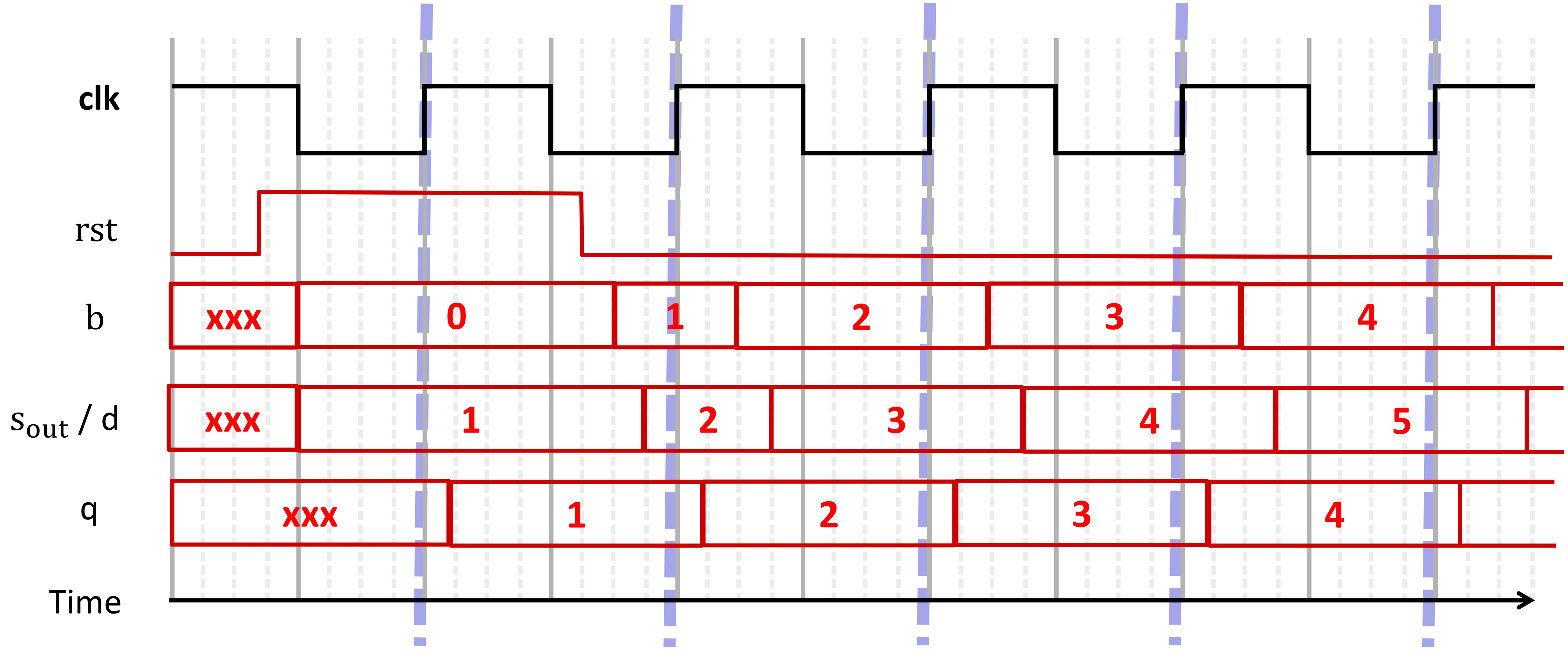




Synchronous waveforms

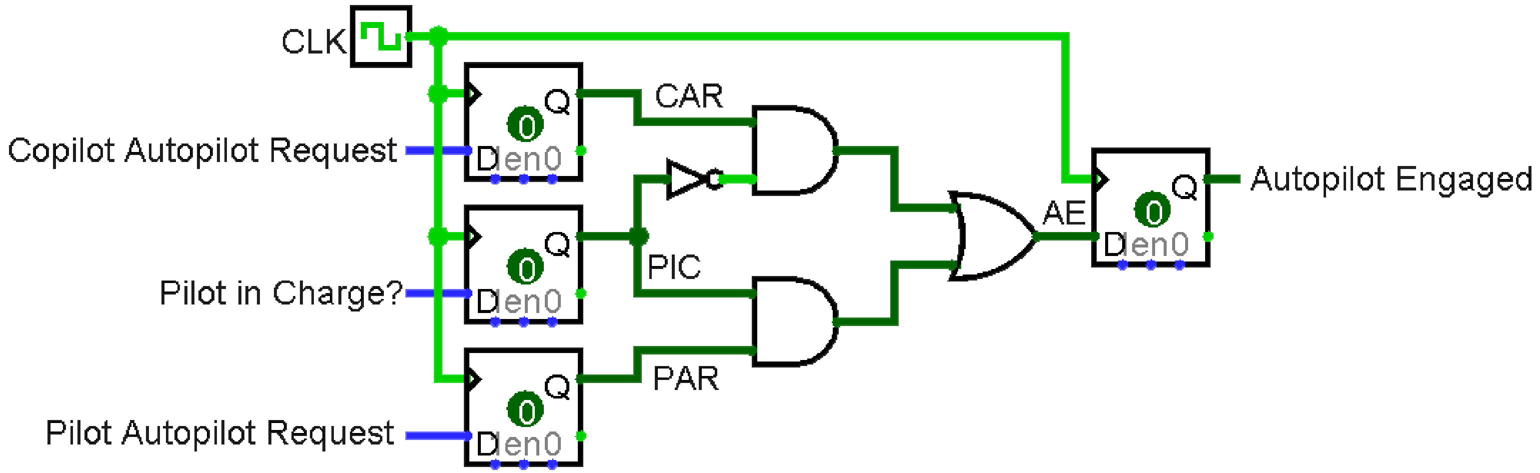
Now a propagation delay of 3ns (1 tick) per block

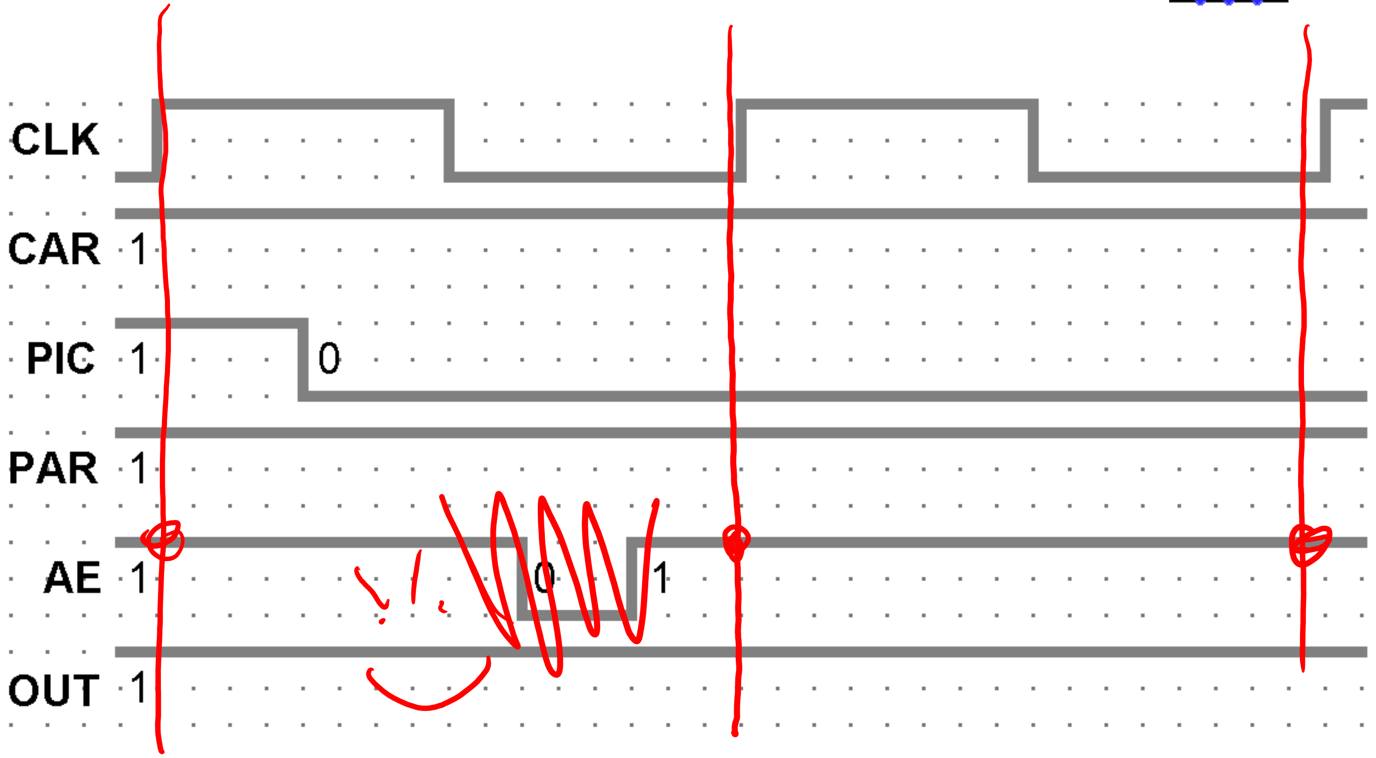




Autopilot Revisited

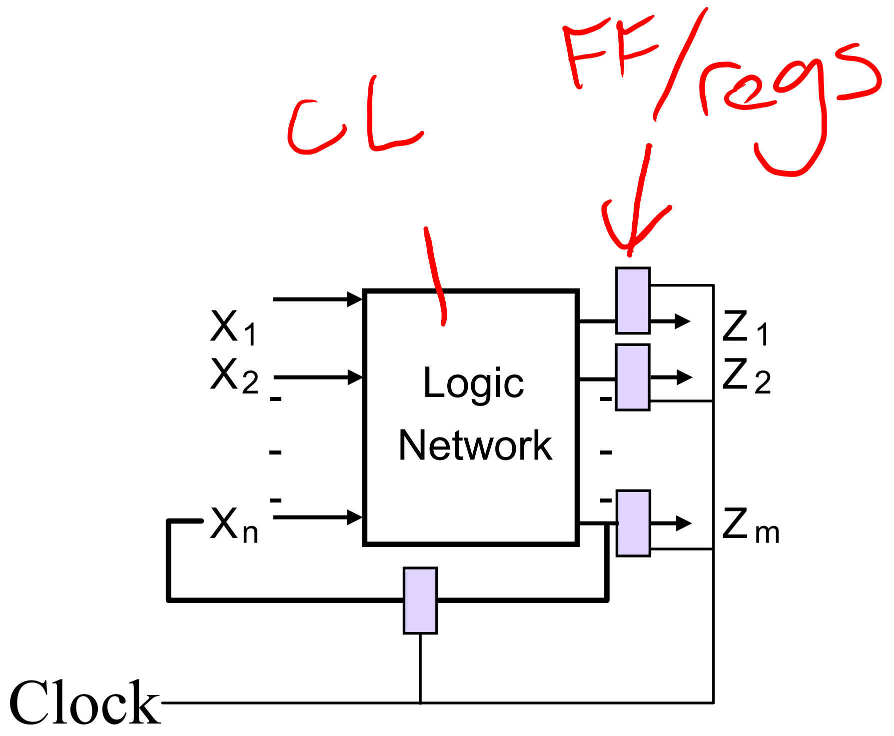
- Flip-flops "filter out" circuit hazards!

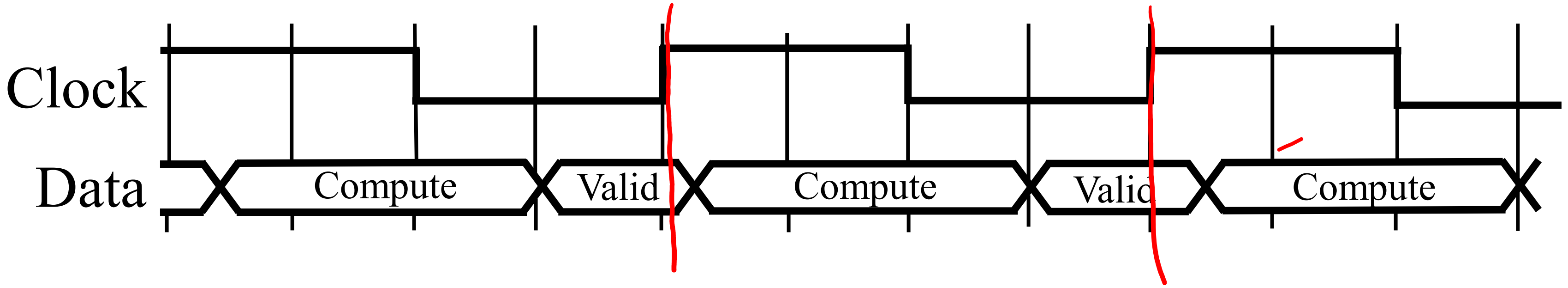




Safe Sequential Circuits

- 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 ;)





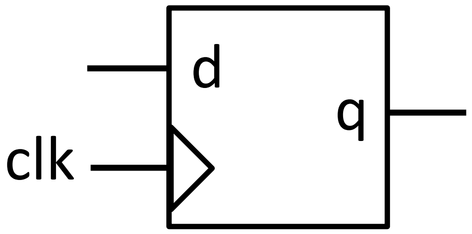
Lecture Outline

- Multiplexors
- Adders
- Sequential Logic in theory
- **Sequential Logic in Verilog**

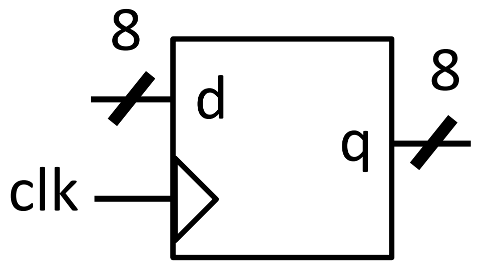
Verilog: Basic D Flip-Flop, Register

```
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
```



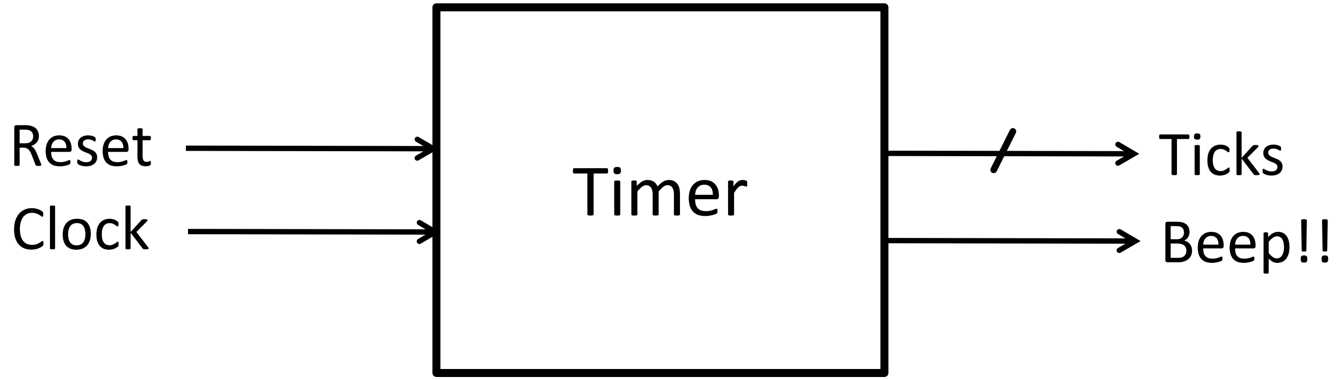
Reminder: "always_comb" blocks

- 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…

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 (1/2)

- Multiplexors switch signals to the output
  - Illustrated in block diagrams as trapezoids with labelled inputs and a select signal
- 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

Summary (2/2)

- State elements controlled by clock
  - Store information
  - Control the flow of information between other state elements and combinational logic

- Registers implemented from flip-flops
  - Triggered by CLK, pass input to output, can reset