

State and Finite State Machines

Prof. Hakim Weatherspoon

CS 3410, Spring 2015

Computer Science

Cornell University

See P&H Appendix B.7, B.8, B.10, B.11

Announcements

Make sure you are

- Registered for class, can access CMS
- Have a Section you can go to.
- *Lab Sections are required.*
 - “Make up” lab sections **only 8:40am Wed, Thur, or Fri**
 - Bring laptop to Labs
- Project partners are required for projects.
 - Have project partner in same Lab Section, if possible

HW1 and Lab1 available today, after lecture

- *Do HW1 problems with lecture, i.e. finish part1 and 2 this week*
- Work alone on both
- But, use your resources
 - Lab Section, Piazza.com, Office Hours, Homework Help Session,
 - Class notes, book, Sections, CSUGLab

Announcements

Make sure to go to *your* Lab Section this week

Completed **Lab1** due *before* winter break, Friday, Feb 13th

Note, a Design Document is due when you submit Lab1 final circuit

Work **alone**

Homework1 is available, after lecture

Due a week before prelim1, Monday, February 23rd

Work on problems incrementally, as we cover them in lecture

Office Hours for help

Work **alone**

Work alone, **BUT** use your resources

- Lab Section, Piazza.com, Office Hours
- Class notes, book, Sections, CSUGLab

Announcements

Check online syllabus/schedule

- <http://www.cs.cornell.edu/Courses/CS3410/2015sp/schedule.html>
- Slides and Reading for lectures
- Office Hours
- ***Pictures of all TAs***
- Homework and Programming Assignments
- **Dates to keep in Mind**
 - Prelims: Tue Mar 3rd and Thur April 30th
 - ***Lab 1: Due next Friday, Feb 13th before Winter break***
 - Proj2: Due Thur Mar 26th before Spring break
 - Final Project: Due when final would be (not known until Feb 14t

Schedule is subject to change

Collaboration, Late, Re-grading Policies

“Black Board” Collaboration Policy

- Can discuss approach together on a “black board”
- Leave and write up solution independently
- Do not copy solutions

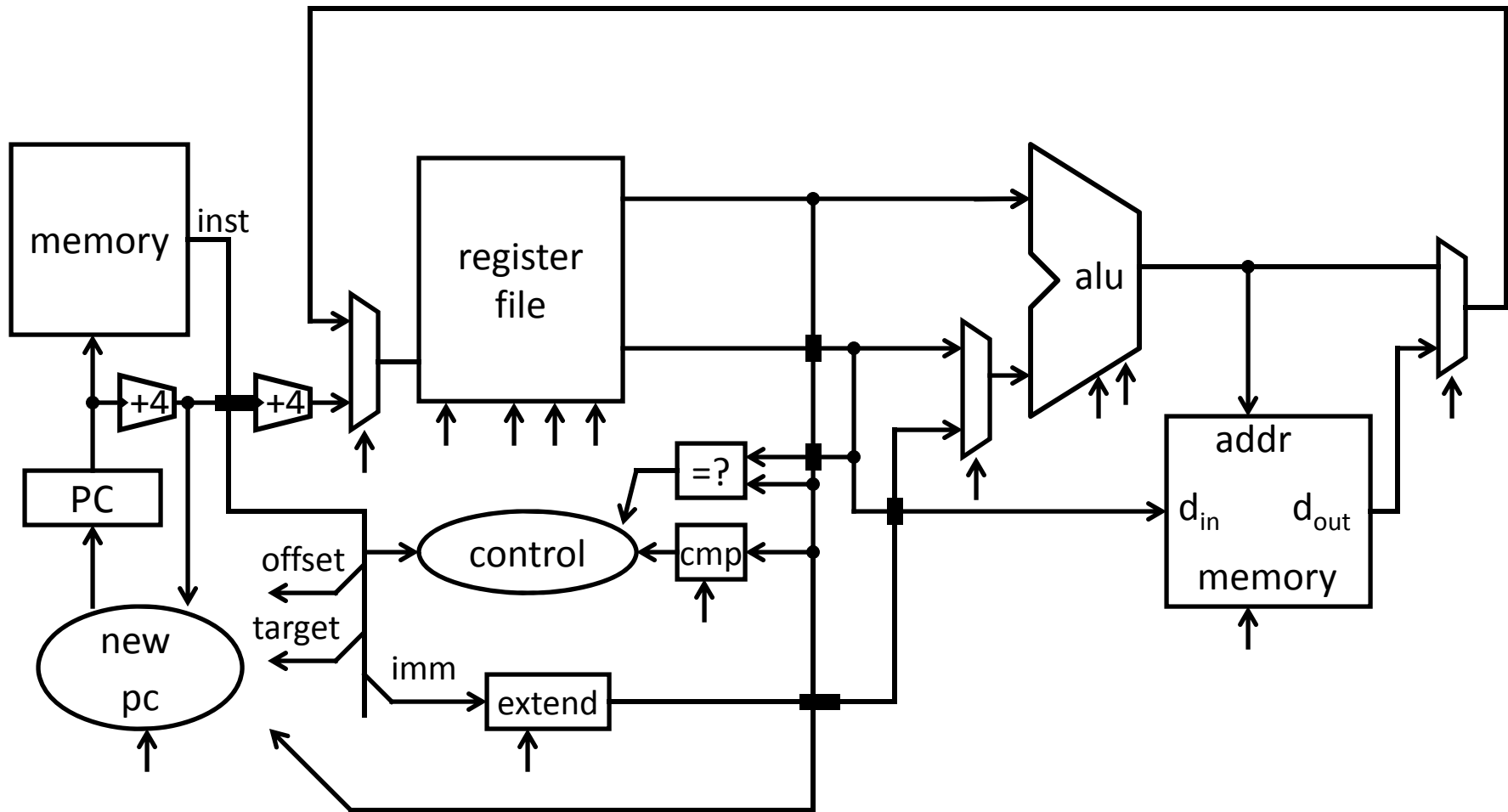
Late Policy

- Each person has a total of **four** “slip days”
- Max of **two** slip days for any individual assignment
- Slip days deducted first for *any* late assignment, cannot selectively apply slip days
- For projects, slip days are deducted from all partners
- 25% deducted per day late after slip days are exhausted

Regrade policy

- Submit written request to lead PhD TA (Deniz, Vishal, Adam, Praveen), and lead TA will pick a different grader
- Submit another written request, lead TA will regrade directly
- Submit yet another written request for professor to regrade.

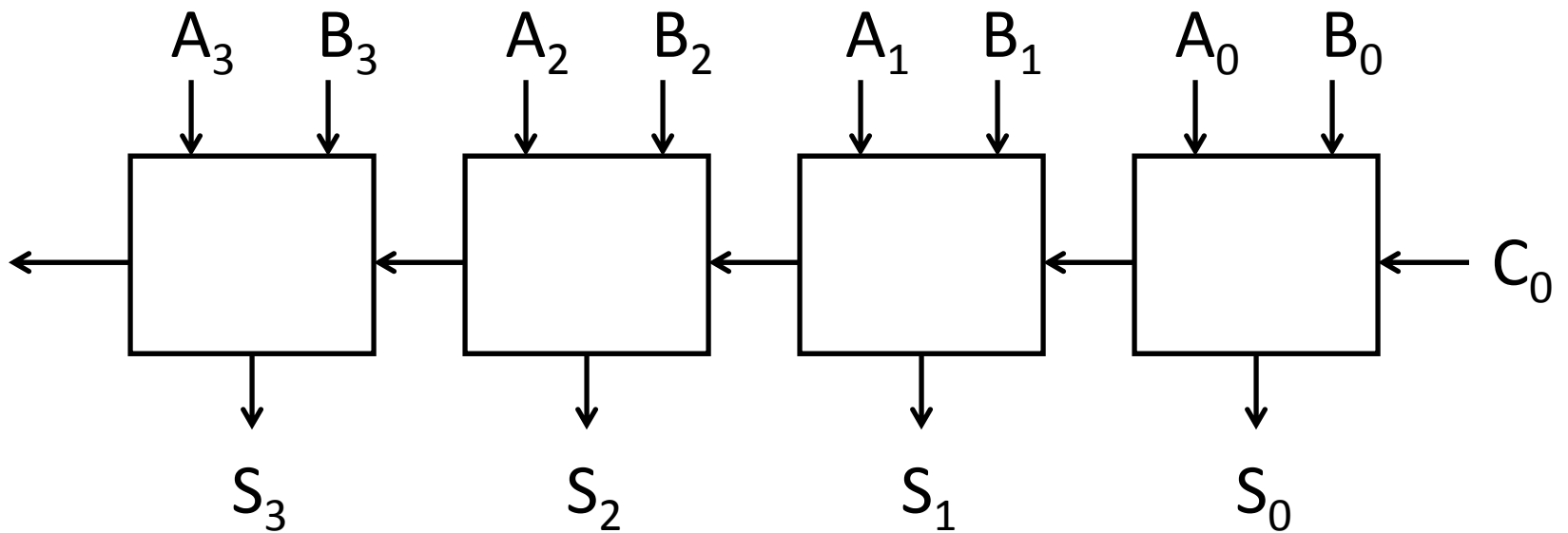
Big Picture: Building a Processor



A Single cycle processor

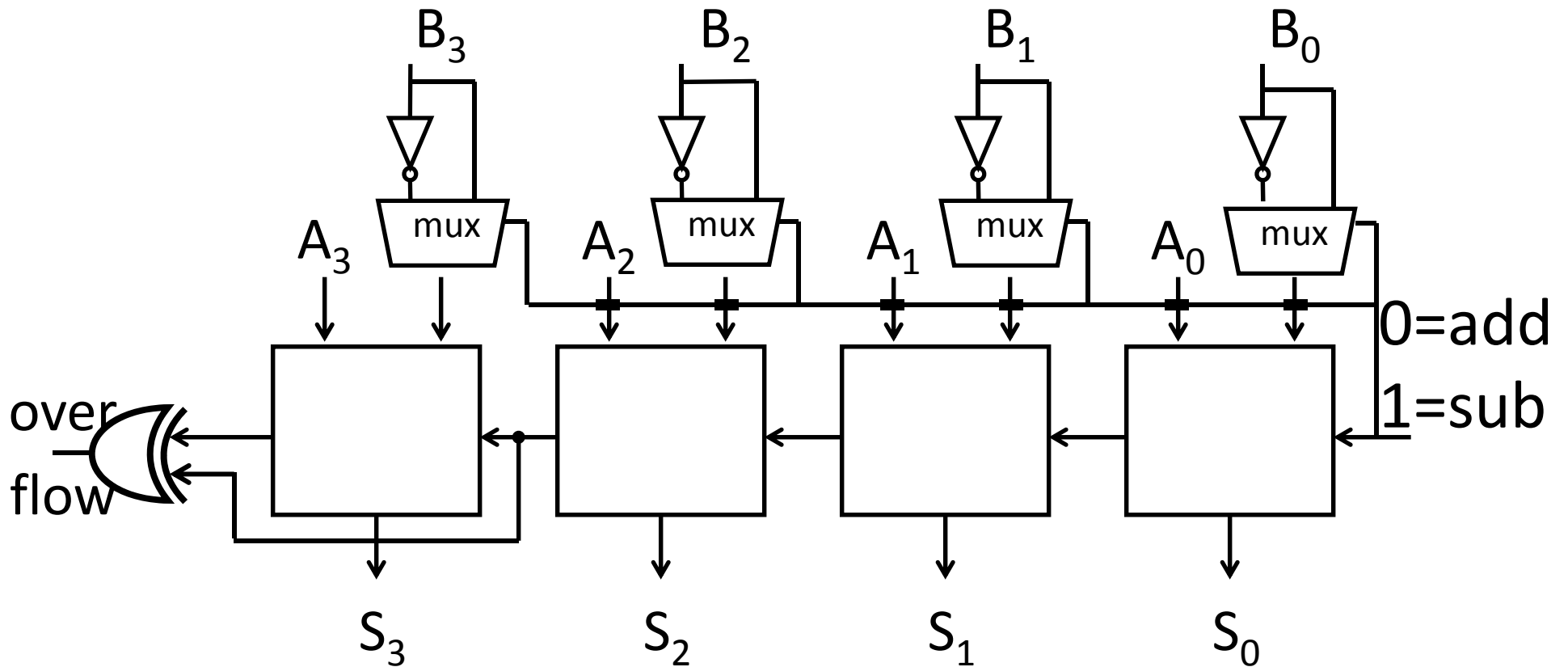
Review

- We can generalize 1-bit Full Adders to 32 bits, 64 bits ...

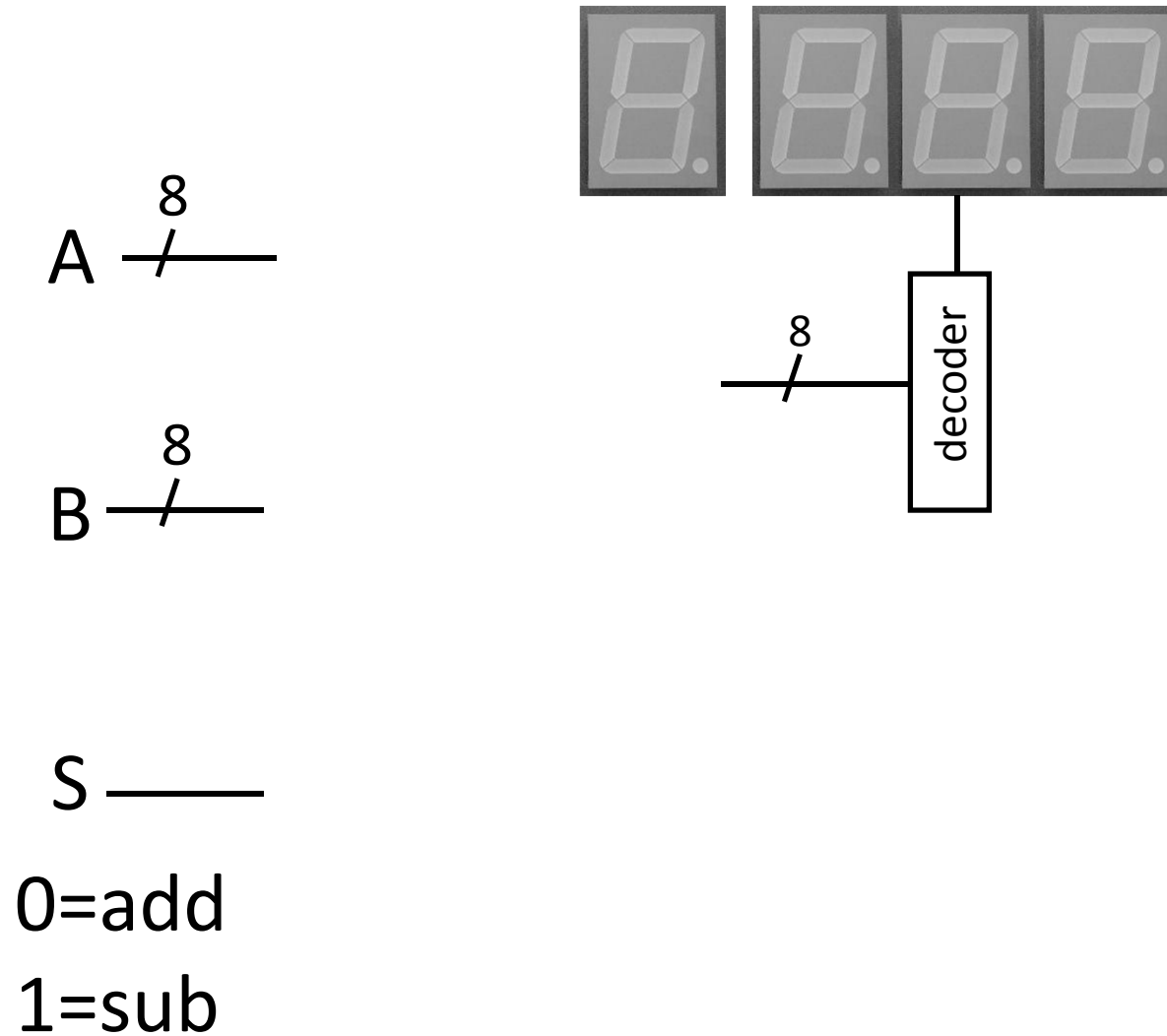


Review

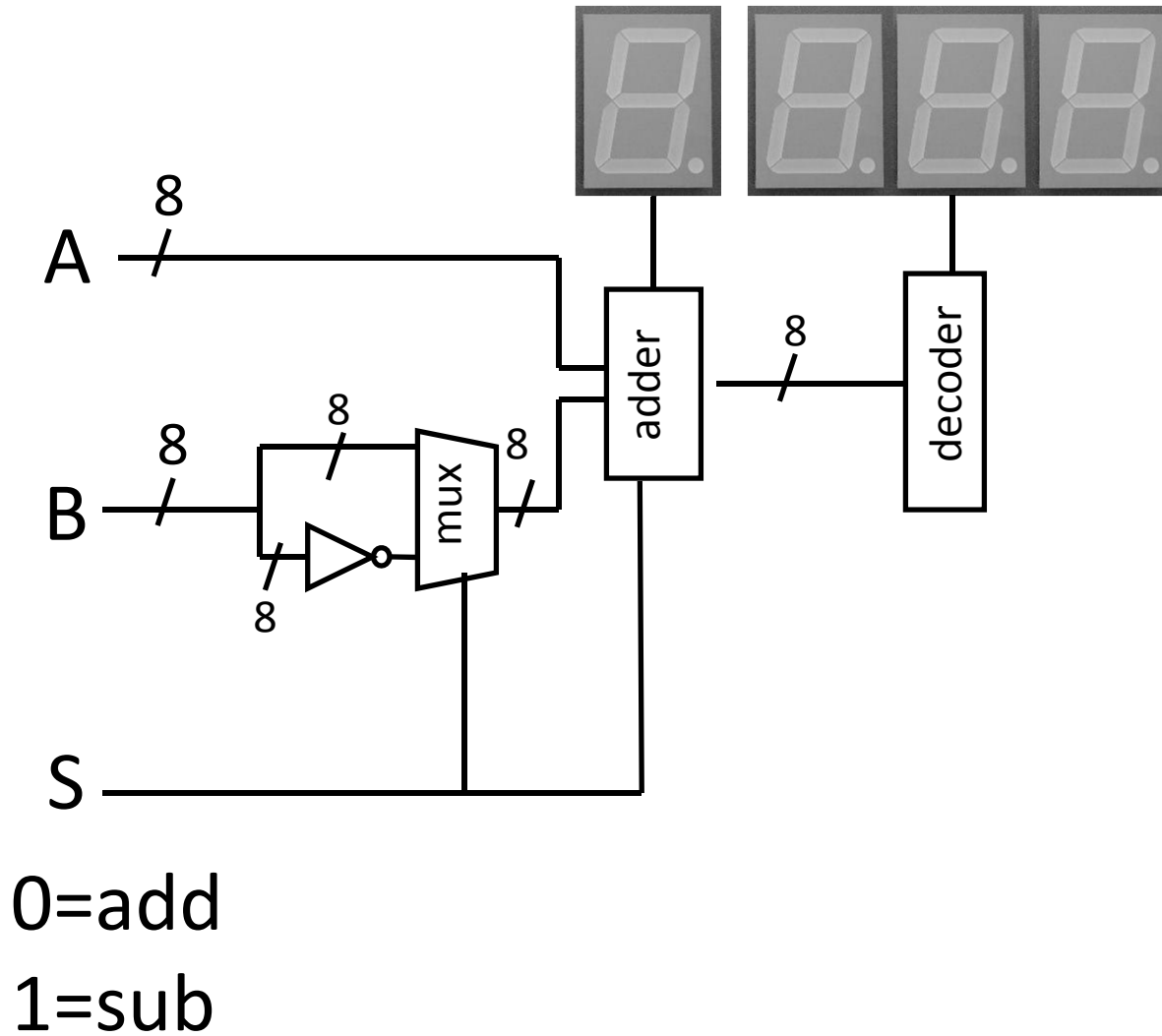
- We can generalize 1-bit Full Adders to 32 bits, 64 bits ...



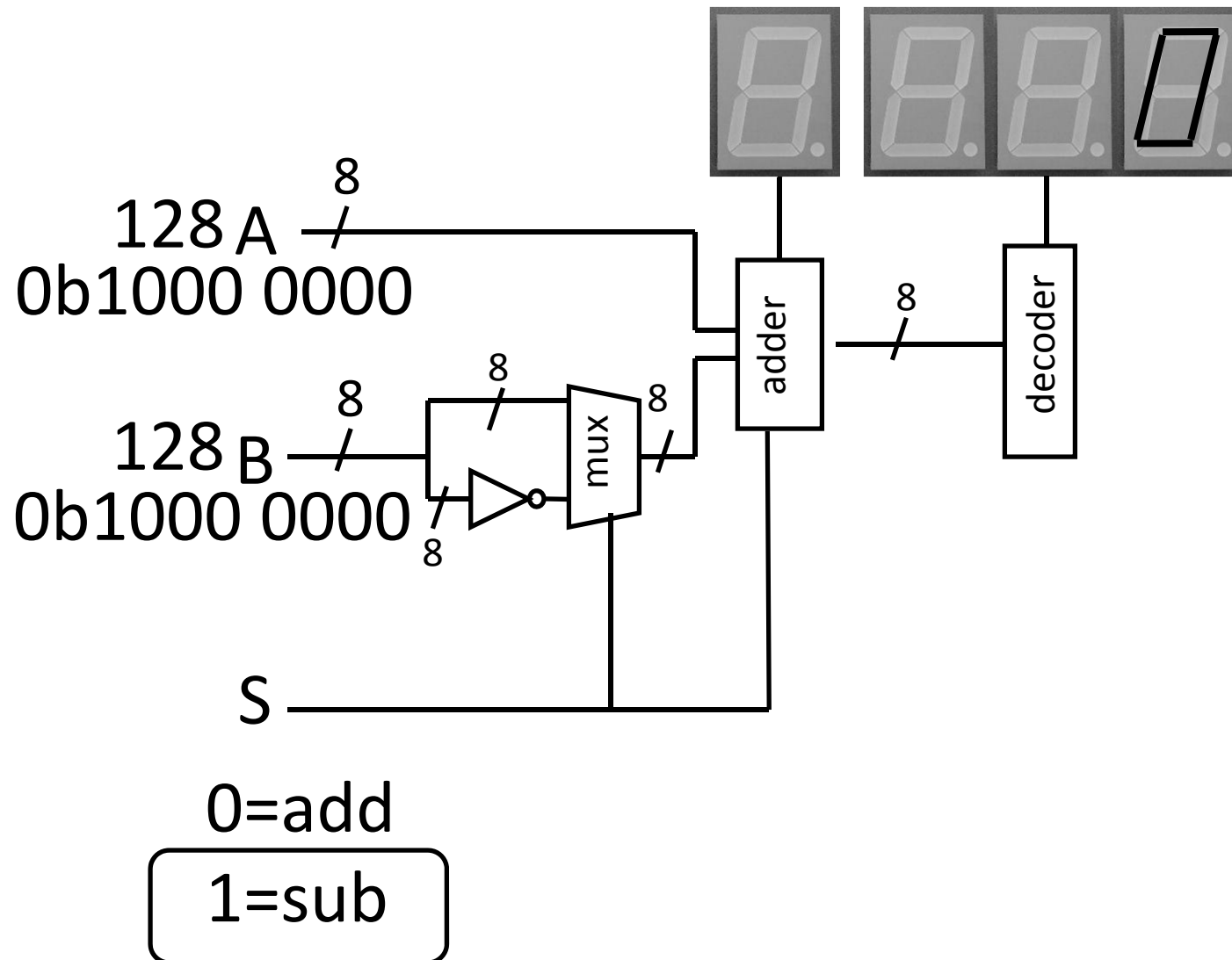
Example: A Calculator



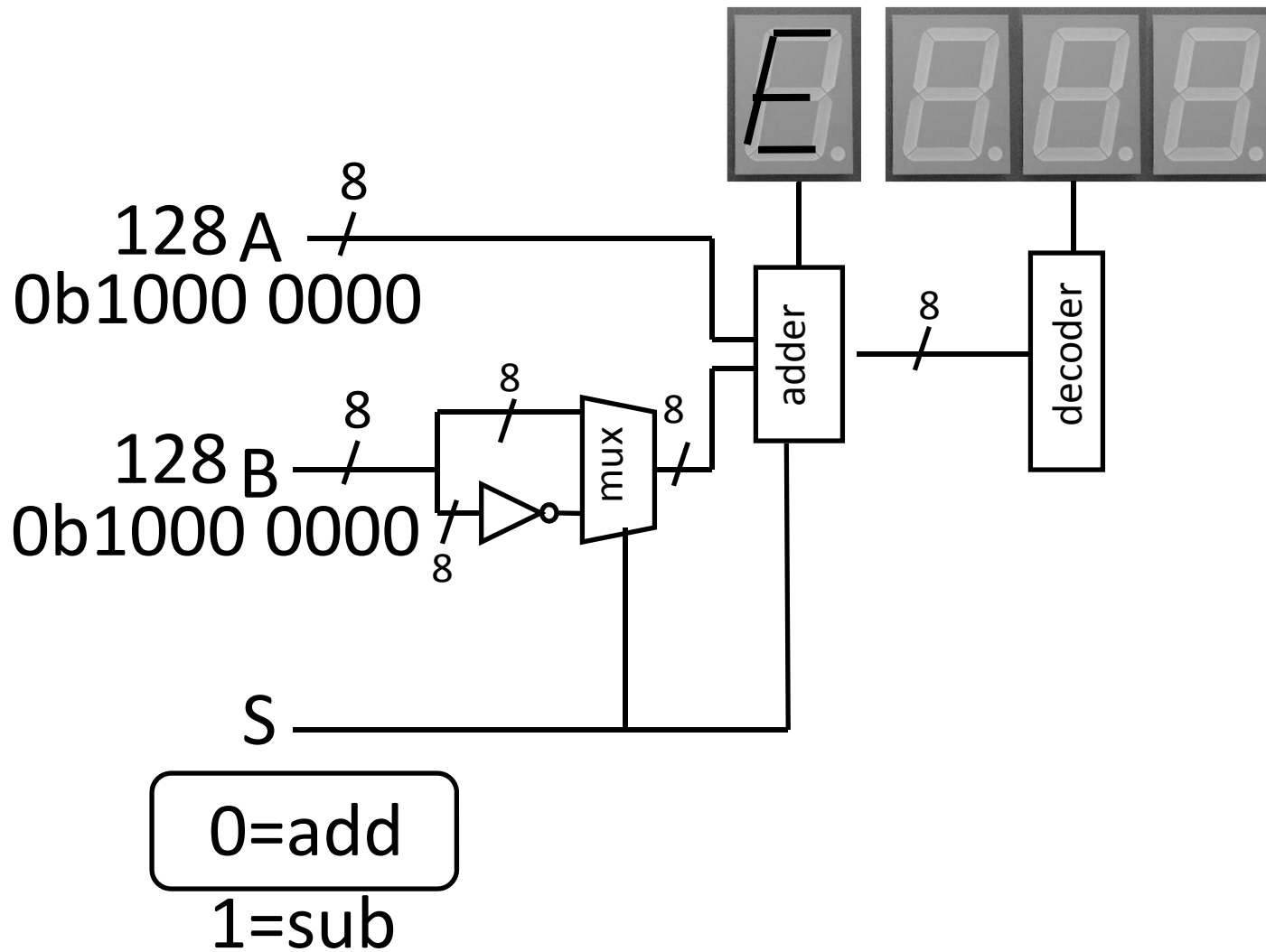
Example: A Calculator



Example: A Calculator

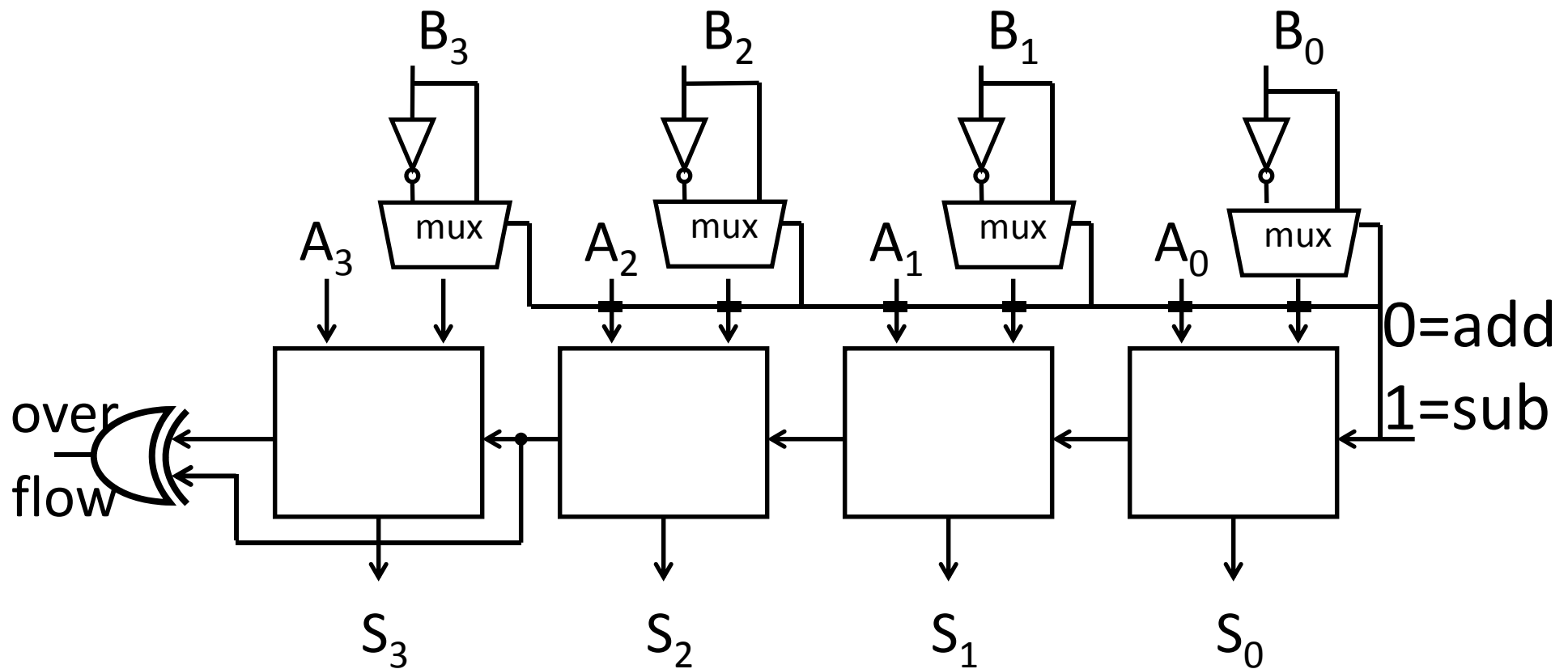


Example: A Calculator



Review: Efficiency and Generality

- We can generalize 1-bit Full Adders to 32 bits, 64 bits ...
- How long does it take to compute a result?

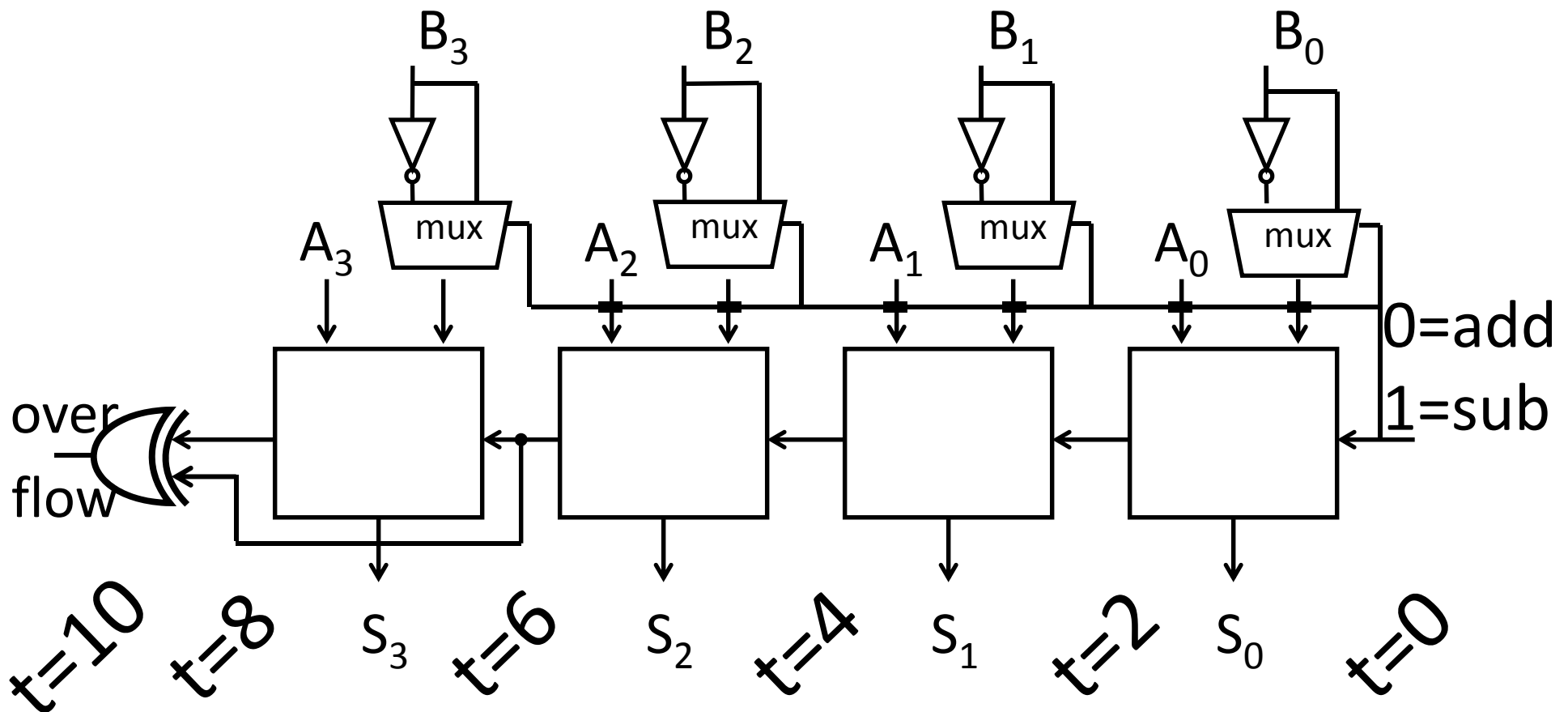


Review: Efficiency and Generality

- We can generalize 1-bit Full Adders to 32 bits, 64 bits ...
 - How long does it take to compute a result?
-
- A) 2 ns
 - B) 2 gate delays
 - C) 10 ns
 - D) 10 gate delays
 - E) 8 gate delays

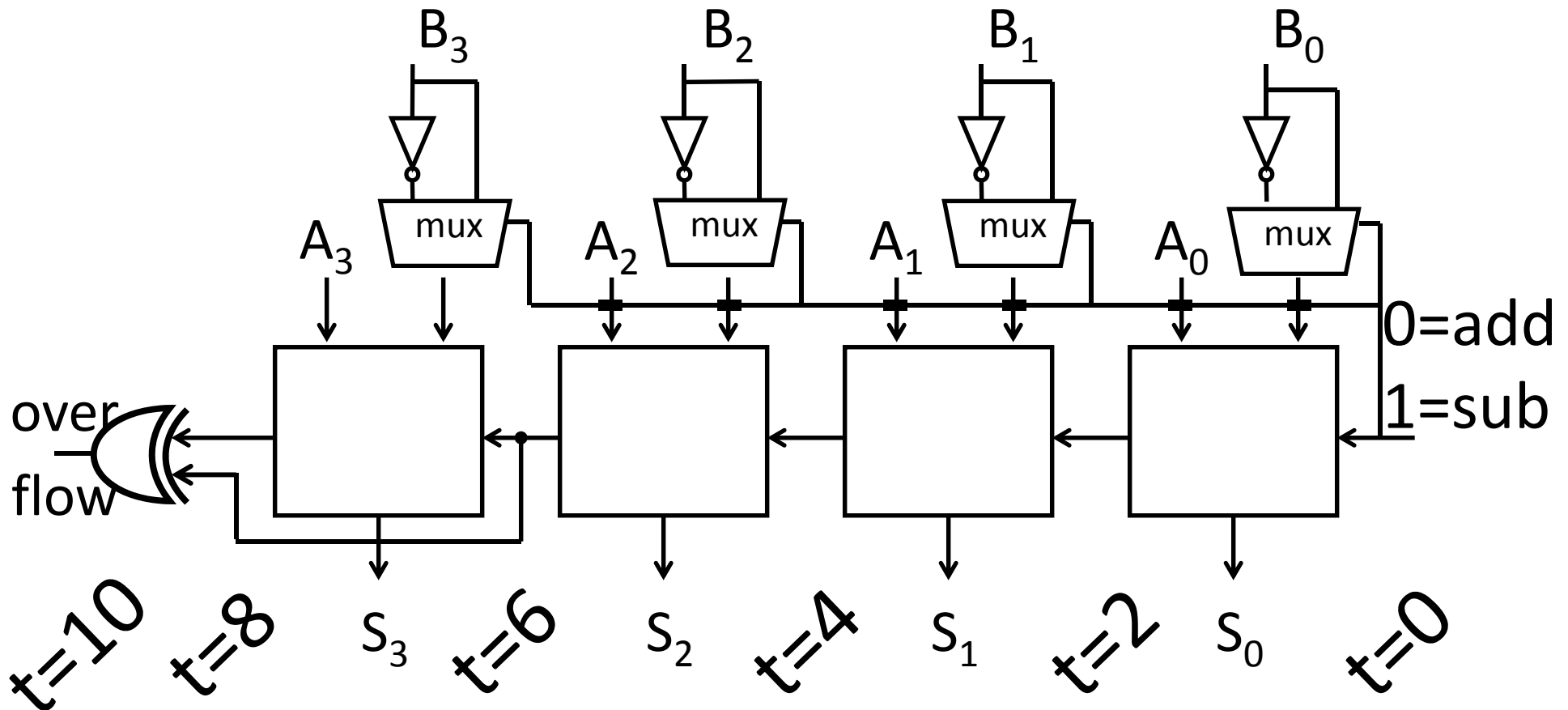
Review: Efficiency and Generality

- We can generalize 1-bit Full Adders to 32 bits, 64 bits ...
- How long does it take to compute a result?



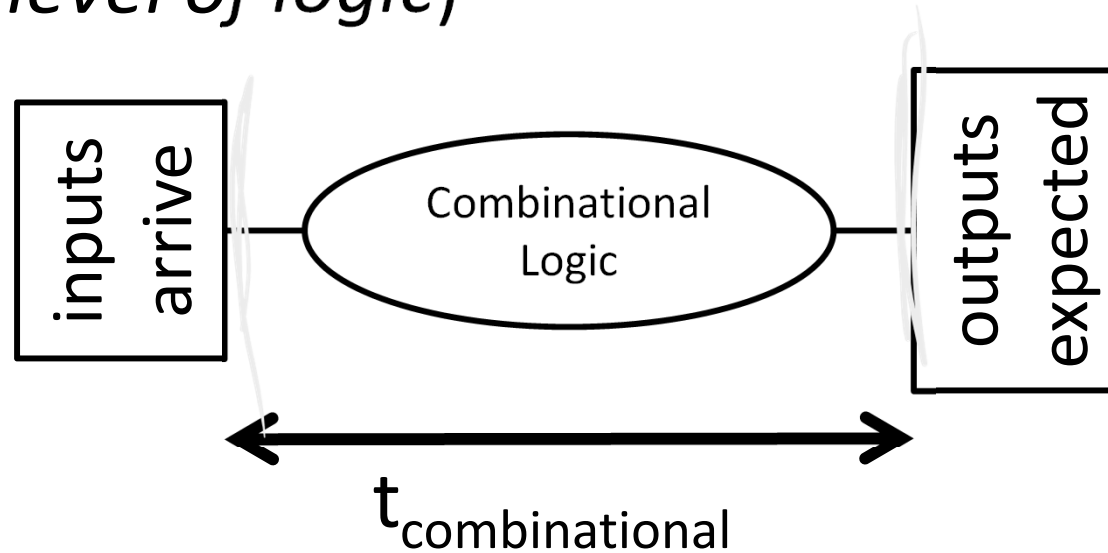
Review: Efficiency and Generality

- We can generalize 1-bit Full Adders to 32 bits, 64 bits ...
- How long does it take to compute a result?
- Can we **store** the result?

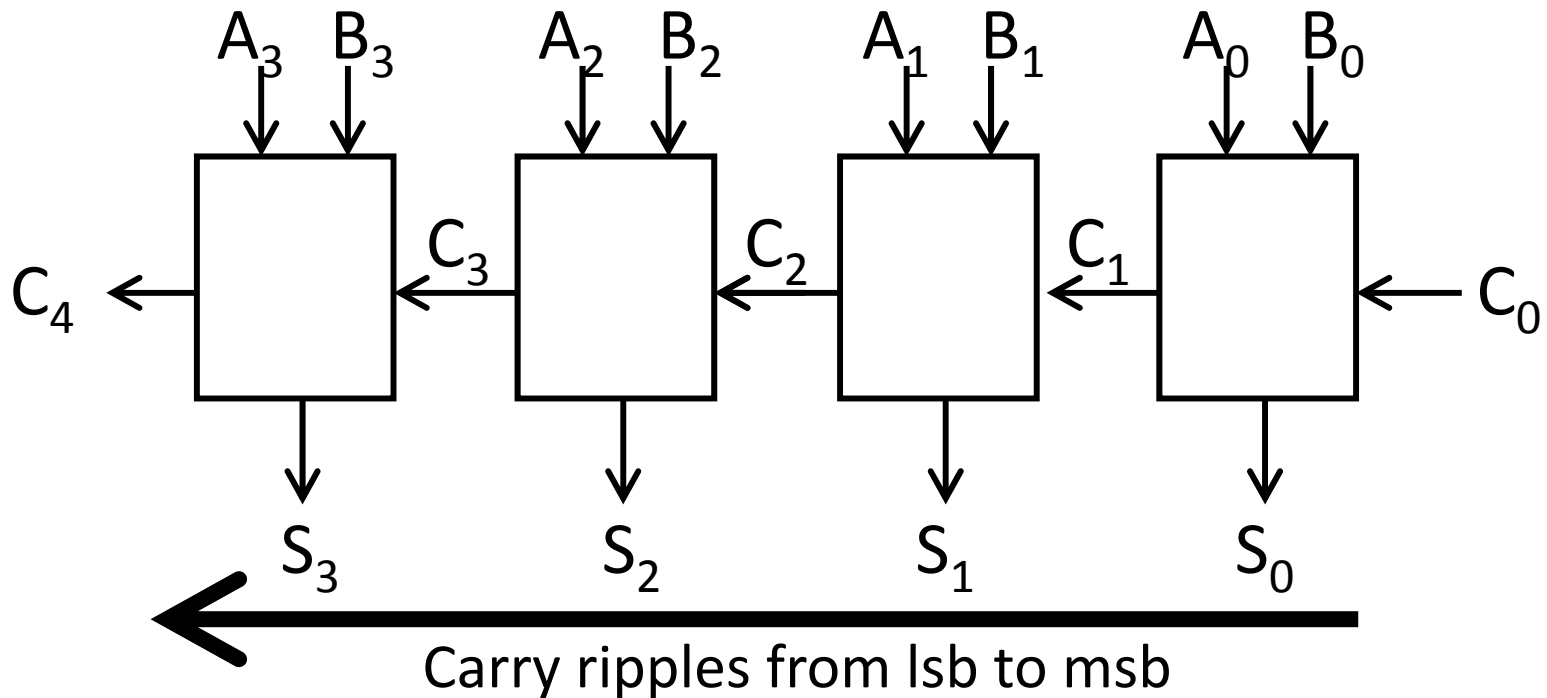


Performance

Speed of a circuit is affected by the number of gates in series (on the *critical path* or the *deepest level of logic*)



4-bit Ripple Carry Adder

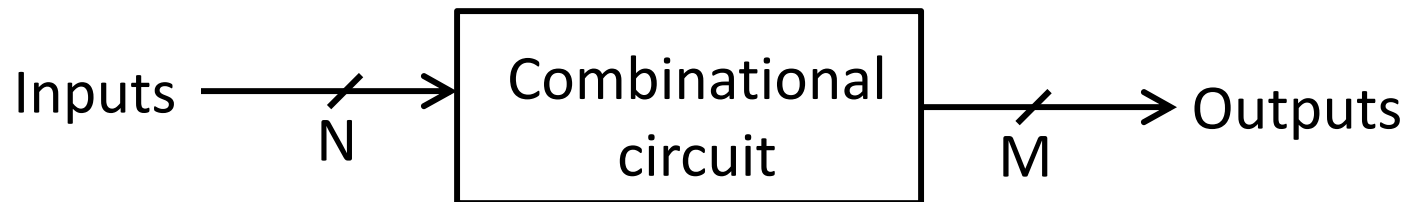


- First full adder, 2 gate delay
- Second full adder, 2 gate delay
- ...

Stateful Components

Until now is combinational logic

- Output is computed when inputs are present
- System has no internal state
- Nothing computed in the present can depend on what happened in the past!



Need a way to record data

Need a way to build stateful circuits

Need a state-holding device

Finite State Machines

Goals for Today

State

- How do we store *one* bit?
- Attempts at storing (and changing) one bit
 - Set-Reset Latch
 - D Latch
 - D Flip-Flops
 - Master-Slave Flip-Flops
- Register: storing more than one bit, N-bits

Basic Building Blocks

- Decoders and Encoders

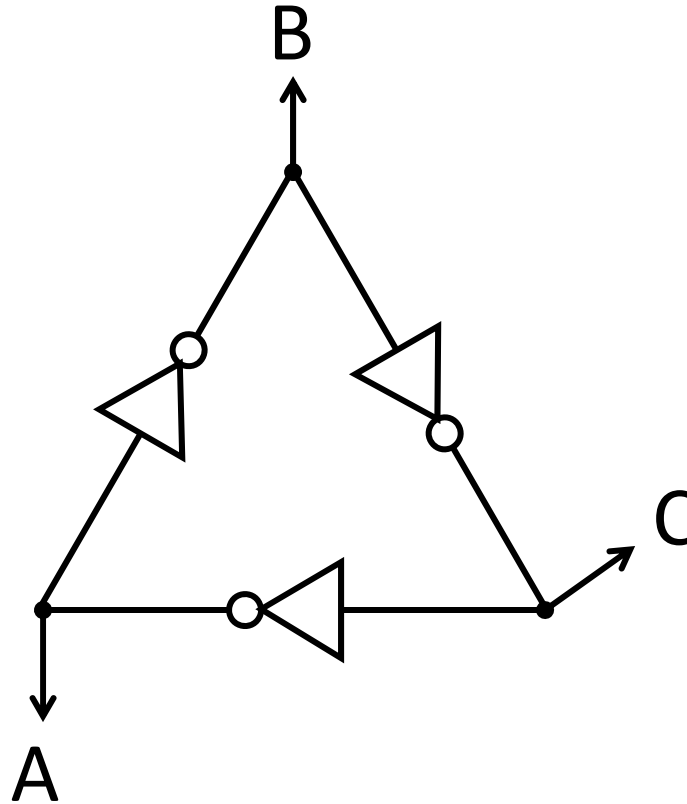
Finite State Machines (FSM)

- How do we design logic circuits with state?
- Types of FSMs: Mealy and Moore Machines
- Examples: Serial Adder and a Digital Door Lock

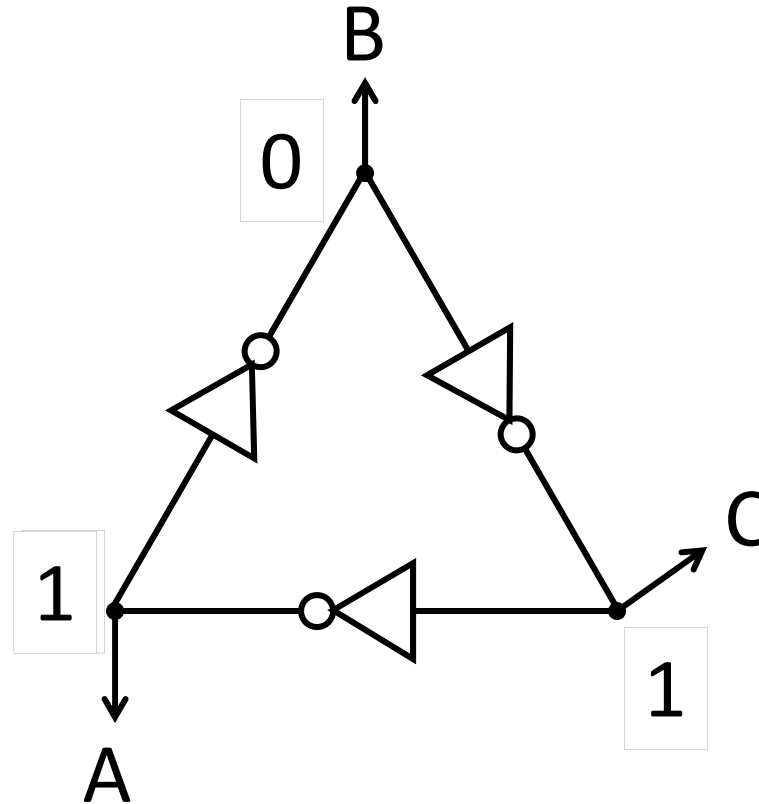
Goal

How do we store store *one* bit?

First Attempt: Unstable Devices



First Attempt: Unstable Devices

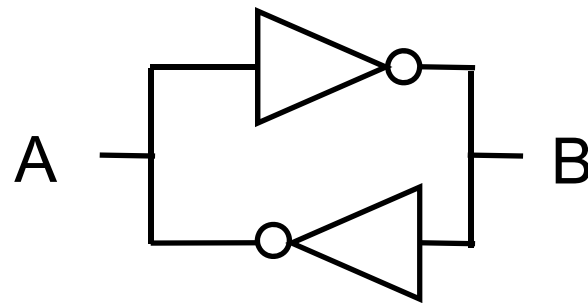


Does not work!

- Unstable
- Oscillates wildly!

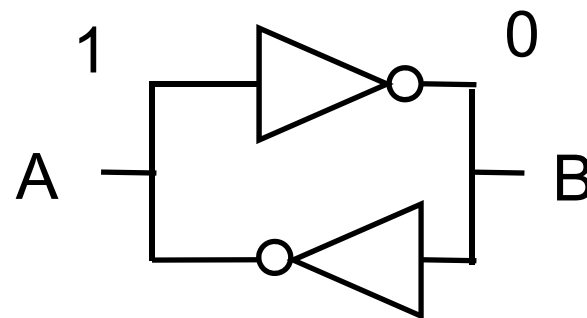
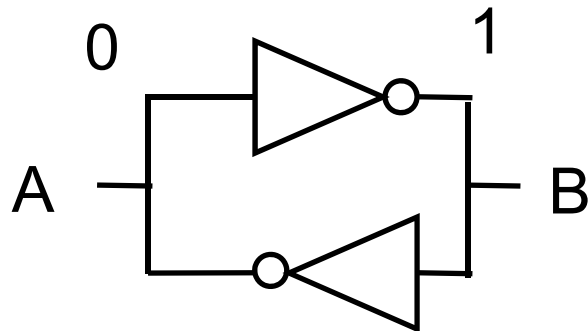
Second Attempt: Bistable Devices

- Stable and unstable equilibria?



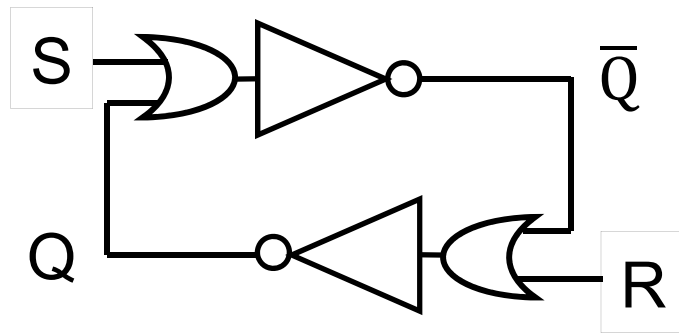
A Simple Device

In stable state, $\bar{A} = B$

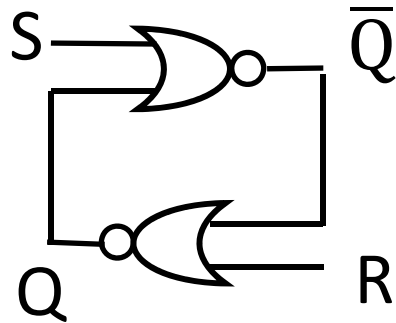


How do we change the state?

Third Attempt: Set-Reset Latch



Third Attempt: Set-Reset Latch



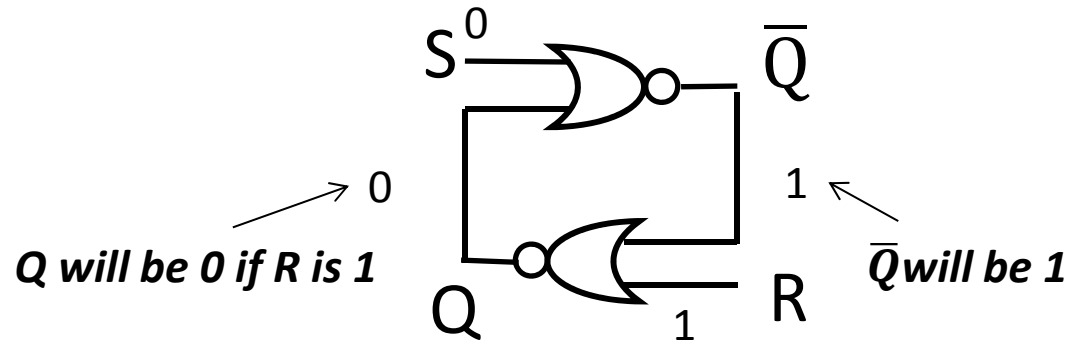
| A | B | OR | NOR |
|---|---|----|-----|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |

| S | R | Q | \bar{Q} |
|---|---|---|-----------|
| 0 | 0 | | |
| 0 | 1 | | |
| 1 | 0 | | |
| 1 | 1 | | |

Set-Reset (S-R) Latch

Stores a value Q and its complement

Third Attempt: Set-Reset Latch



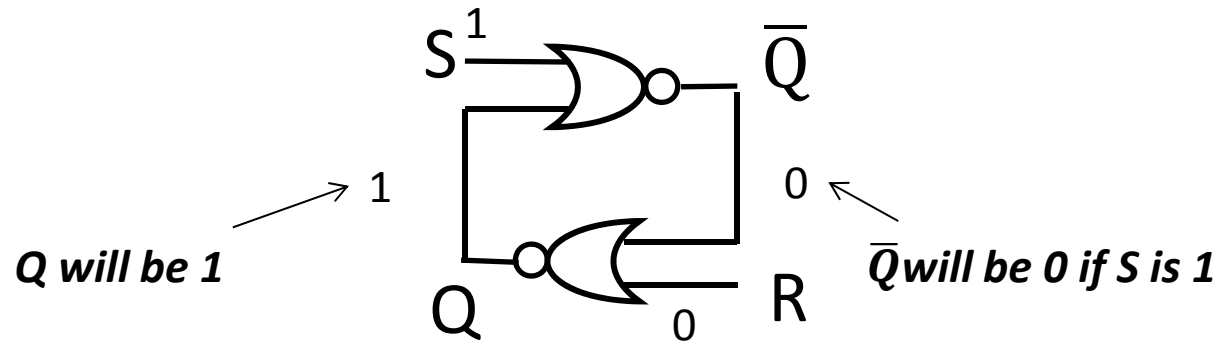
| A | B | OR | NOR |
|---|---|----|-----|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |

| S | R | Q | \bar{Q} |
|---|---|---|-----------|
| 0 | 0 | | |
| 0 | 1 | 0 | 1 |
| 1 | 0 | | |
| 1 | 1 | | |

Set-Reset (S-R) Latch

Stores a value Q and its complement

Third Attempt: Set-Reset Latch



| A | B | OR | NOR |
|---|---|----|-----|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |

| S | R | Q | \bar{Q} |
|---|---|---|-----------|
| 0 | 0 | | |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | | |

Set-Reset (S-R) Latch

Stores a value Q and its complement

What are the values for Q and \bar{Q} ?

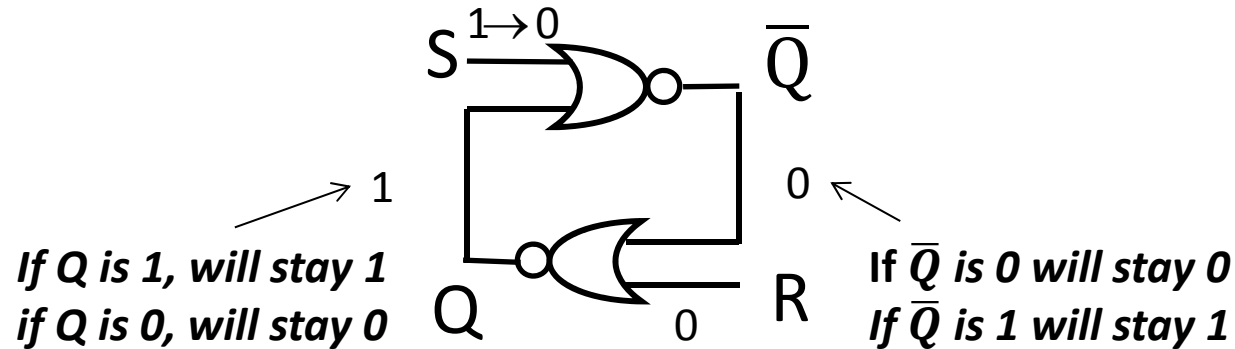
a) 0 and 0

b) 0 and 1

c) 1 and 0

d) 1 and 1

Third Attempt: Set-Reset Latch



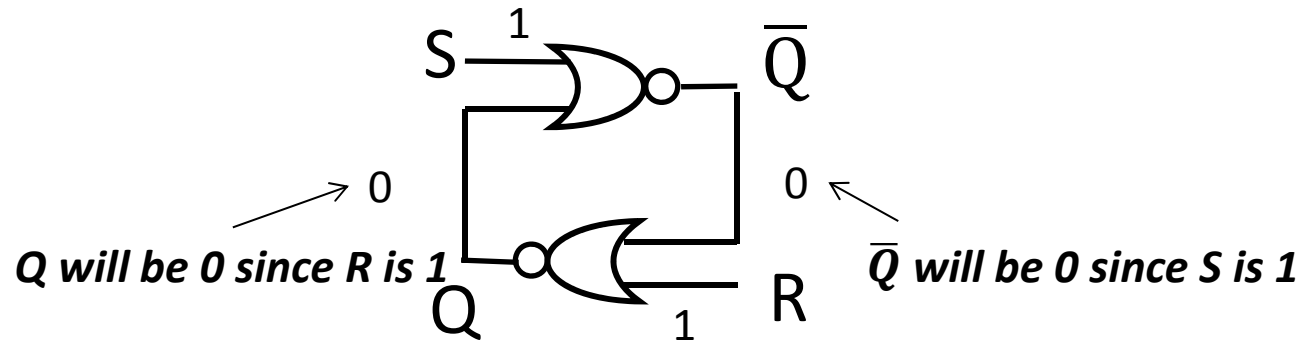
| A | B | OR | NOR |
|---|---|----|-----|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |

| S | R | Q | \bar{Q} |
|---|---|---|-----------|
| 0 | 0 | Q | \bar{Q} |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | | |

Set-Reset (S-R) Latch

Stores a value Q and its complement

Third Attempt: Set-Reset Latch



| A | B | OR | NOR |
|---|---|----|-----|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |

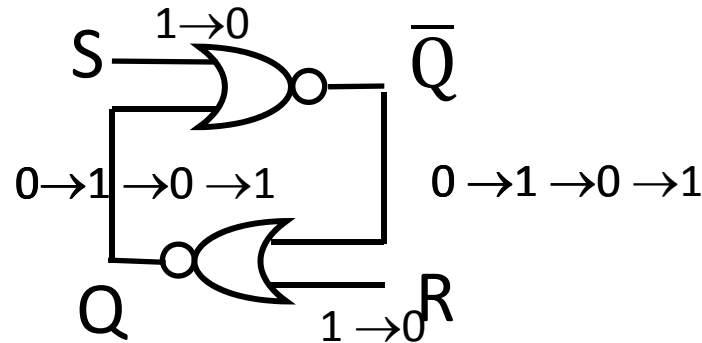
| S | R | Q | \bar{Q} |
|---|---|---|-----------|
| 0 | 0 | Q | \bar{Q} |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | ? | ? |

Set-Reset (S-R) Latch

Stores a value Q and its complement

What happens when S,R changes from 1,1 to 0,0?

Third Attempt: Set-Reset Latch



| A | B | OR | NOR |
|---|---|----|-----|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |

| S | R | Q | \bar{Q} |
|---|---|-----------|-----------|
| 0 | 0 | Q | \bar{Q} |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | forbidden | |

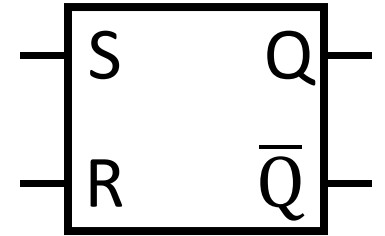
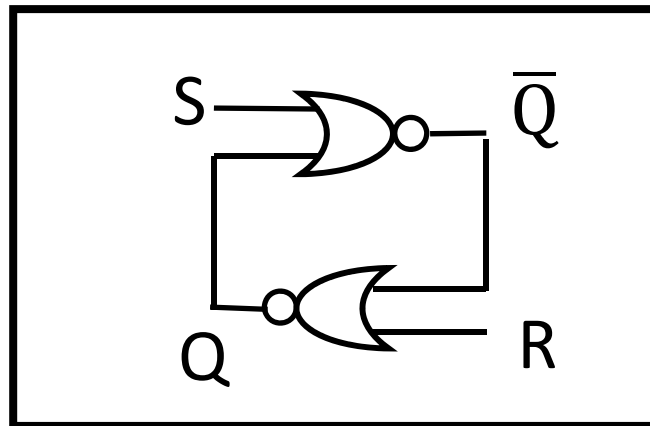
Set-Reset (S-R) Latch

Stores a value Q and its complement

What happens when S,R changes from 1,1 to 0,0?

Q and \bar{Q} become unstable and will oscillate wildly between values 0,0 to 1,1 to 0,0 to 1,1 ...

Third Attempt: Set-Reset Latch



| S | R | Q | Q̄ | |
|---|---|-----------|----|-------|
| 0 | 0 | Q | Q̄ | hold |
| 0 | 1 | 0 | 1 | reset |
| 1 | 0 | 1 | 0 | set |
| 1 | 1 | forbidden | | |

Set-Reset (S-R) Latch

Stores a value Q and its complement

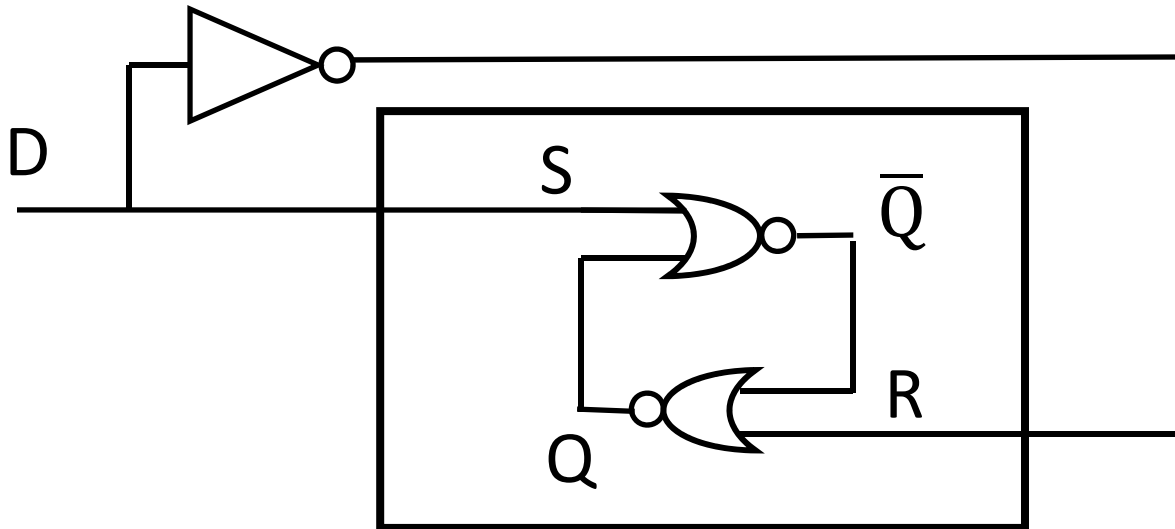
Takeaway

Set-Reset (SR) Latch can store one bit and we can change the value of the stored bit. But, SR Latch has a forbidden state.

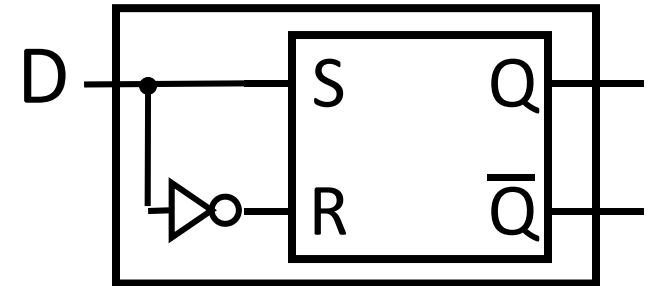
Next Goal

How do we avoid the forbidden state of S-R Latch?

Fourth Attempt: (Unclocked) D Latch



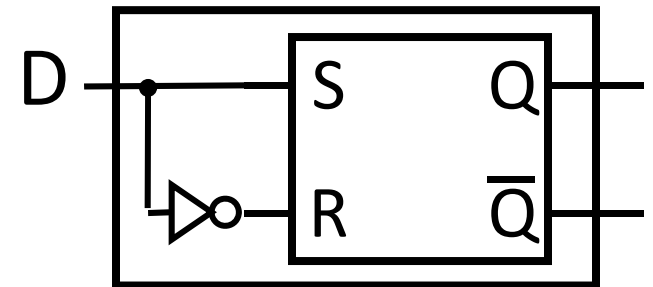
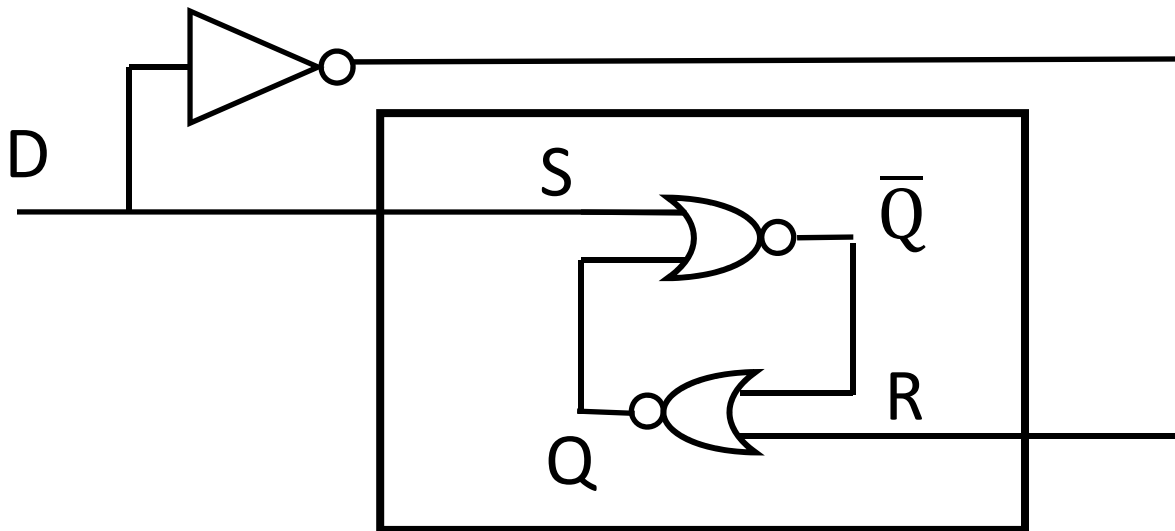
Fill in the truth table?



| D | Q | \bar{Q} |
|---|---|-----------|
| 0 | | |
| 1 | | |

| A | B | OR | NOR |
|---|---|----|-----|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |

Fourth Attempt: (Unclocked) D Latch



Fill in the truth table?

Data (D) Latch

- Easier to use than an SR latch
- No possibility of entering an undefined state

When D changes, Q changes

- ... immediately (...after a delay of 2 Ors and 2 NOTs)

Need to control when the output changes

| D | Q | \bar{Q} |
|---|---|-----------|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

| A | B | OR | NOR |
|---|---|----|-----|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |

Takeaway

Set-Reset (SR) Latch can store one bit and we can change the value of the stored bit. But, SR Latch has a forbidden state.

(Unclocked) D Latch can store and change a bit like an SR Latch while avoiding the forbidden state.

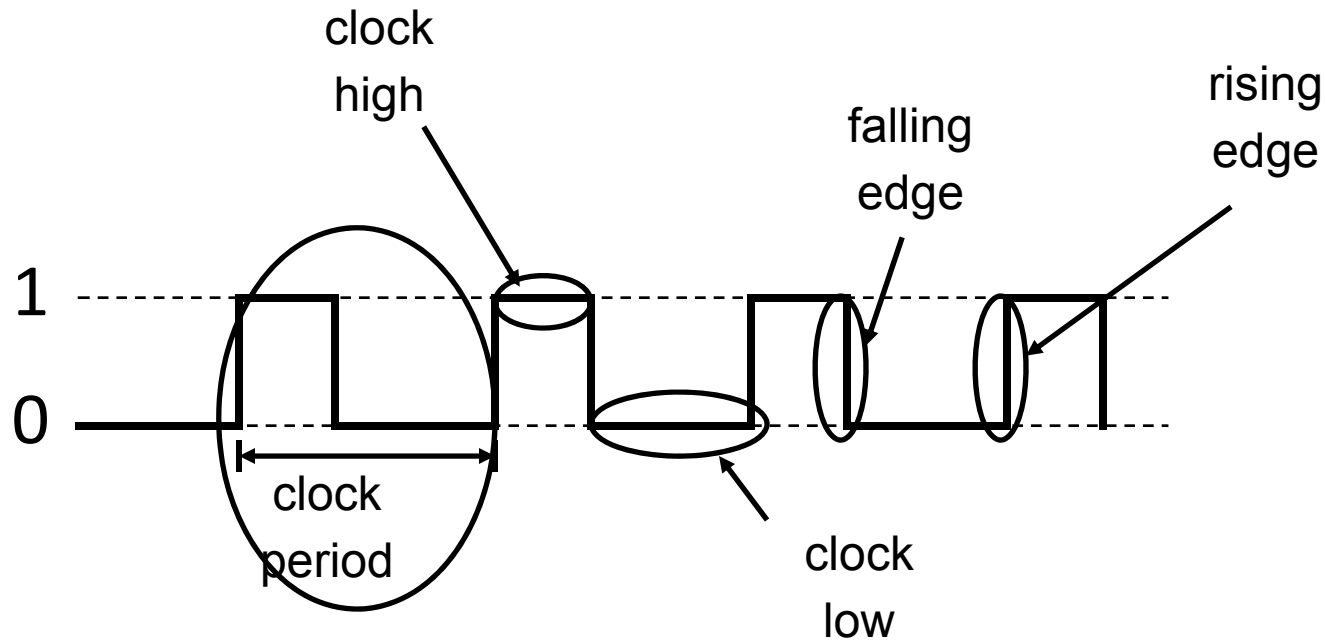
Next Goal

How do we coordinate state changes to a D Latch?

Clocks

Clock helps coordinate state changes

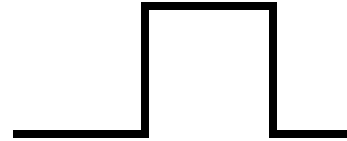
- Usually generated by an oscillating crystal
- Fixed period; frequency = $1/\text{period}$



Clock Disciplines

Level sensitive

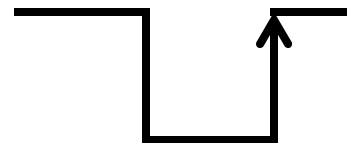
- State changes when clock is high (or low)



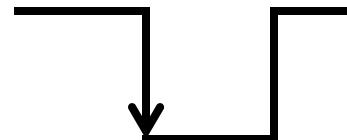
Edge triggered

- State changes at clock edge

positive edge-triggered



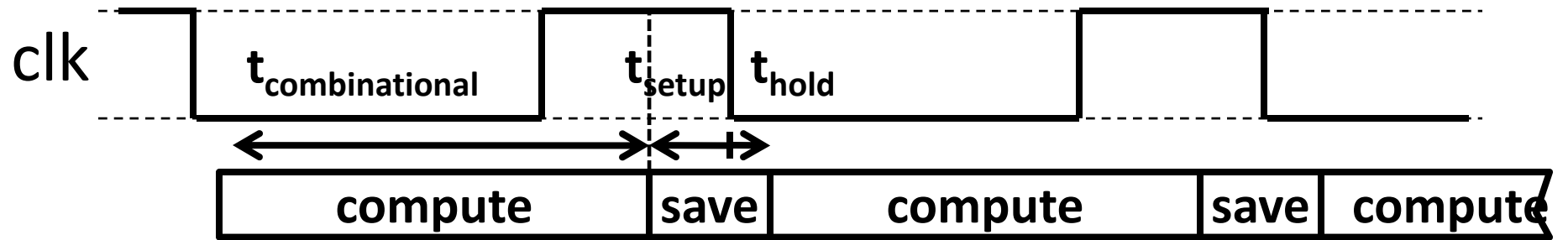
negative edge-triggered



Clock Methodology

Clock Methodology

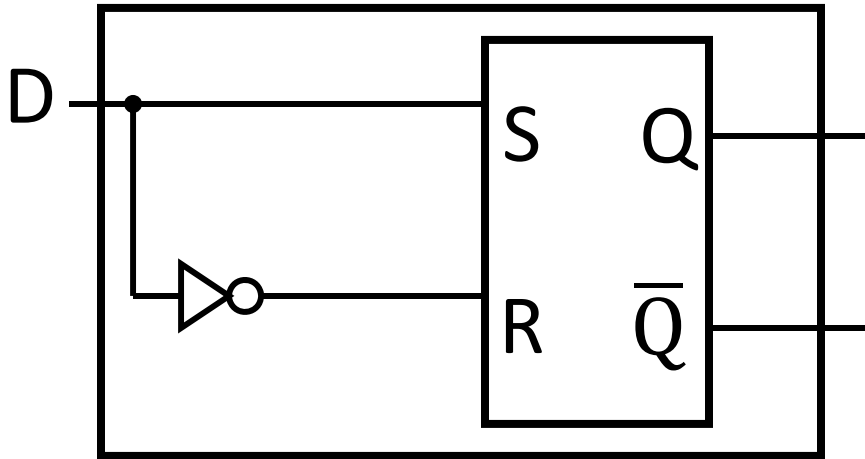
- Negative edge, synchronous



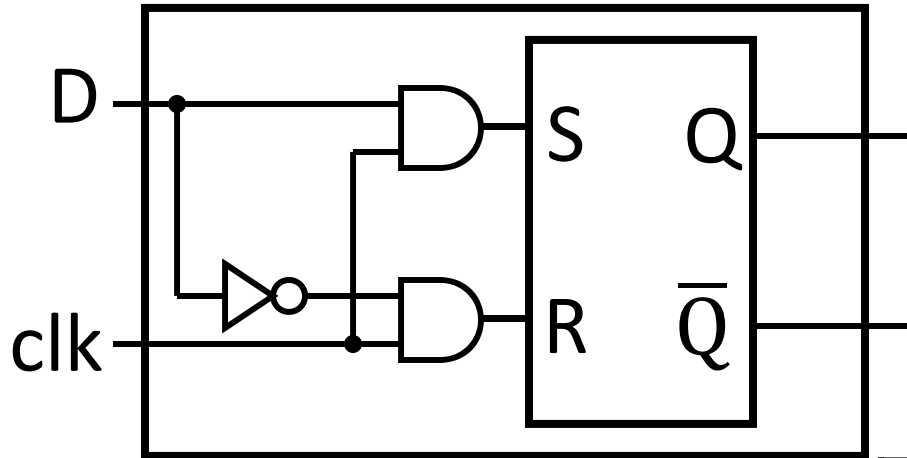
– Edge-Triggered: Signals must be stable near falling clock edge

- Positive edge synchronous

Fifth Attempt: D Latch with Clock



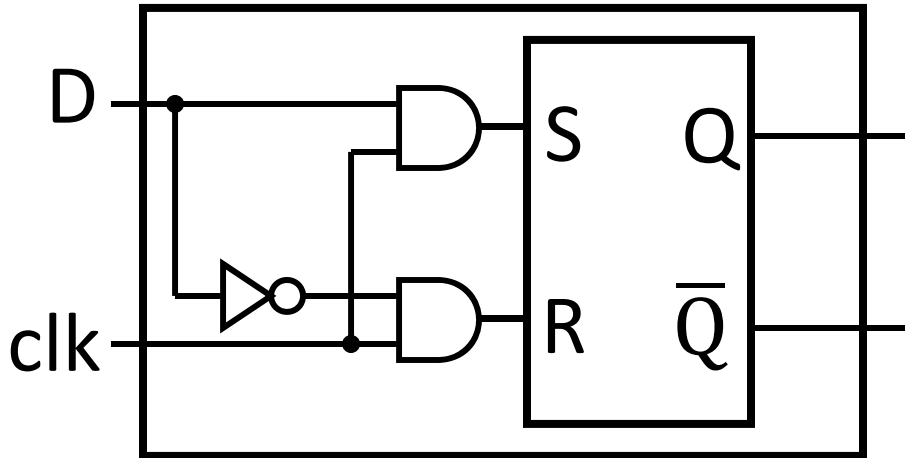
Fifth Attempt: D Latch with Clock



Fill in the truth table

| clk | D | Q | \bar{Q} |
|-----|---|---|-----------|
| 0 | 0 | | |
| 0 | 1 | | |
| 1 | 0 | | |
| 1 | 1 | | |

Fifth Attempt: D Latch with Clock



Fill in the truth table

| S | R | Q | \bar{Q} | |
|---|---|-----------|-----------|-------|
| 0 | 0 | Q | \bar{Q} | hold |
| 0 | 1 | 0 | 1 | reset |
| 1 | 0 | 1 | 0 | set |
| 1 | 1 | forbidden | | |

| clk | D | Q | \bar{Q} |
|-----|---|---|-----------|
| 0 | 0 | Q | \bar{Q} |
| 0 | 1 | Q | \bar{Q} |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Fifth Attempt: D Latch with Clock

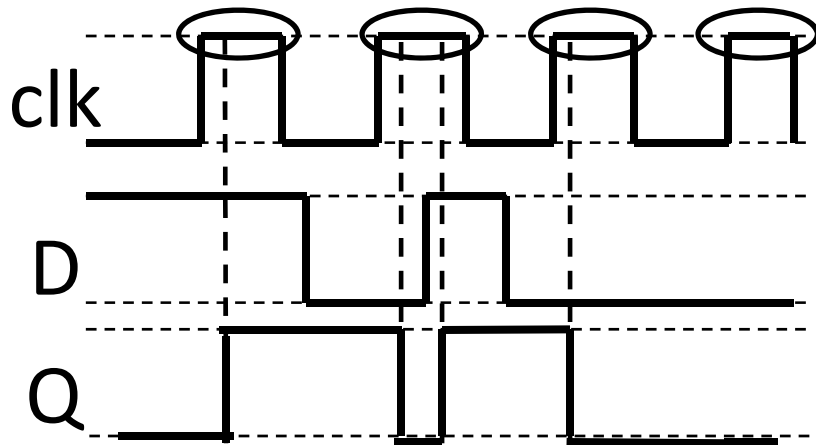
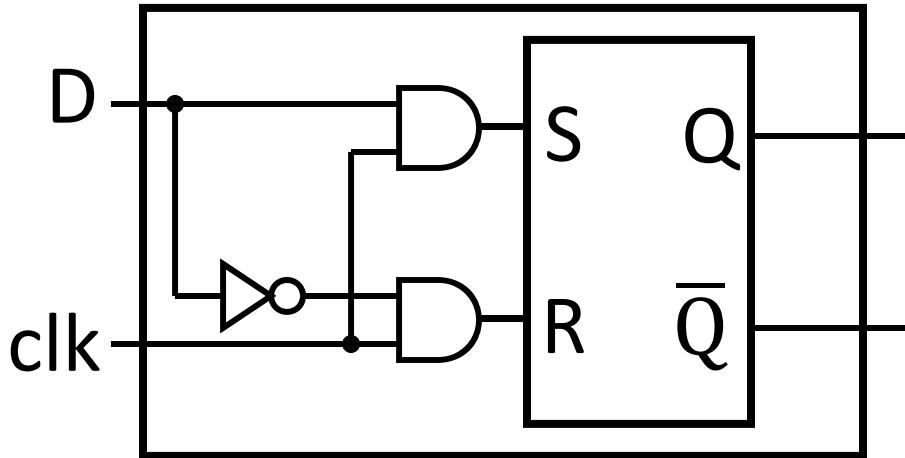
Level Sensitive D Latch

Clock high:

set/reset (according to D)

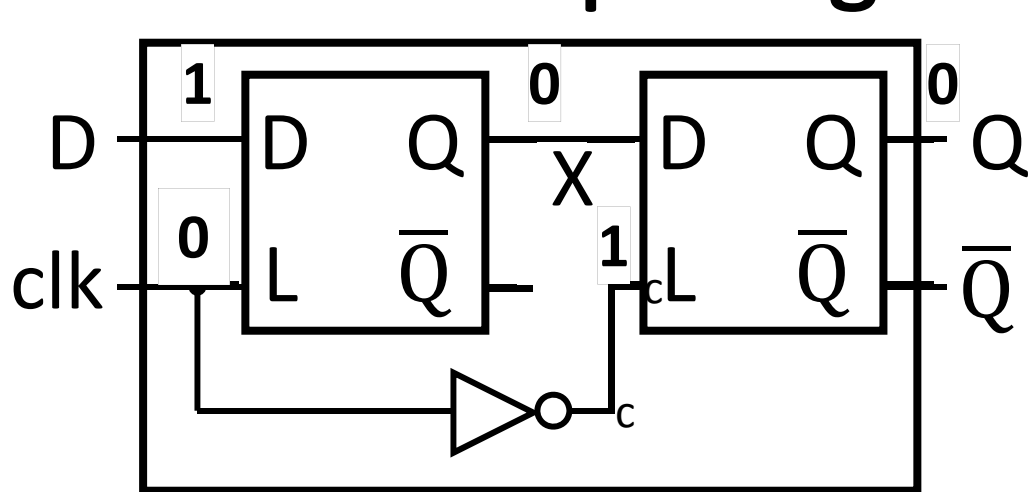
Clock low:

keep state (ignore D)



| clk | D | Q | \bar{Q} |
|-----|---|---|-----------|
| 0 | 0 | Q | \bar{Q} |
| 0 | 1 | Q | \bar{Q} |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

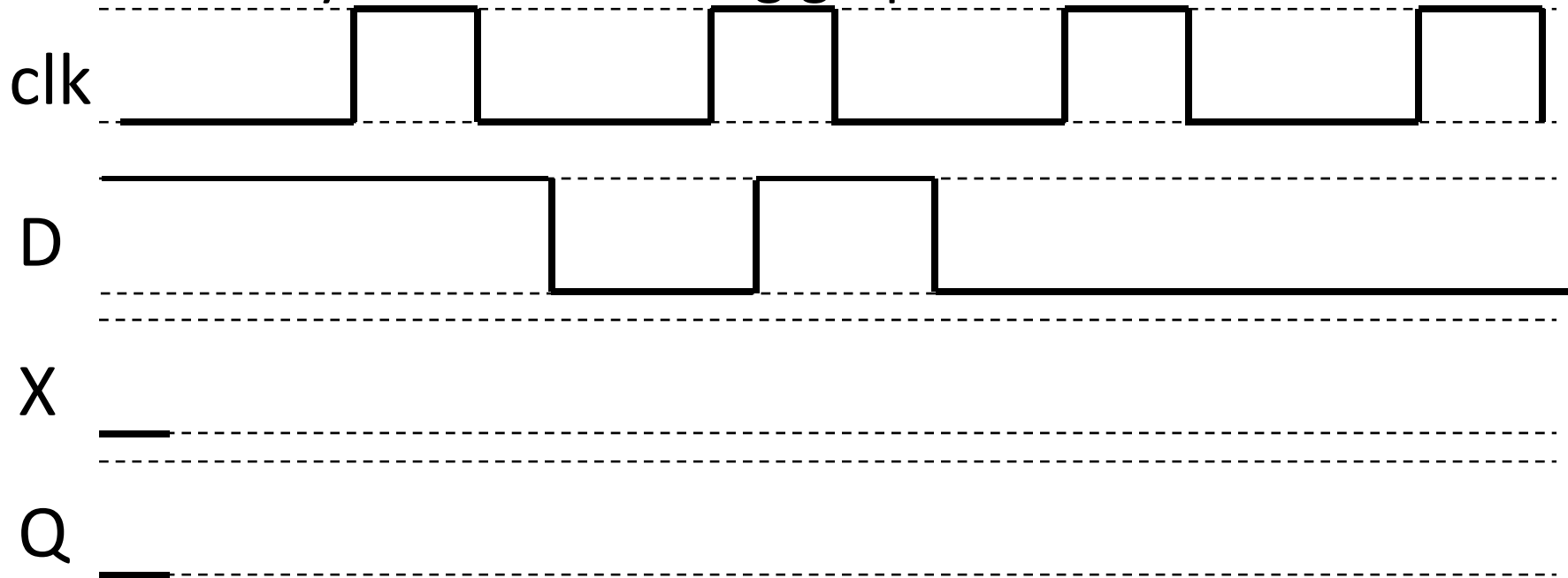
Sixth Attempt: Edge-Triggered D Flip-Flop



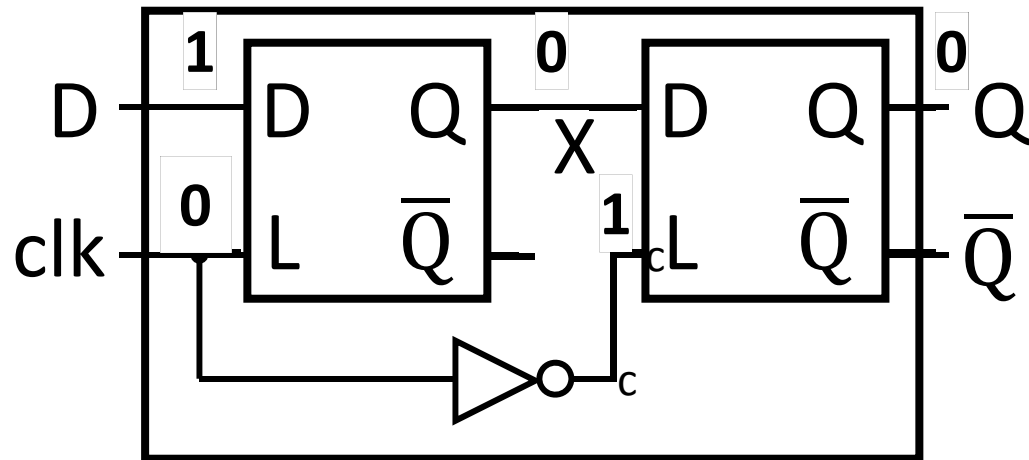
D Flip-Flop

- Edge-Triggered
- Data captured when clock is high
- Output changes only on falling edges

Activity#1: Fill in timing graph and values for X and Q

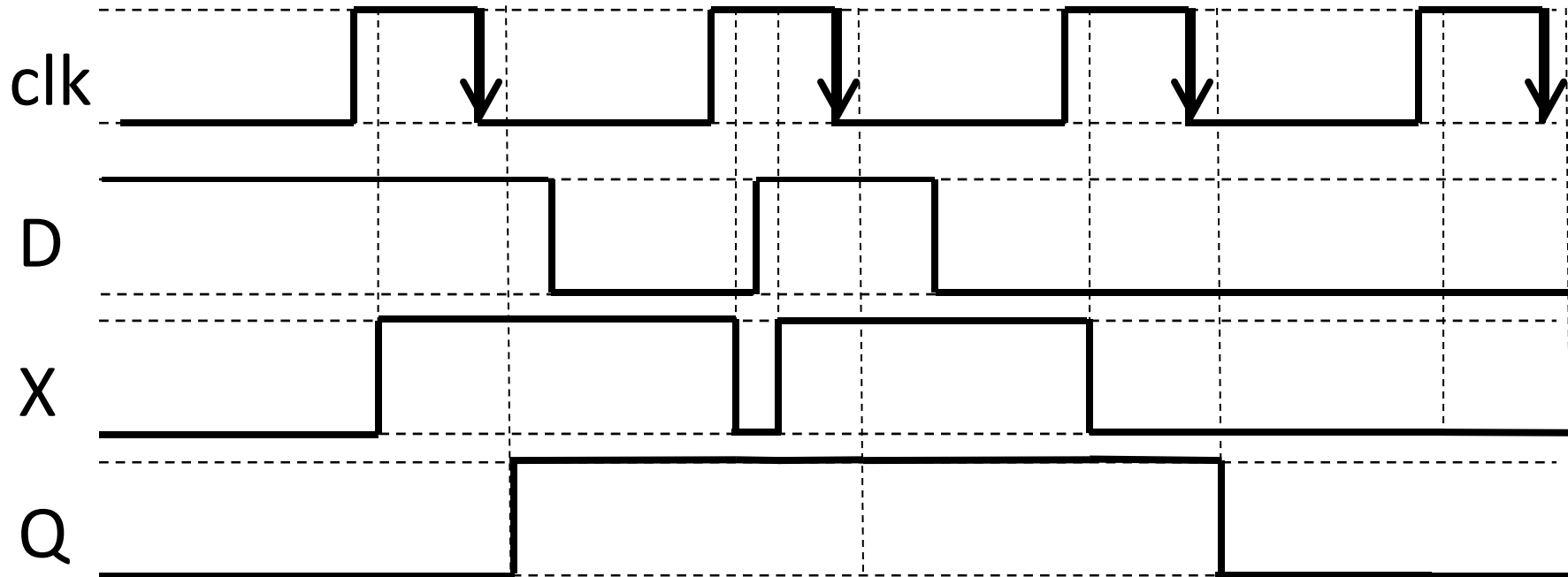


Sixth Attempt: Edge-Triggered D Flip-Flop



D Flip-Flop

- Edge-Triggered
- Data captured when clock is high
- Output changes only on falling edges



Takeaway

Set-Reset (SR) Latch can store one bit and we can change the value of the stored bit. But, SR Latch has a forbidden state.

(Unclocked) D Latch can store and change a bit like an SR Latch while avoiding a forbidden state.

An Edge-Triggered D Flip-Flop (aka Master-Slave D Flip-Flop) stores one bit. The bit can be changed in a synchronized fashion on the edge of a clock signal.

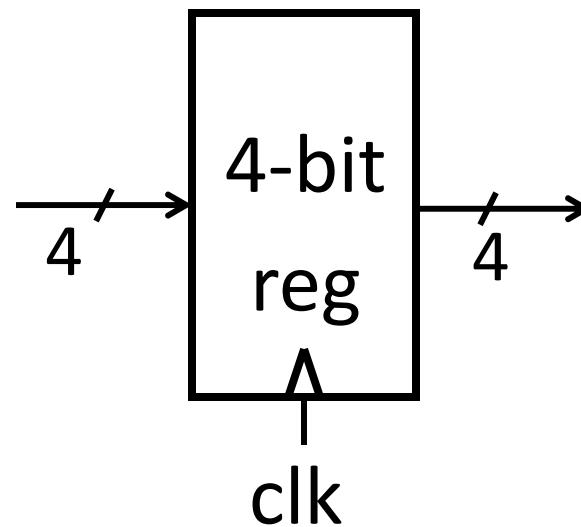
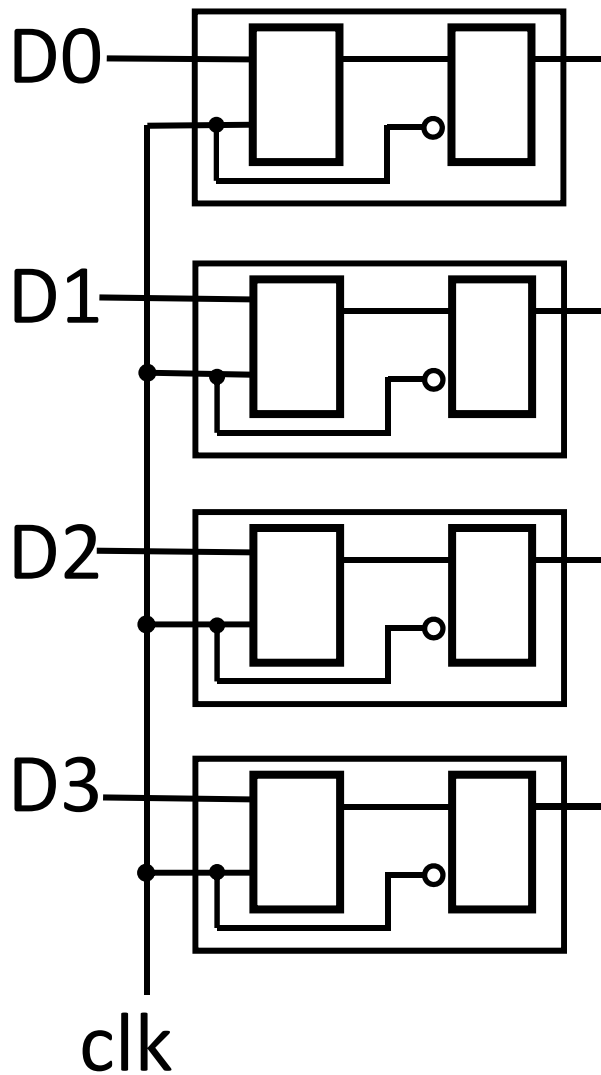
Next Goal

How do we store more than one bit, N bits?

Registers

Register

- D flip-flops in parallel
- shared clock
- extra clocked inputs: write_enable, reset, ...



Takeaway

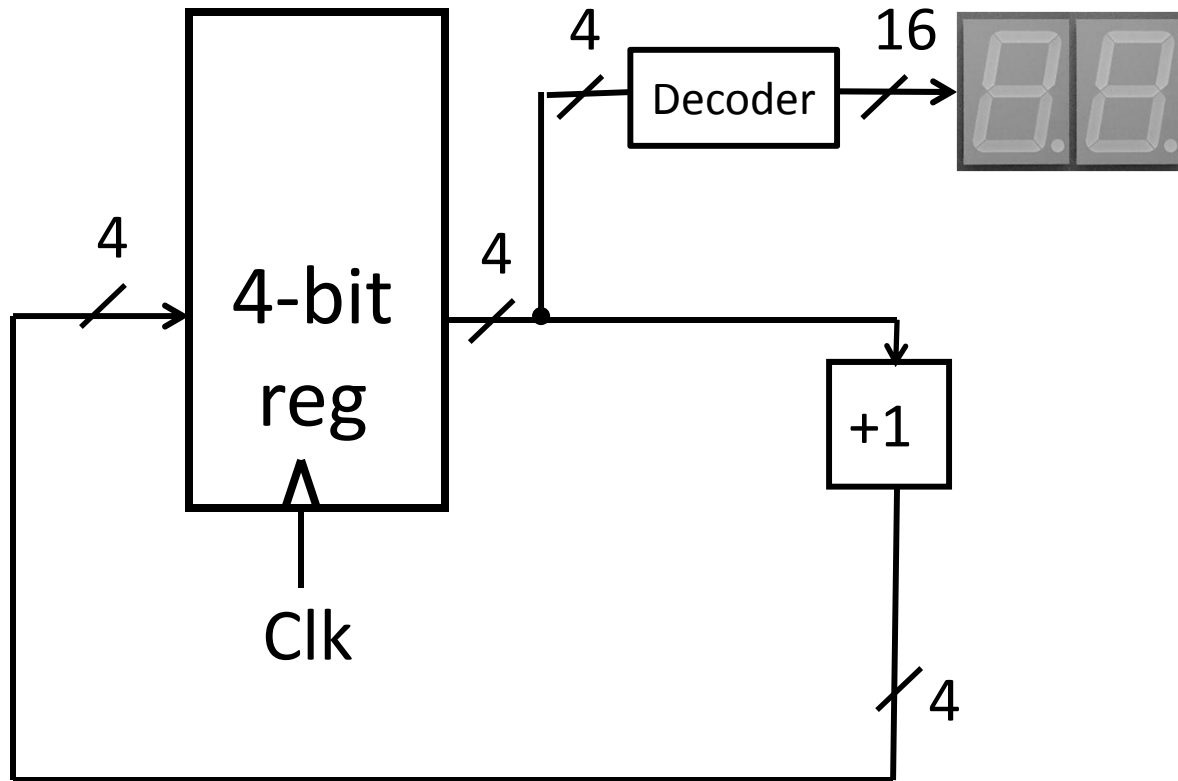
Set-Reset (SR) Latch can store one bit and we can change the value of the stored bit. But, SR Latch has a forbidden state.

(Unclocked) D Latch can store and change a bit like an SR Latch while avoiding a forbidden state.

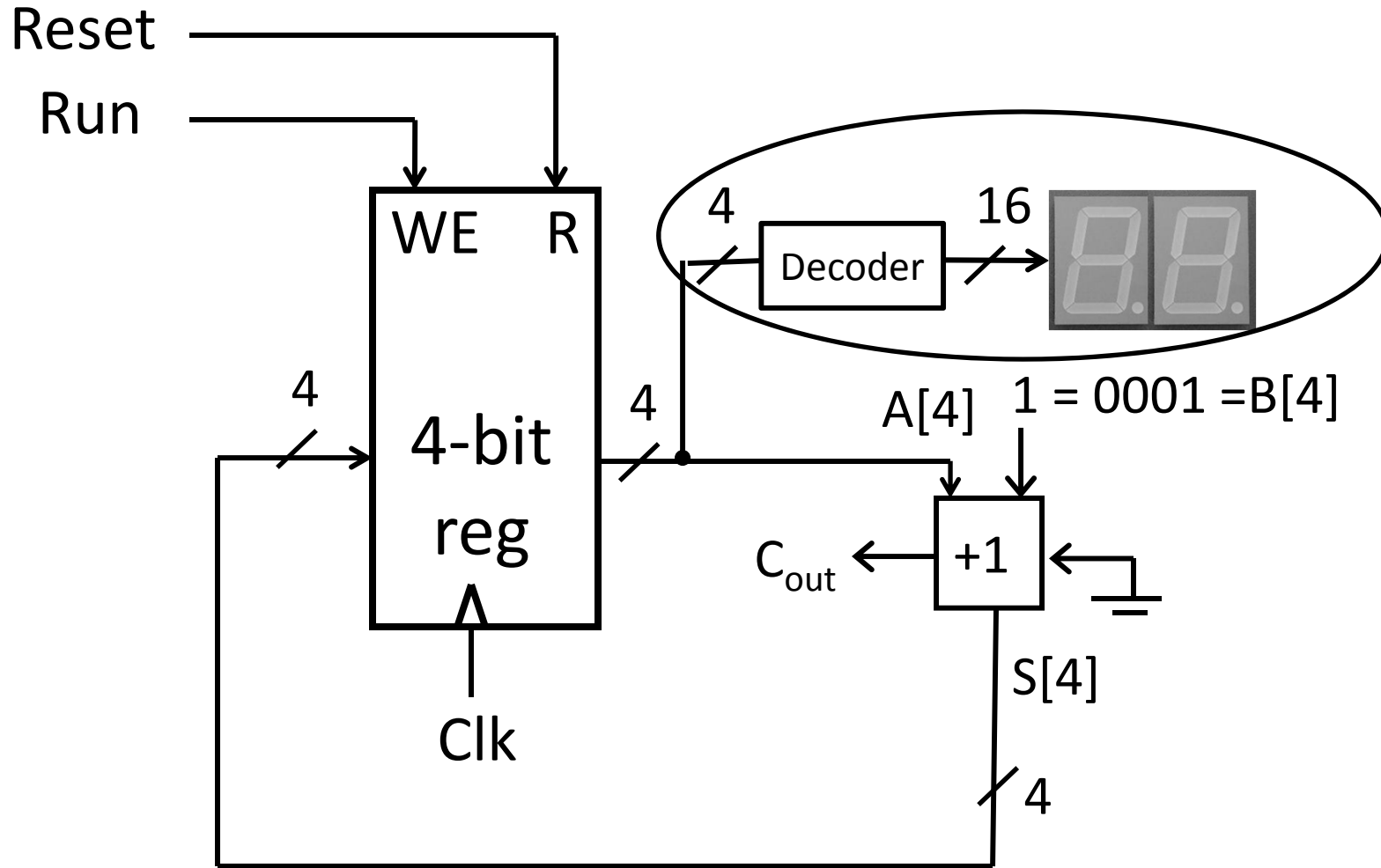
An Edge-Triggered D Flip-Flop (aka Master-Slave D Flip-Flop) stores one bit. The bit can be changed in a synchronized fashion on the edge of a clock signal.

An N -bit **register** stores N -bits. It is created with N D-Flip-Flops in parallel along with a shared clock.

An Example: What will this circuit do?



An Example: What will this circuit do?



Decoder Example: 7-Segment LED

7-Segment LED

- photons emitted when electrons fall into holes



Decoder Example: 7-Segment LED

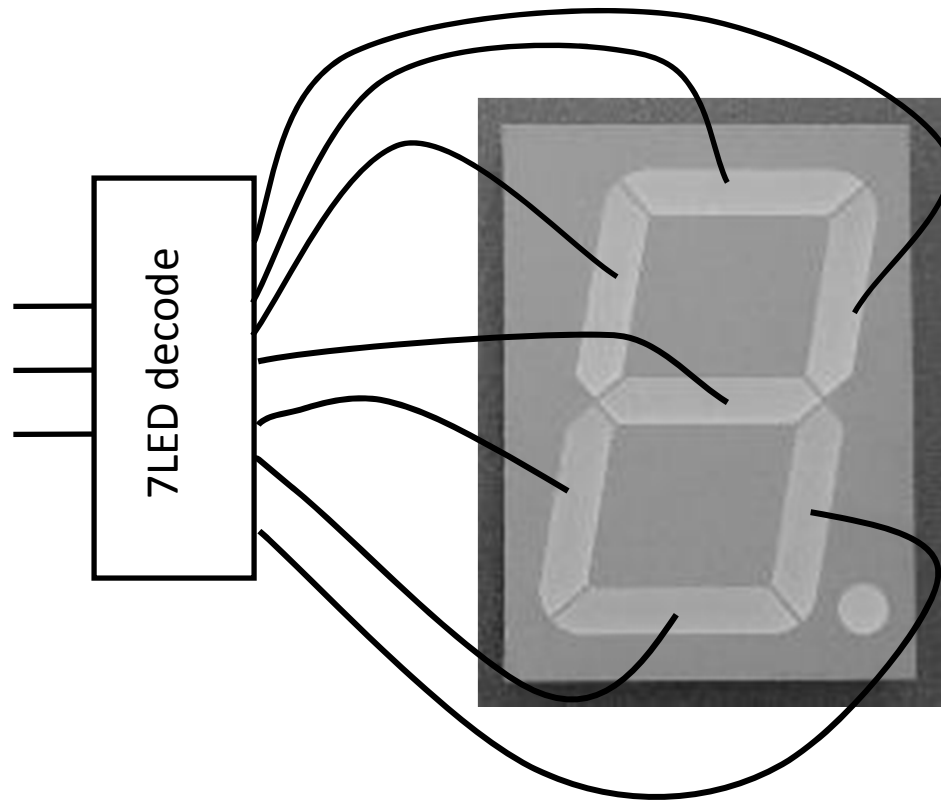
7-Segment LED

- photons emitted when electrons fall into holes



d0 1 2 3 4 5 6 7
5 = 0 1 1 0 1 0 1 1

Decoder Example: 7-Segment LED Decoder



3 inputs

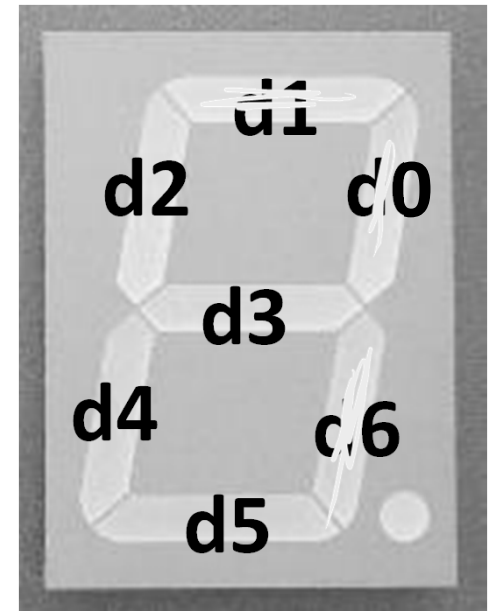
- encode 0 – 7 in binary

7 outputs

- one for each LED

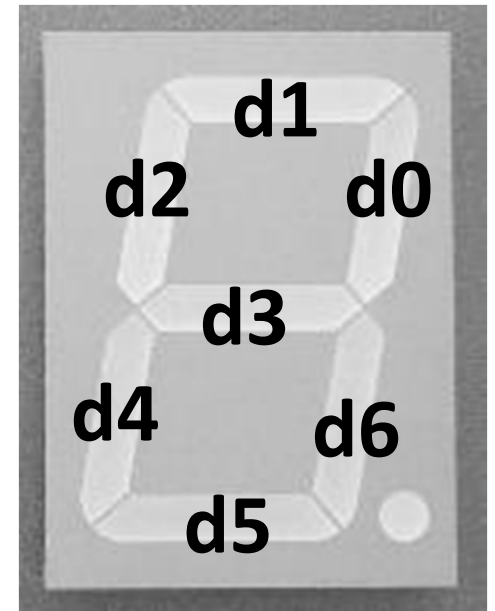
7 Segment LED Decoder Implementation

| b2 | b1 | b0 | d6 | d5 | d4 | d3 | d2 | d1 | d0 |
|----|----|----|----|----|----|----|----|----|----|
| 0 | 0 | 0 | | | | | | | |
| 0 | 0 | 1 | | | | | | | |
| 0 | 1 | 0 | | | | | | | |
| 0 | 1 | 1 | | | | | | | |
| 1 | 0 | 0 | | | | | | | |
| 1 | 0 | 1 | | | | | | | |
| 1 | 1 | 0 | | | | | | | |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |

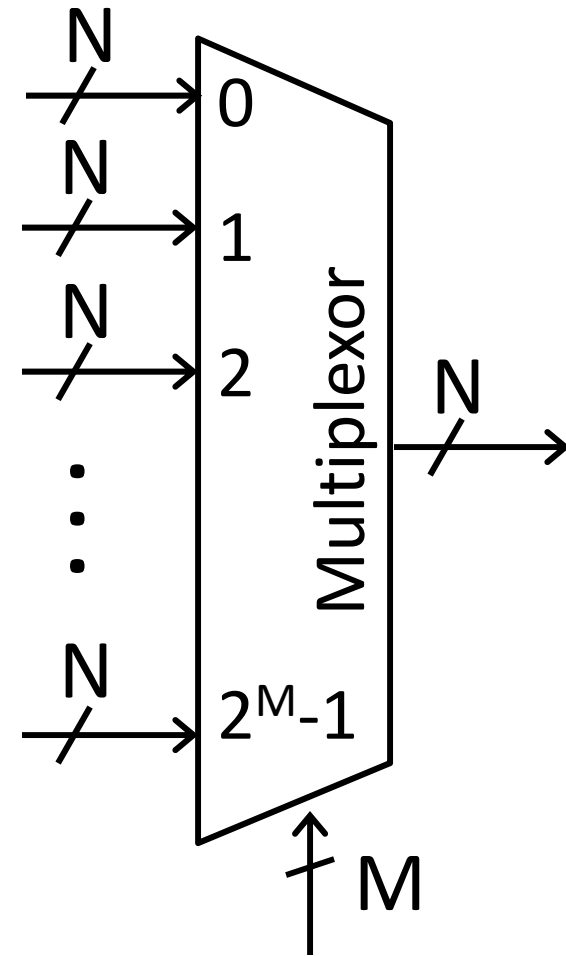
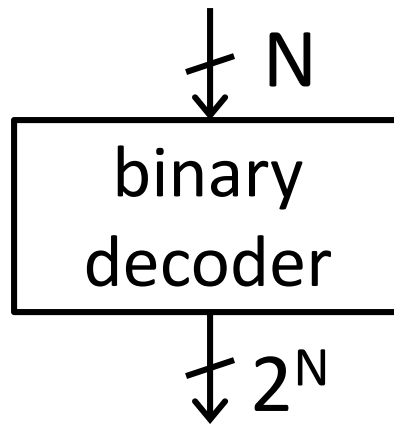
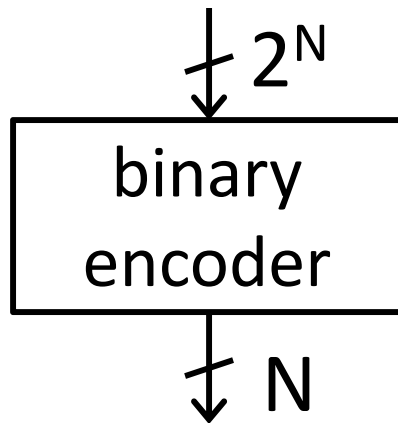


7 Segment LED Decoder Implementation

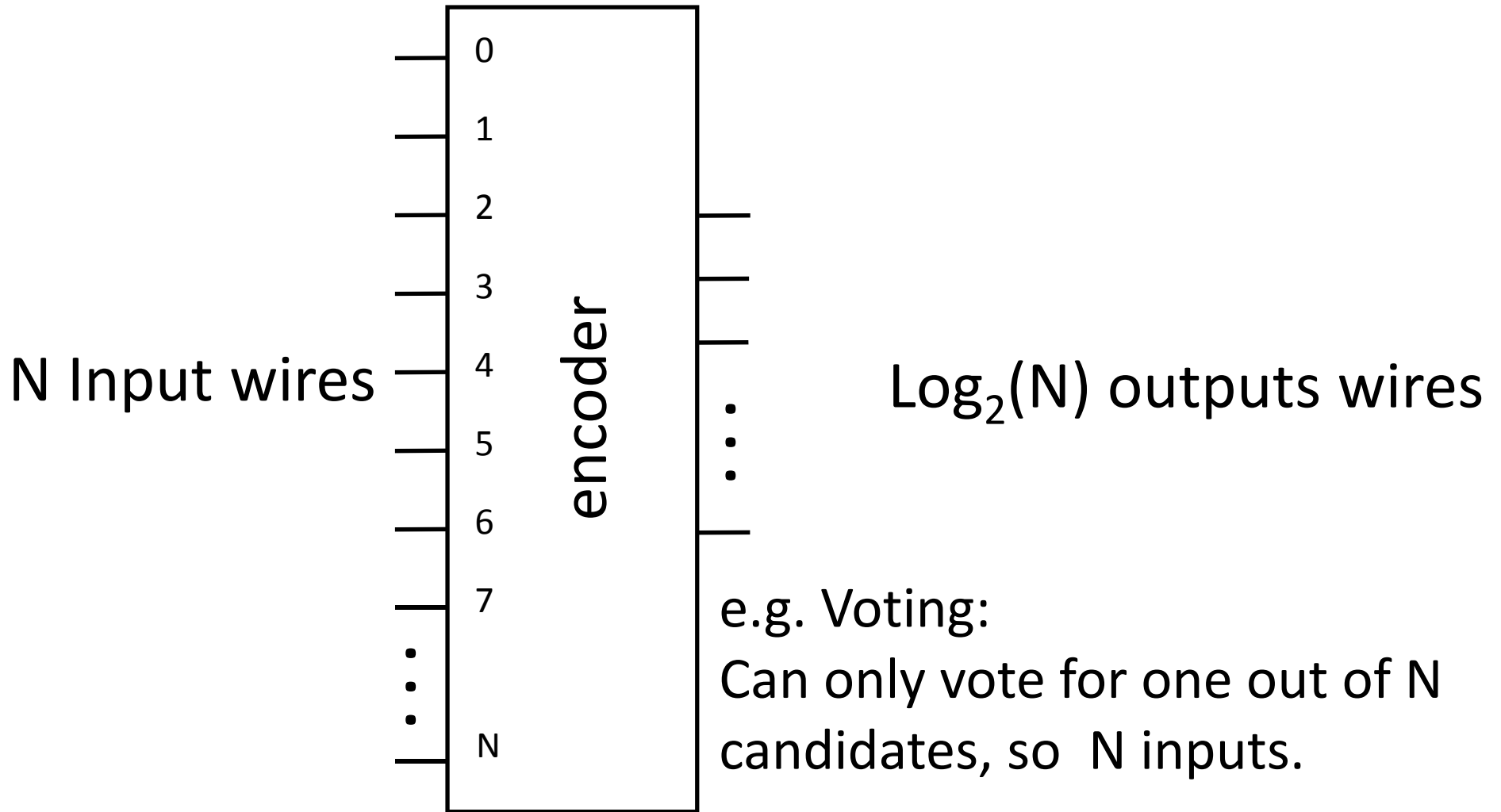
| b2 | b1 | b0 | d6 | d5 | d4 | d3 | d2 | d1 | d0 |
|----|----|----|----|----|----|----|----|----|----|
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |



Basic Building Blocks We have Seen

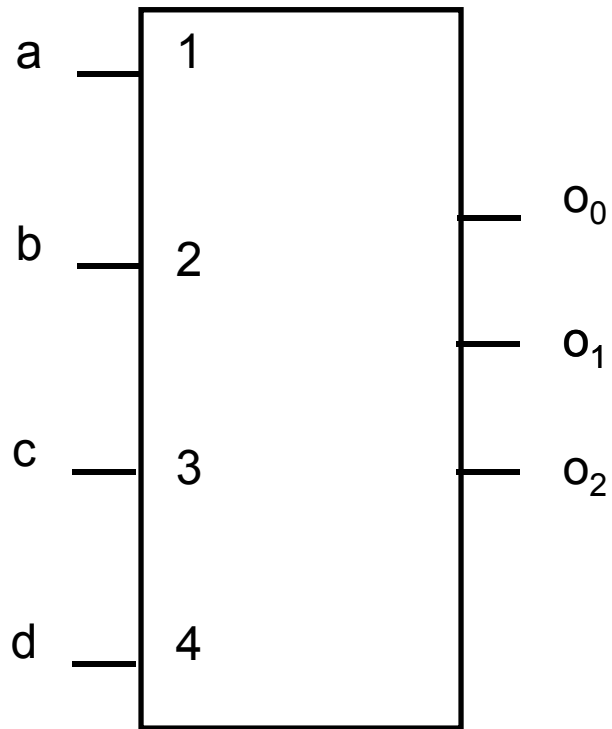


Encoders



But can encode vote efficiently with binary encoding.

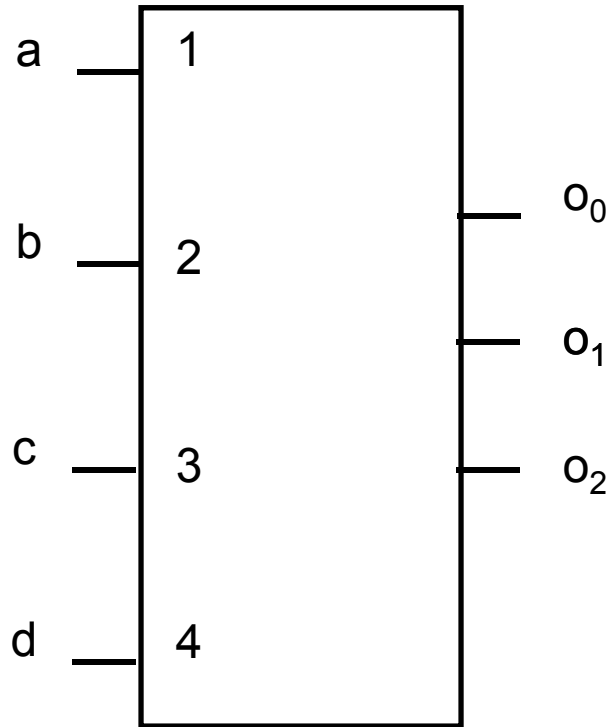
Example Encoder Truth Table



A 3-bit
encoder
with 4 inputs
for simplicity

| a | b | c | d | | | | |
|---|---|---|---|--|--|--|--|
| 0 | 0 | 0 | 0 | | | | |
| 1 | 0 | 0 | 0 | | | | |
| 0 | 1 | 0 | 0 | | | | |
| 0 | 0 | 1 | 0 | | | | |
| 0 | 0 | 0 | 1 | | | | |

Example Encoder Truth Table



A 3-bit
encoder
with 4 inputs
for simplicity

| a | b | c | d | | o_2 | o_1 | o_0 |
|---|---|---|---|--|-------|-------|-------|
| 0 | 0 | 0 | 0 | | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | | 1 | 0 | 0 |

- $o_2 = \overline{abcd}$
- $o_1 = \overline{a}bcd + a\overline{b}cd$
- $o_0 = ab\overline{c}d + abcd\overline{d}$

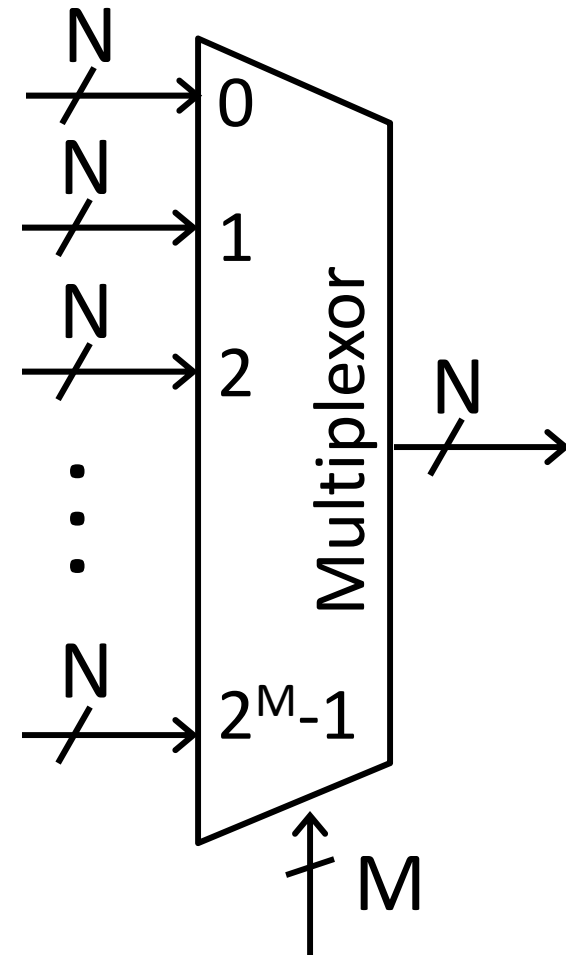
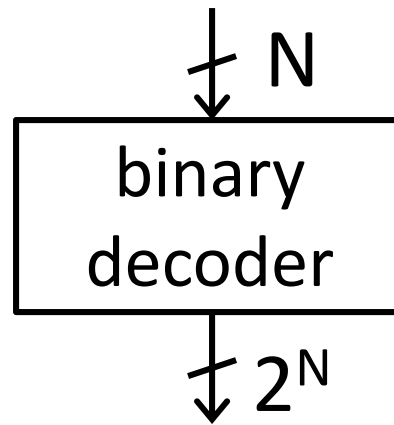
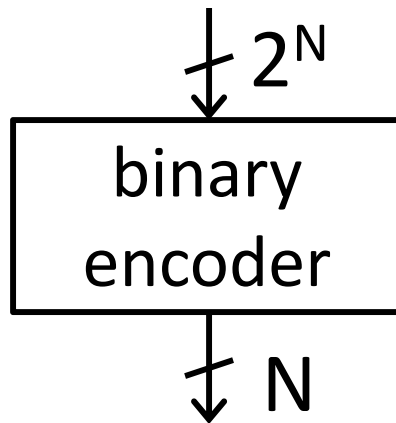
Basic Building Blocks Example: Voting



Ballots

The 3410 optical scan
vote reader
machine

Basic Building Blocks We have Seen



Recap

We can now build interesting devices with sensors

- Using combinational logic

We can also store data values (aka Sequential Logic)

- In state-holding elements
- Coupled with clocks

Goals for Today

State

- How do we store *one* bit?
- Attempts at storing (and changing) one bit
 - Set-Reset Latch
 - D Latch
 - D Flip-Flops
 - Master-Slave Flip-Flops
- Register: storing more than one bit, N-bits

Basic Building Blocks

- Decoders and Encoders

Finite State Machines (FSM)

- How do we design logic circuits with state?
- Types of FSMs: Mealy and Moore Machines
- Examples: Serial Adder and a Digital Door Lock

Goals for Today

State

- How do we store *one* bit?
- Attempts at storing (and changing) one bit
 - Set-Reset Latch
 - D Latch
 - D Flip-Flops
 - Master-Slave Flip-Flops
- Register: storing more than one bit, N-bits

Basic Building Blocks

- Decoders and Encoders

Finite State Machines (FSM)

- How do we design logic circuits with state?
- Types of FSMs: Mealy and Moore Machines
- Examples: Serial Adder and a Digital Door Lock

Finite State Machines

Next Goal

How do we design logic circuits with state?

Finite State Machines

An electronic machine which has

- external inputs
- externally visible outputs
- internal state

Output and next state depend on

- inputs
- current state

Abstract Model of FSM

Machine is

$$M = (S, I, O, \delta)$$

S : Finite set of states

I : Finite set of inputs

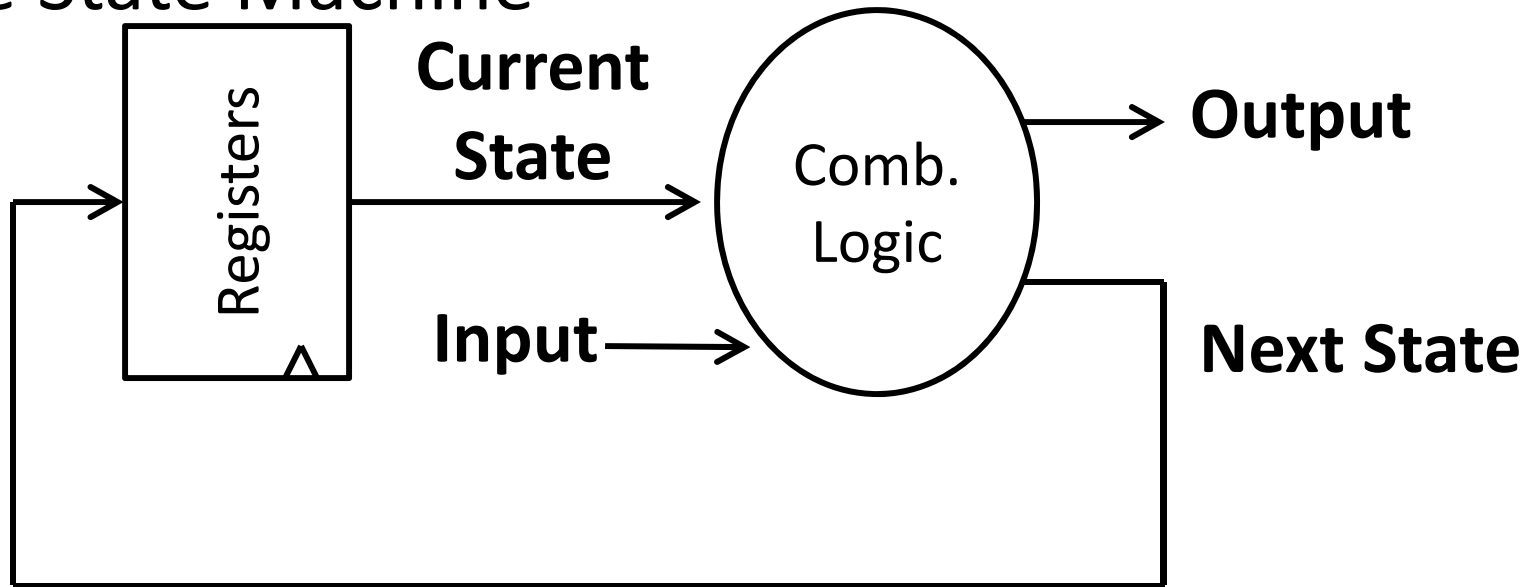
O : Finite set of outputs

δ : State transition function

Next state depends on present input *and* present state

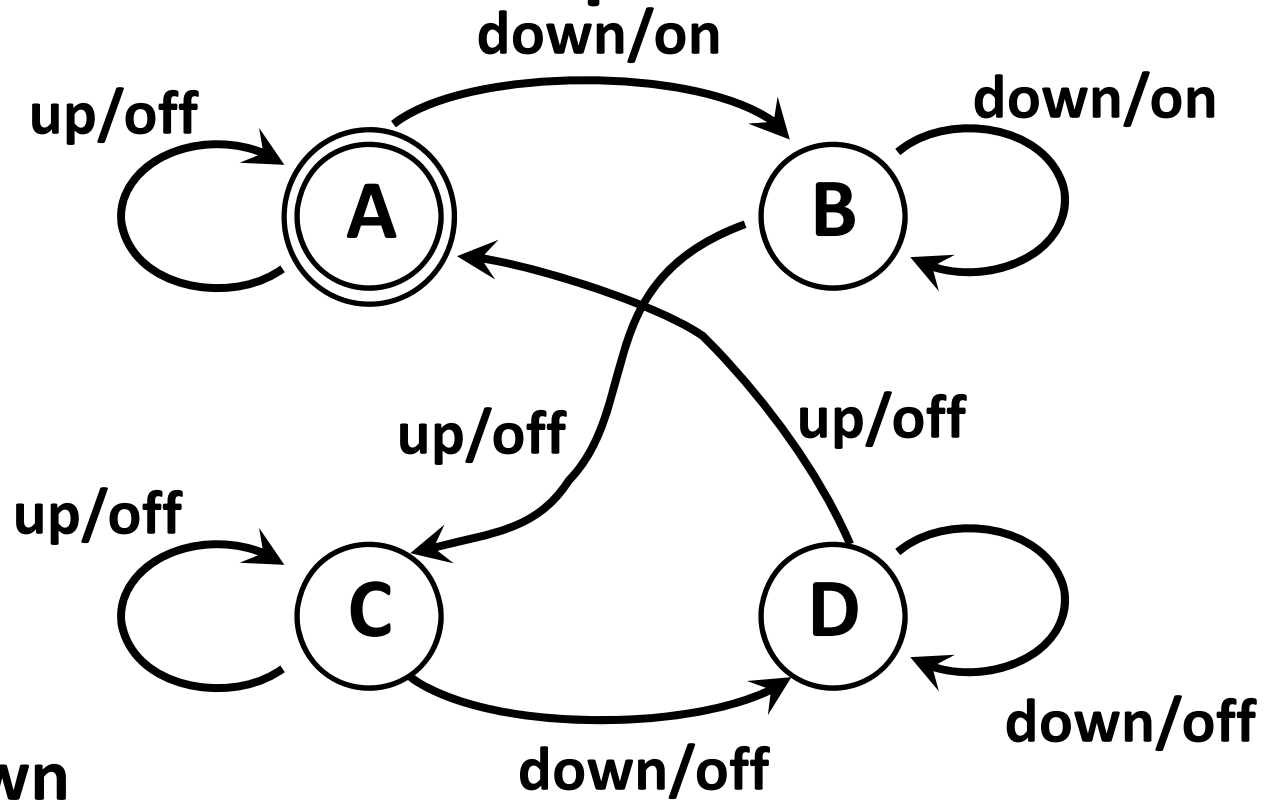
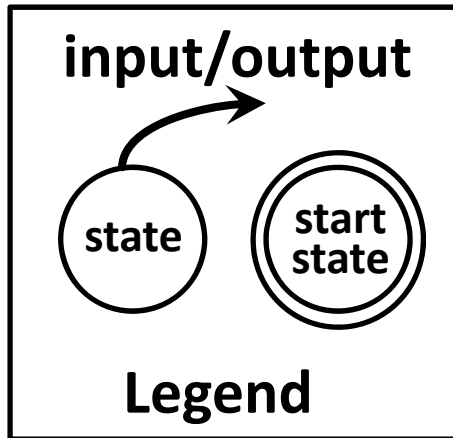
Automata Model

Finite State Machine



- inputs from external world
- outputs to external world
- internal state
- combinational logic

FSM Example

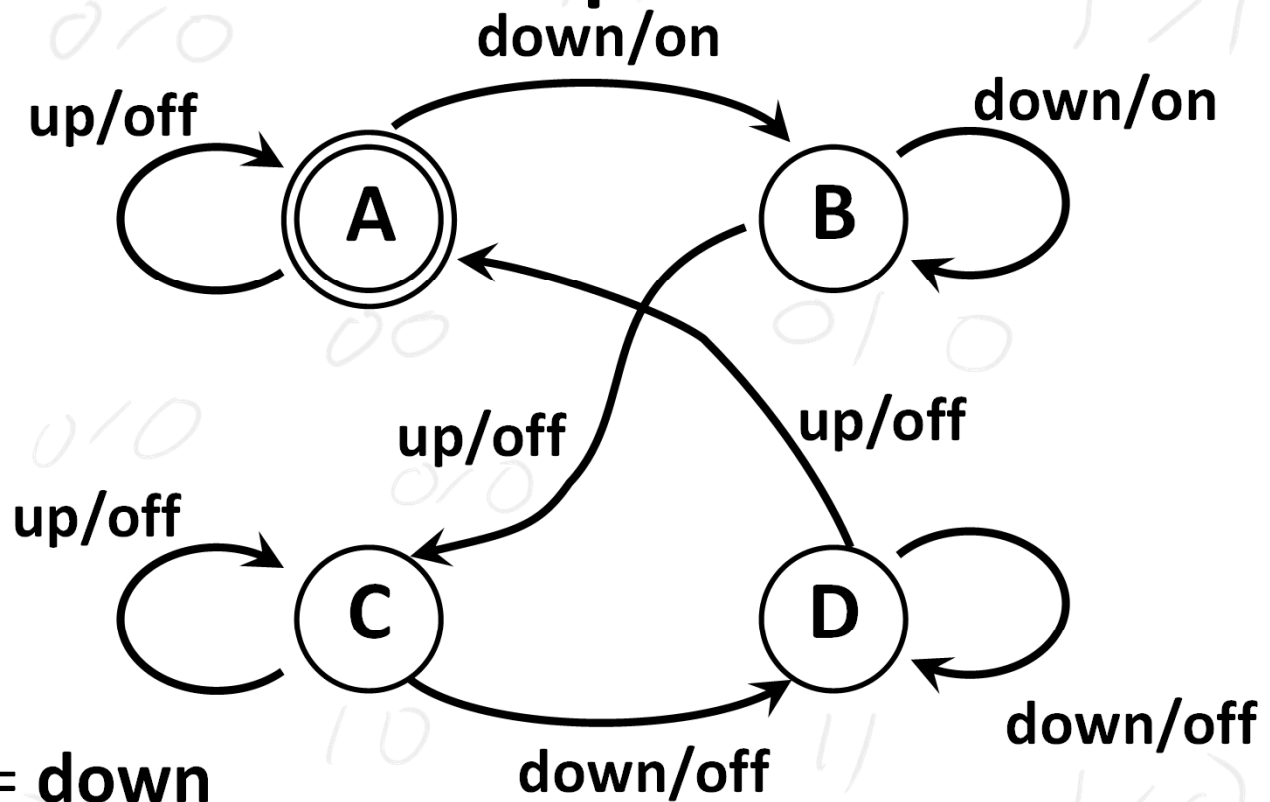
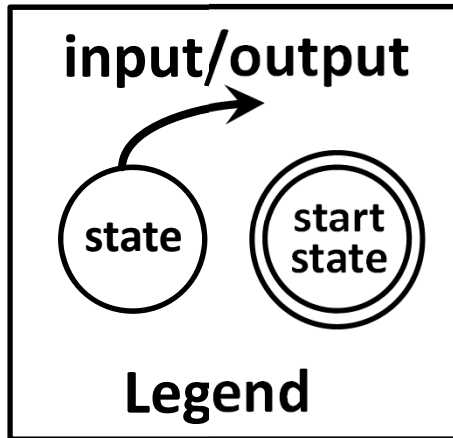


Input: **up** or **down**

Output: **on** or **off**

States: **A**, **B**, **C**, or **D**

FSM Example

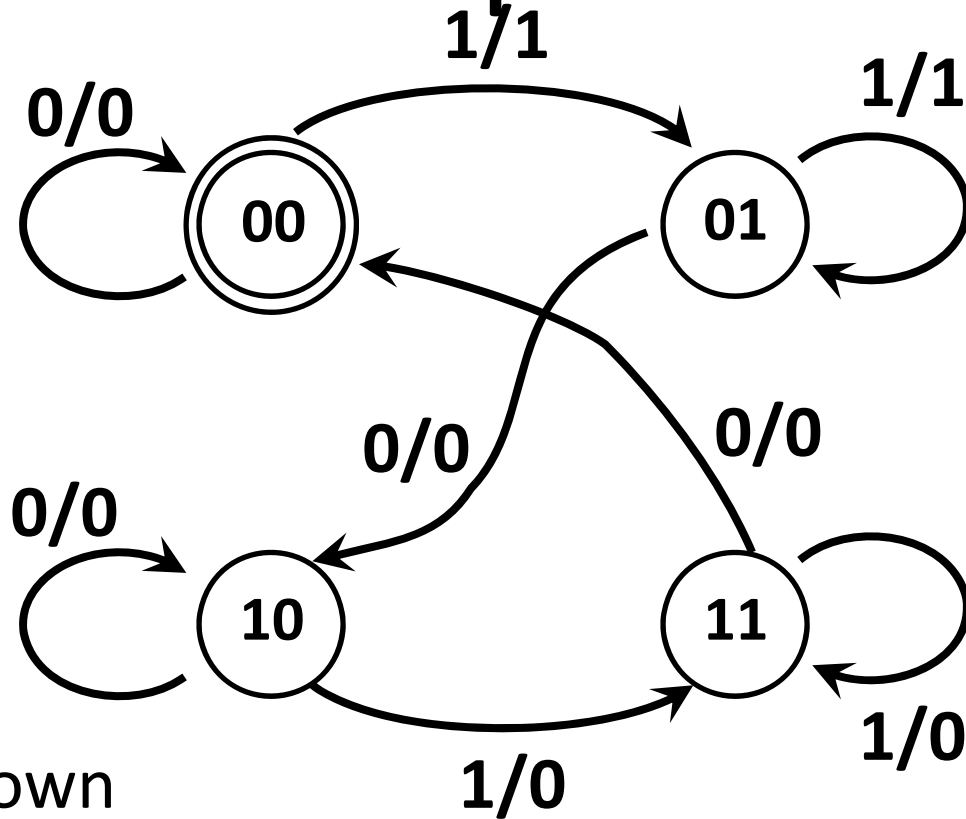
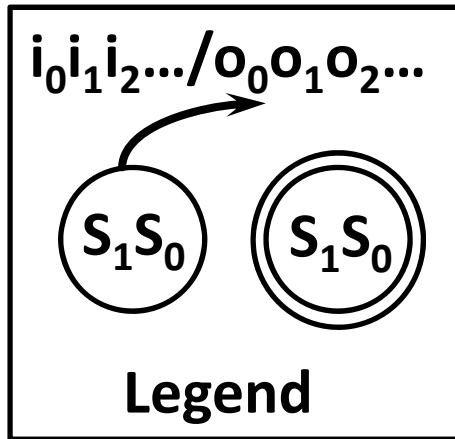


Input: = **up** or = **down**

Output: = **on** or = **off**

States: = **A**, = **B**, = **C**, or = **D**

FSM Example



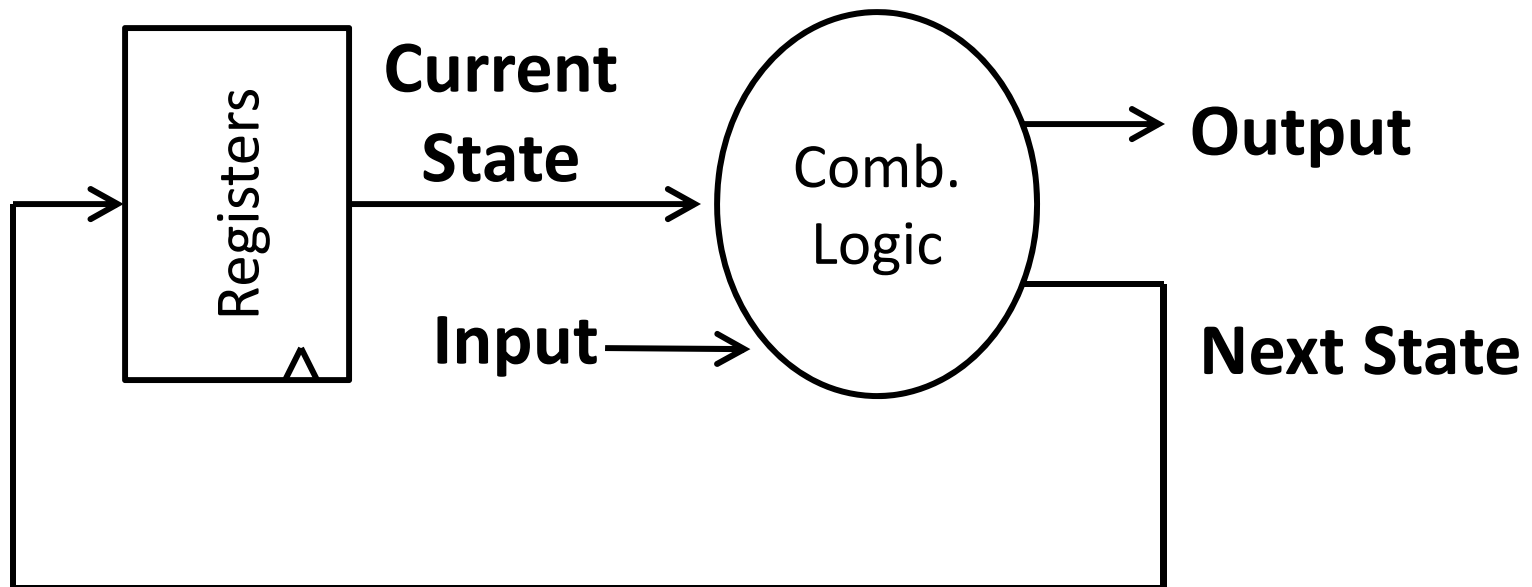
Input: **0**=up or **1**=down

Output: **1**=on or **0**=off

States: **00**=A, **01**=B, **10**=C, or **11**=D

Mealy Machine

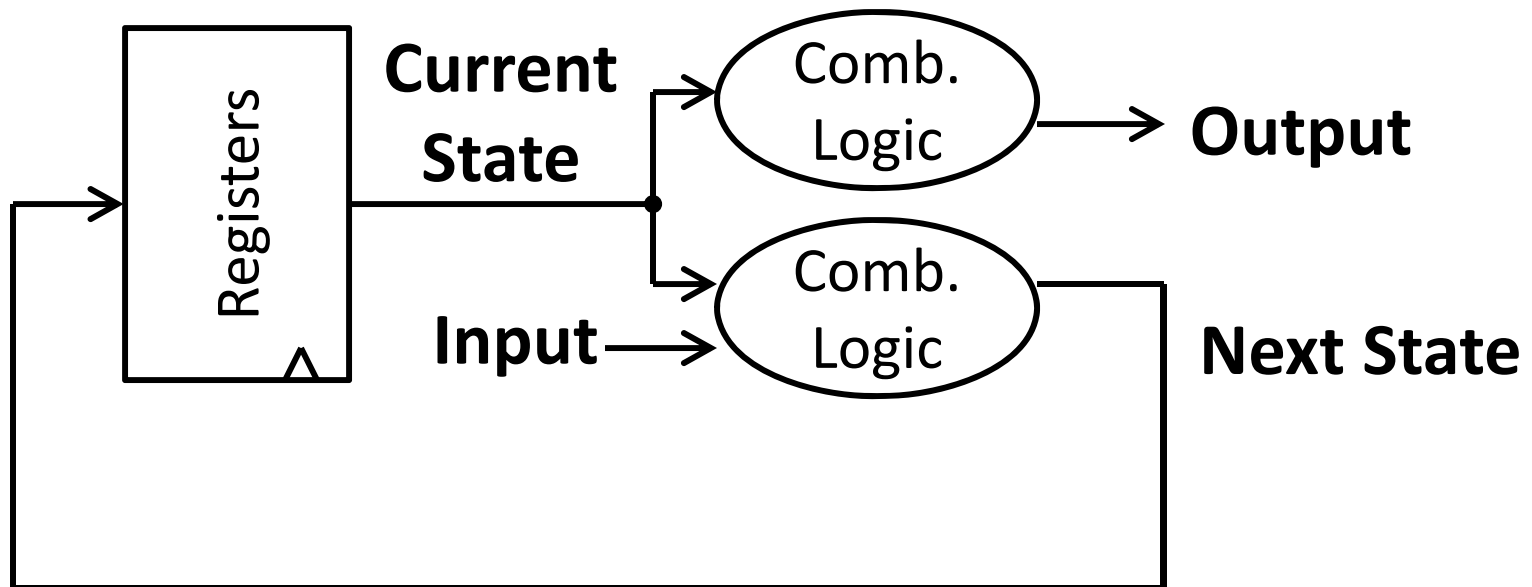
General Case: Mealy Machine



Outputs and next state depend on both current state and input

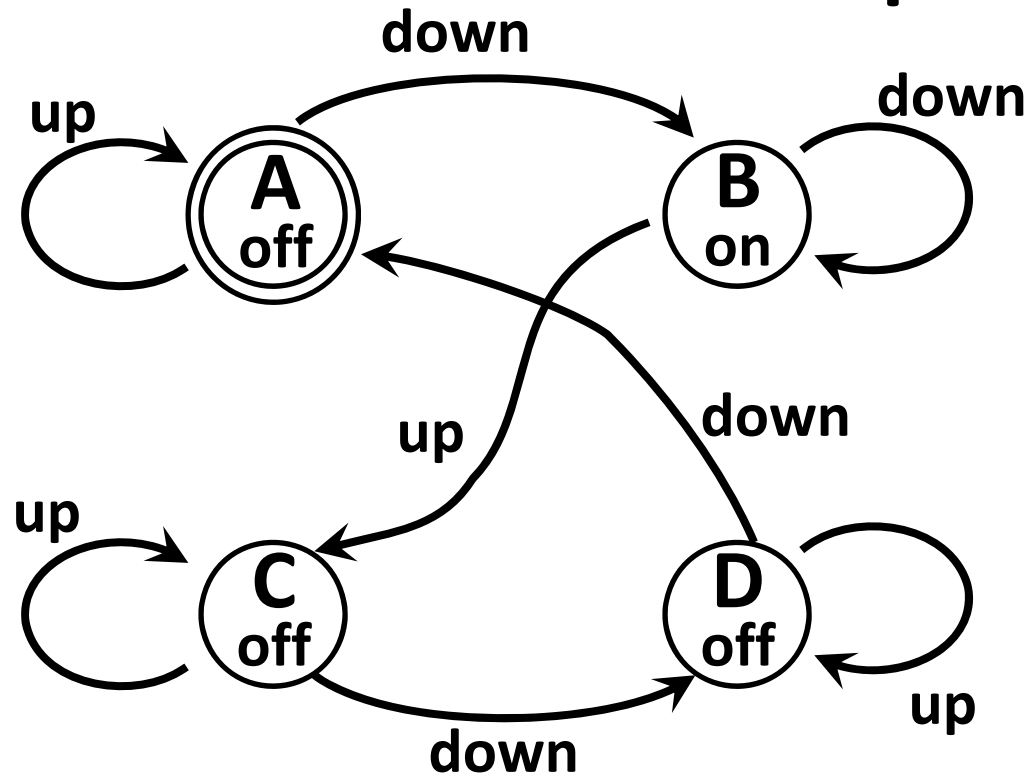
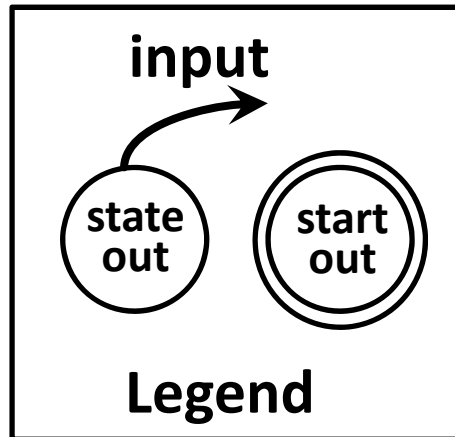
Moore Machine

Special Case: Moore Machine



Outputs depend only on current state

Moore Machine FSM Example

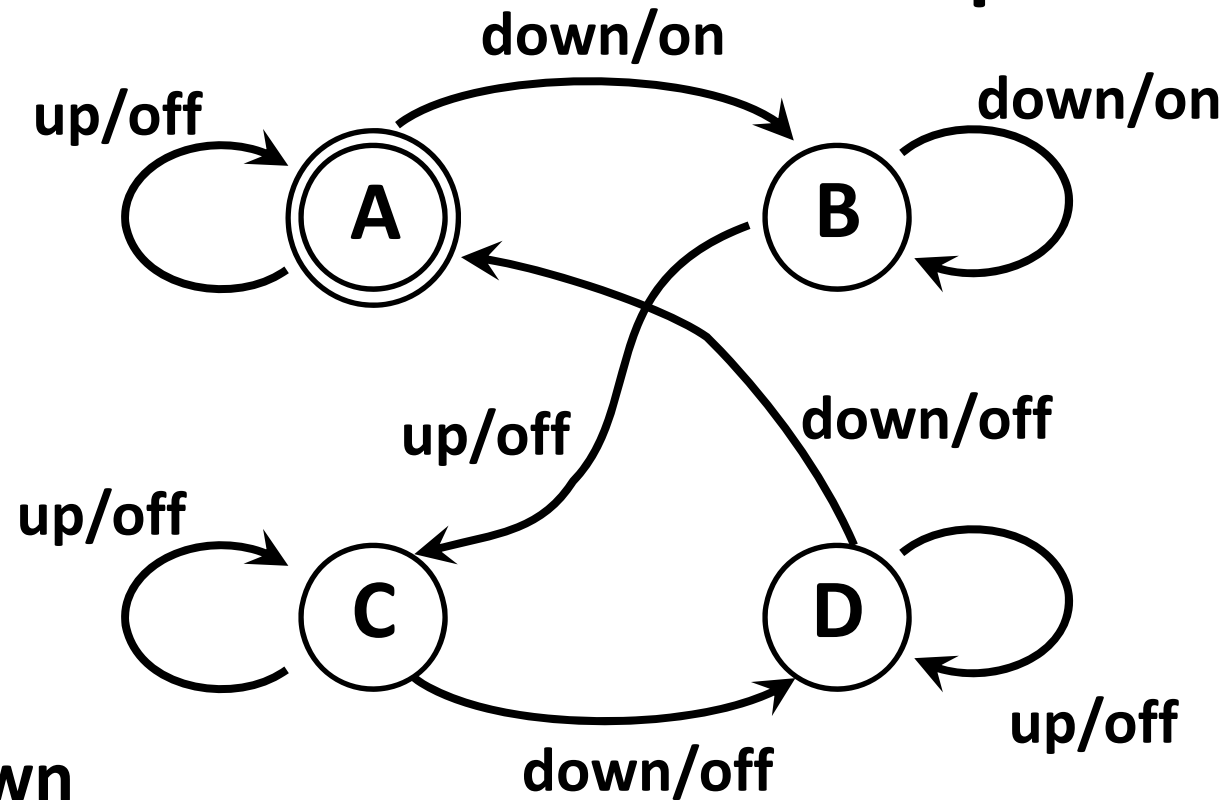
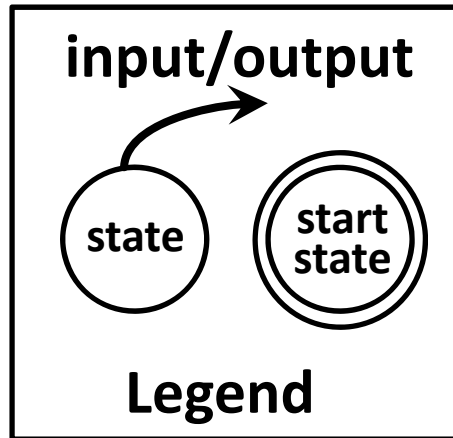


Input: **up** or **down**

Output: **on** or **off**

States: **A, B, C, or D**

Mealy Machine FSM Example



Input: **up** or **down**

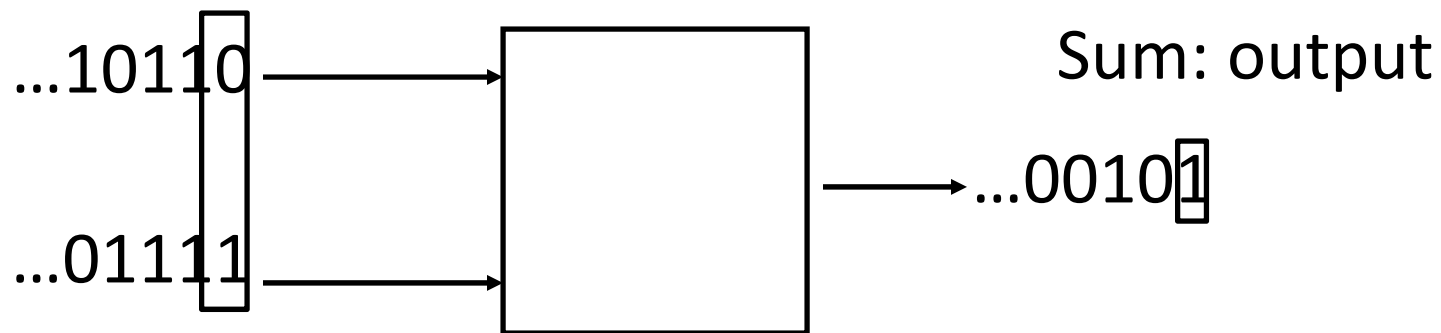
Output: **on** or **off**

States: **A**, **B**, **C**, or **D**

Activity#2: Create a Logic Circuit for a Serial Adder

Add two infinite input bit streams

- streams are sent with least-significant-bit (lsb) first

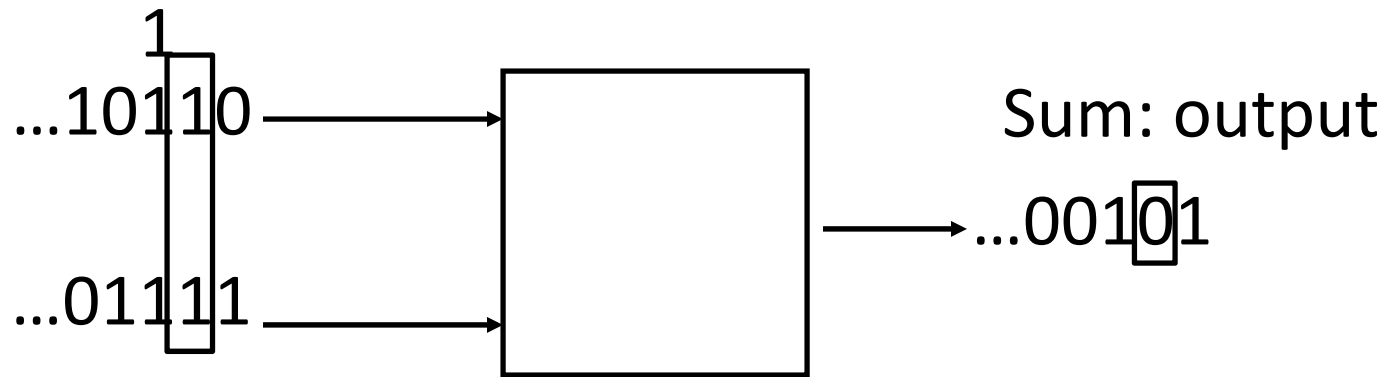


Activity#2: Create a Logic Circuit for a Serial Adder

Add two infinite input bit streams

- streams are sent with least-significant-bit (lsb) first

Carry-out

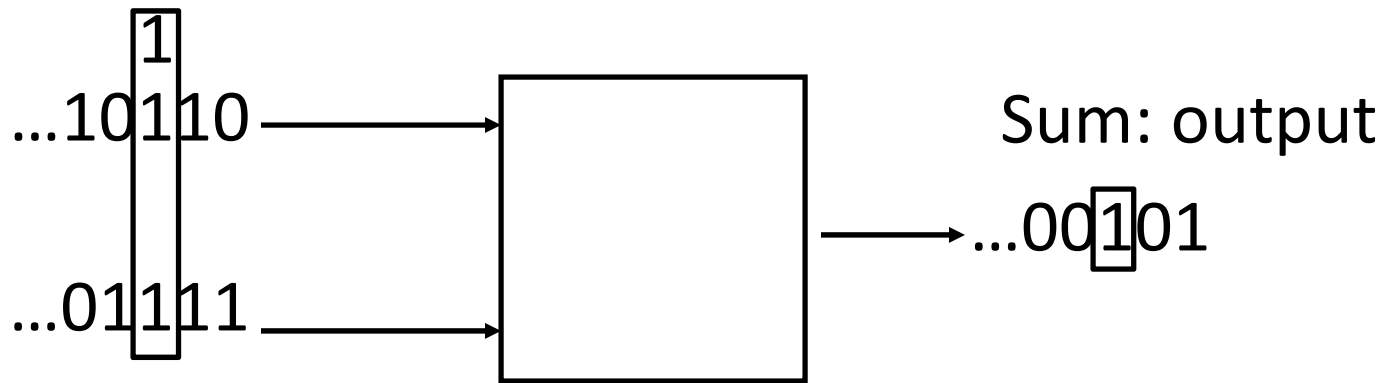


Activity#2: Create a Logic Circuit for a Serial Adder

Add two infinite input bit streams

- streams are sent with least-significant-bit (lsb) first

Carry-*in*

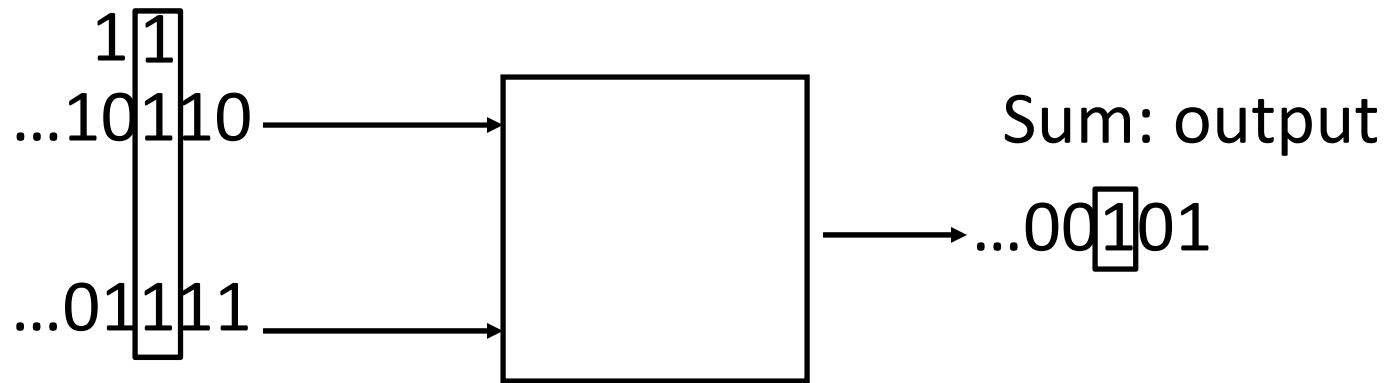


Activity#2: Create a Logic Circuit for a Serial Adder

Add two infinite input bit streams

- streams are sent with least-significant-bit (lsb) first

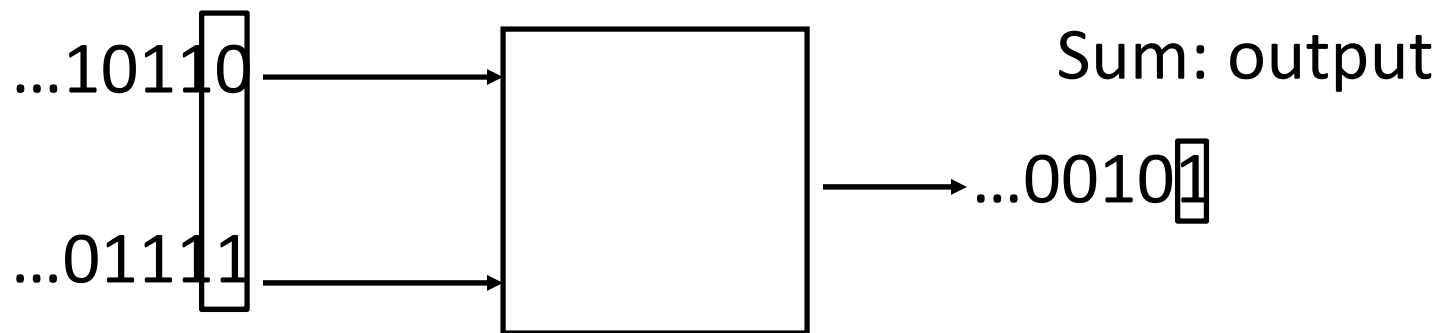
Carry-*out*



Activity#2: Create a Logic Circuit for a Serial Adder

Add two infinite input bit streams

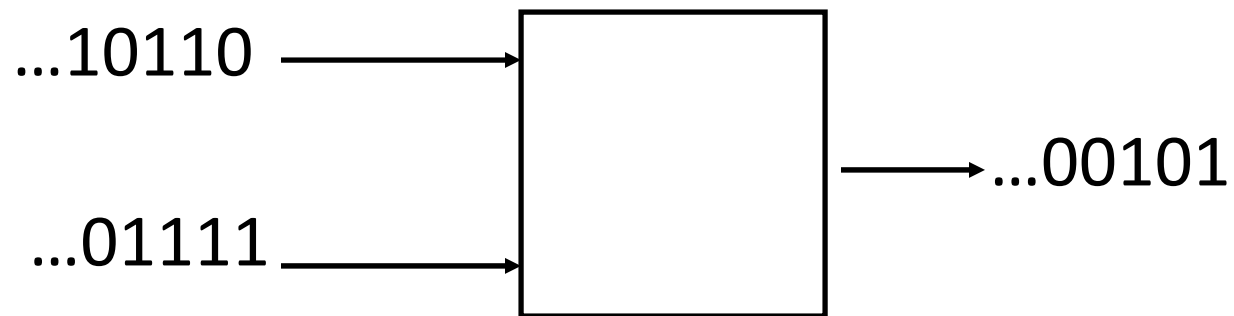
- streams are sent with least-significant-bit (lsb) first
- How many states are needed to represent FSM?
- Draw and Fill in FSM diagram



Strategy:

- (1) Draw a state diagram (e.g. Mealy Machine)
- (2) Write output and next-state tables
- (3) Encode states, inputs, and outputs as bits
- (4) Determine logic equations for next state and outputs

FSM: State Diagram



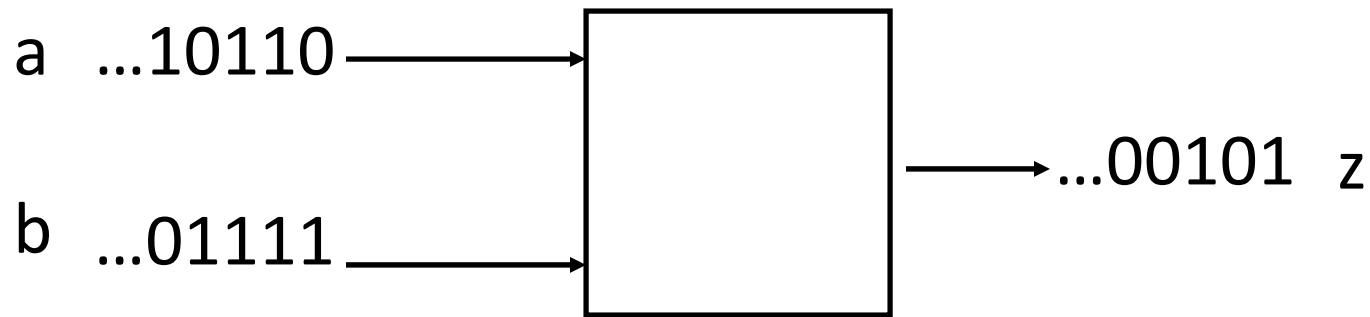
___ states:

Inputs: ??? and ???

Output: ???

• .

FSM: State Diagram



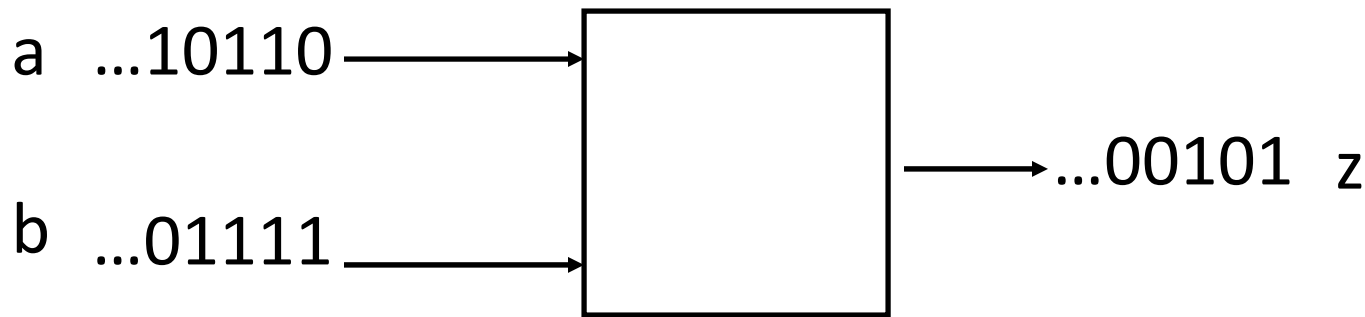
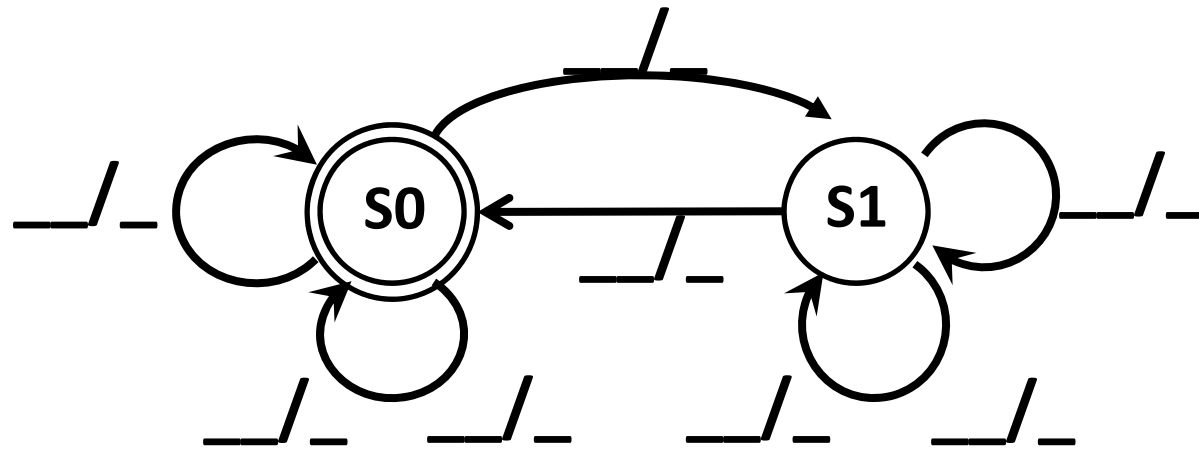
Two states: S_0 (no carry in), S_1 (carry in)

Inputs: a and b

Output: z

- z is the sum of inputs a, b, and carry-in (one bit at a time)
- A carry-out is the next carry-in state.
- .

FSM: State Diagram



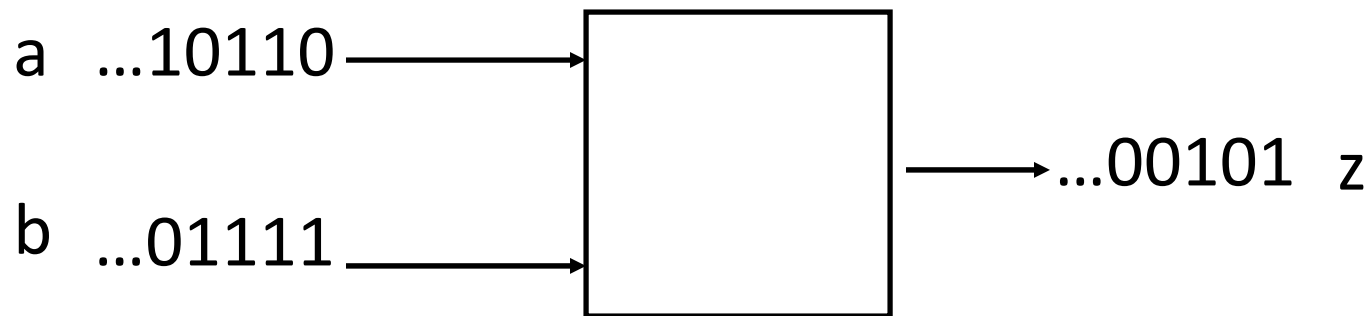
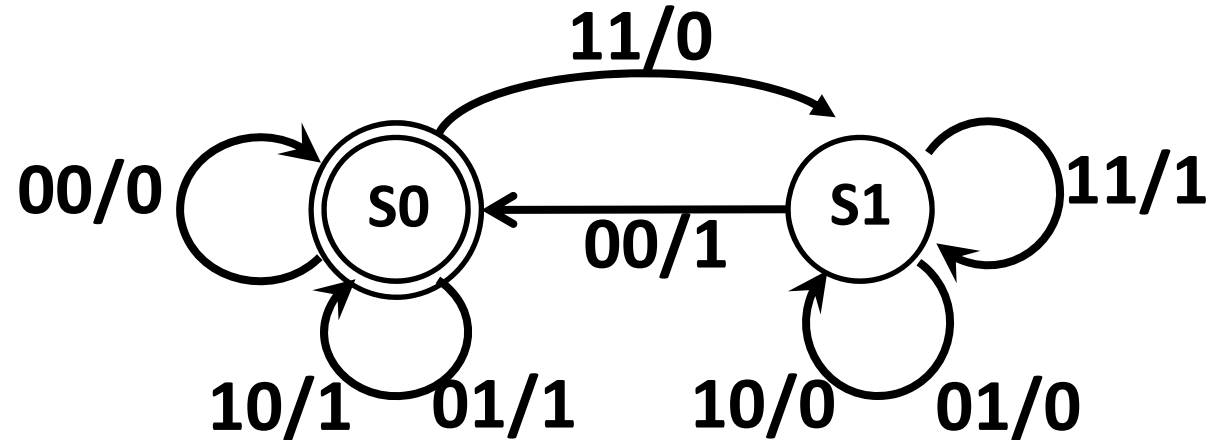
Two states: S0 (no carry in), S1 (carry in)

Inputs: a and b

Output: z

- z is the sum of inputs a, b, and carry-in (one bit at a time)
- A carry-out *is* the next carry-in state.
- Arcs labeled with input bits a and b, and output z

FSM: State Diagram



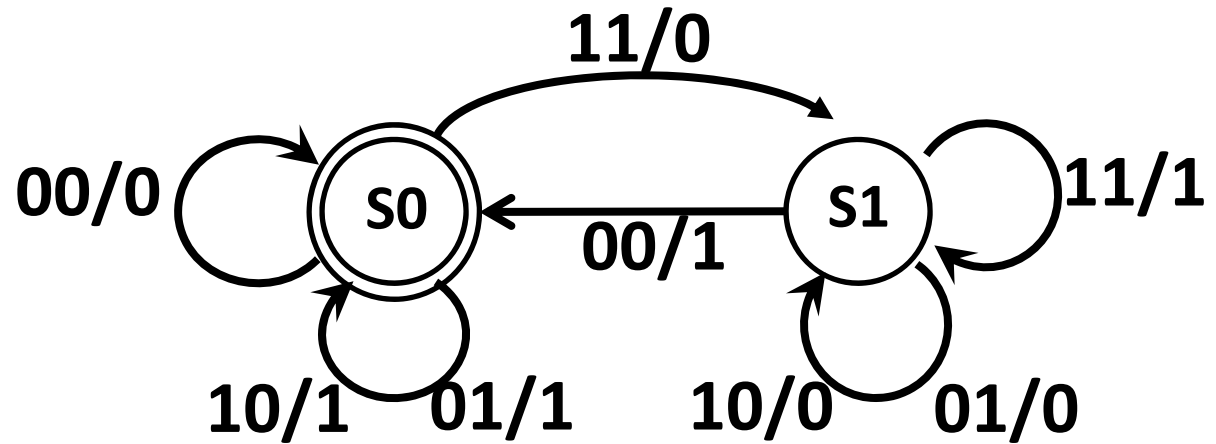
Two states: S0 (no carry in), S1 (carry in)

Inputs: a and b

Output: z

- z is the sum of inputs a, b, and carry-in (one bit at a time)
- A carry-out is the next carry-in state.
- Arcs labeled with input bits a and b, and output z (Mealy Machine)

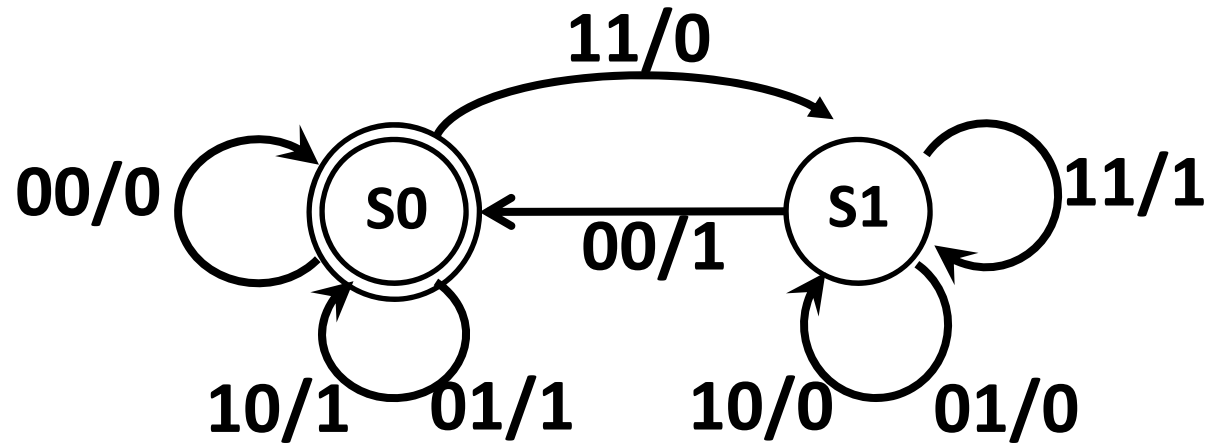
Serial Adder: State Table



| a | b | Current state | z | Next state |
|---|---|---------------|---|------------|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

(2) Write down all input and state combinations

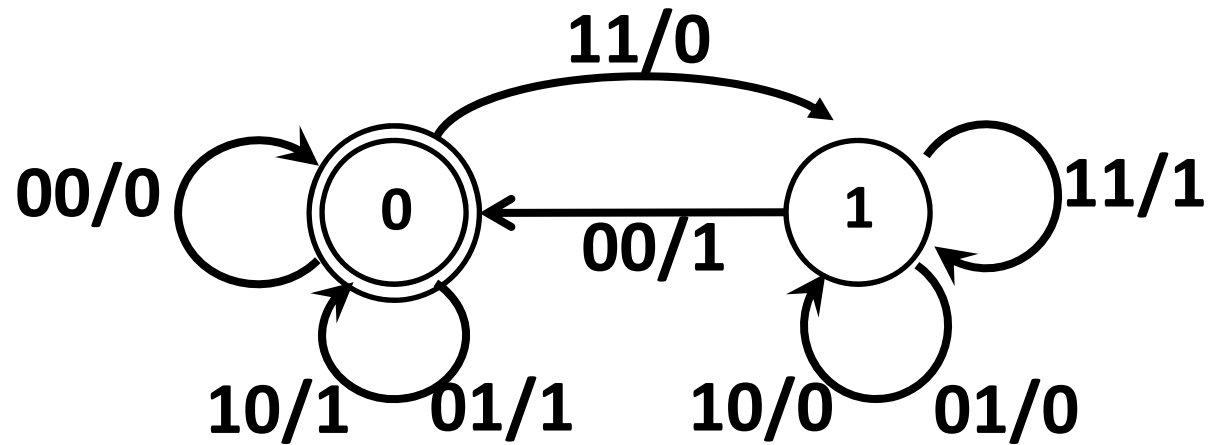
Serial Adder: State Table



| a | b | Current state | z | Next state |
|---|---|---------------|---|------------|
| 0 | 0 | S0 | 0 | S0 |
| 0 | 1 | S0 | 1 | S0 |
| 1 | 0 | S0 | 1 | S0 |
| 1 | 1 | S0 | 0 | S1 |
| 0 | 0 | S1 | 1 | S0 |
| 0 | 1 | S1 | 0 | S1 |
| 1 | 0 | S1 | 0 | S1 |
| 1 | 1 | S1 | 1 | S1 |

(2) Write down all input and state combinations

Serial Adder: State Assignment



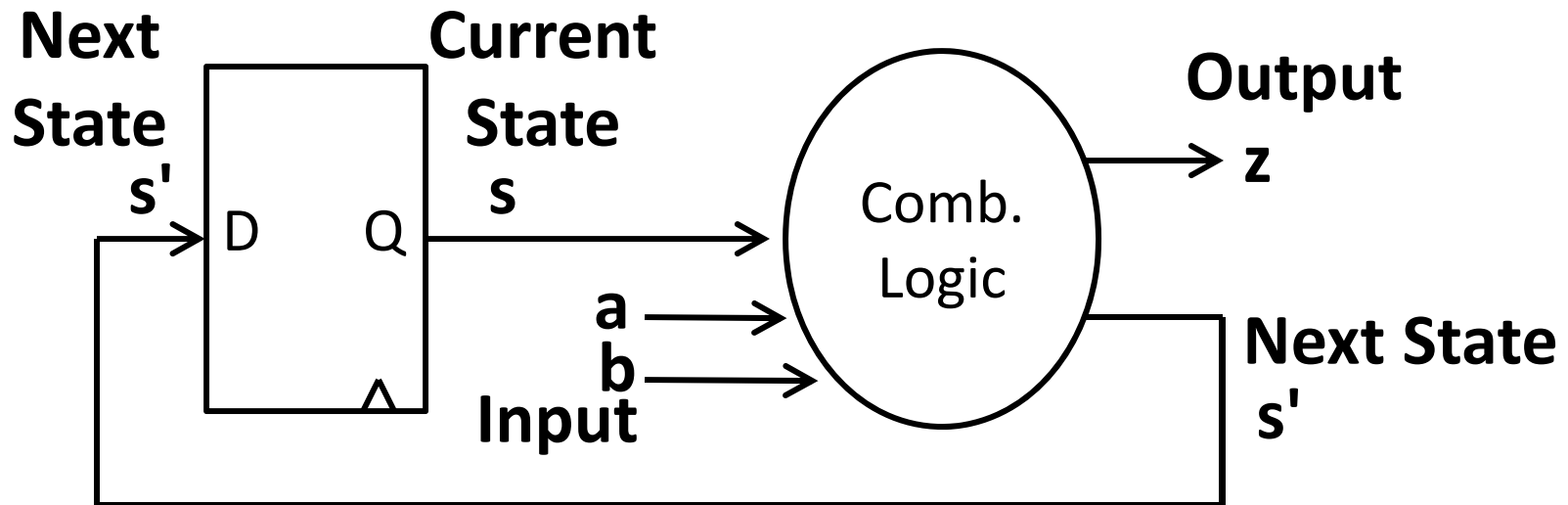
| a | b | s | z | s' |
|---|---|---|---|----|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

(3) Encode states, inputs, and outputs as bits

Two states, so 1-bit is sufficient

- A single flip-flop will encode the state

Serial Adder: Circuit



| a | b | s | z | s' |
|---|---|---|---|----|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

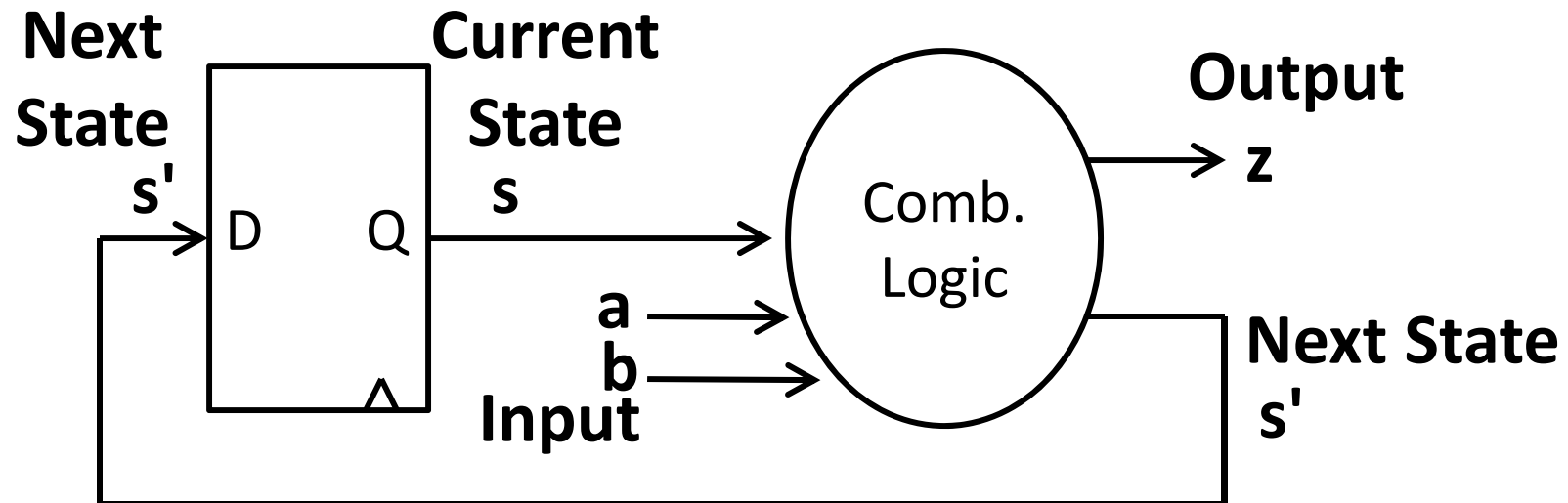
(4) Determine logic equations for next state and outputs

Combinational Logic Equations

$$z = \bar{a}\bar{b}\bar{s} + \bar{a}b\bar{s} + a\bar{b}\bar{s} + abs$$

$$s' = ab\bar{s} + \bar{a}bs + a\bar{b}s + abs$$

Sequential Logic Circuits



$$z = \bar{a}b\bar{s} + a\bar{b}\bar{s} + \bar{a}bs + abs$$

$$s' = ab\bar{s} + \bar{a}bs + a\bar{b}s + abs$$

Strategy:

- (1) Draw a state diagram (e.g. Mealy Machine)
- (2) Write output and next-state tables
- (3) Encode states, inputs, and outputs as bits
- (4) Determine logic equations for next state and outputs

Summary

We can now build interesting devices with sensors

- Using combinational logic

We can also store data values

- Stateful circuit elements (D Flip Flops, Registers, ...)
- Clock to synchronize state changes
- State Machines or Ad-Hoc Circuits