EECS 370 - Lecture 9

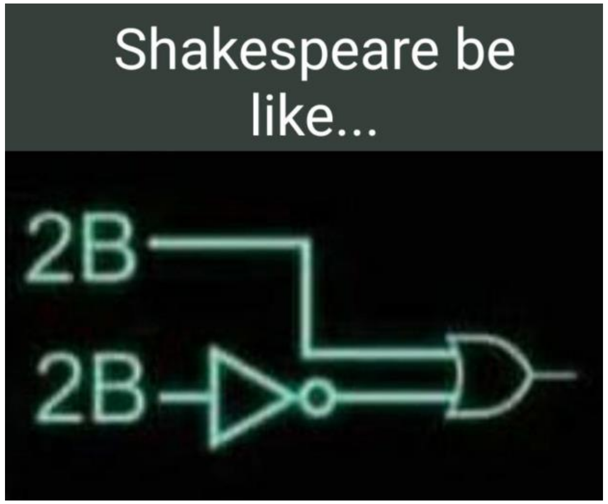
Sequential Logic

Announcements

- P1s & M due tonight!
- P2a due Thu 2/16
  - Walkthrough available on website
- HW 2 due Mon 2/6

Next few lectures: Digital Logic

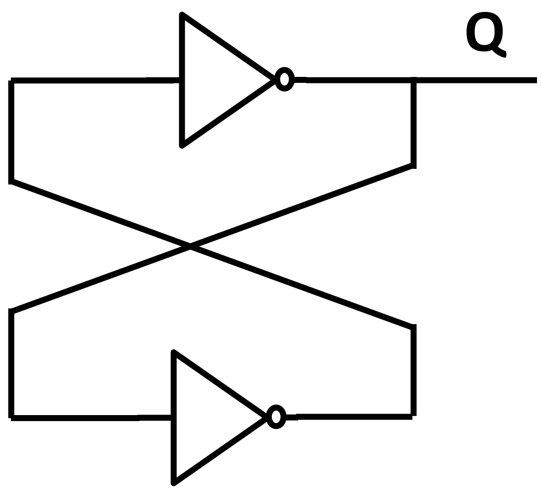
- Lectures 1-7:
  - LC2K and ARMv8/LEGv8 ISAs
  - Converting C to Assembly
  - Function Calls
  - Linking
- Lecture 8:
  - Finish up linking
  - Combinational Logic
- **Today:**
  - **Sequential Logic**



Sequential Logic

- Can we build a processor out of these combinational elements?
- How to build something like a program counter (PC)?
  - Increment it for every instruction… fine, use an adder
  - But only increment it once ready to move on to next instruction
  - That takes a finite amount of time… until then we need to "remember" the current value
- Combinational logic's output is determined from current input
  - But computers have "state" – they remember previous inputs and behave differently based on its history
- Examples of state
  - Registers
  - Memory
  - PC
- Sequential logic's output depends not only on current input, but also the current state
- This lecture will show you how to build sequential logic from gates
  - Key is feedback

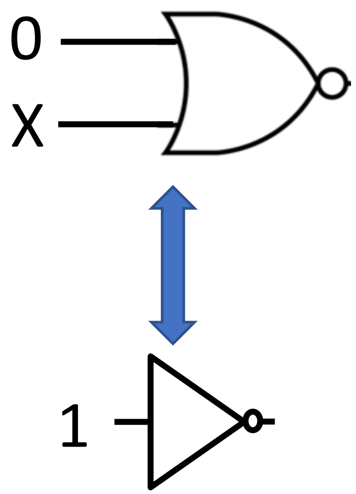
Using feedback to "remember"

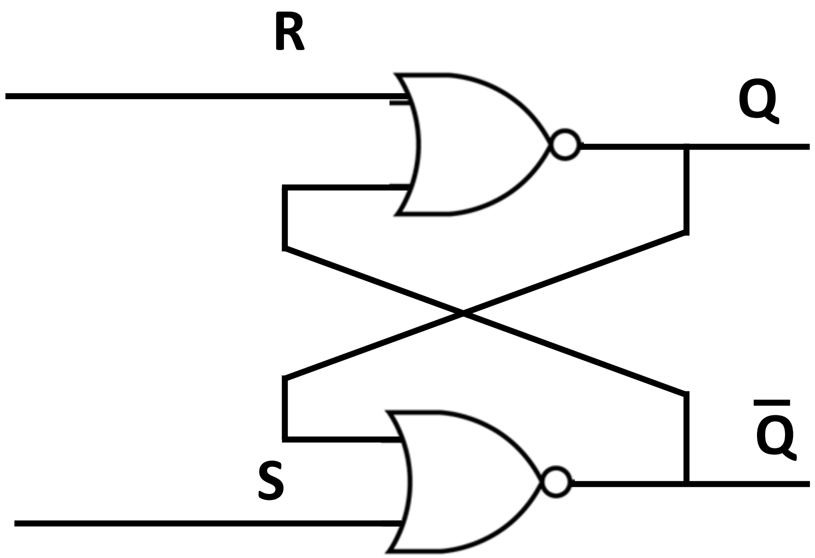


- This remembers its initial value!
- Very basic memory
- What's wrong with this, though?

Let's look at the following circuit

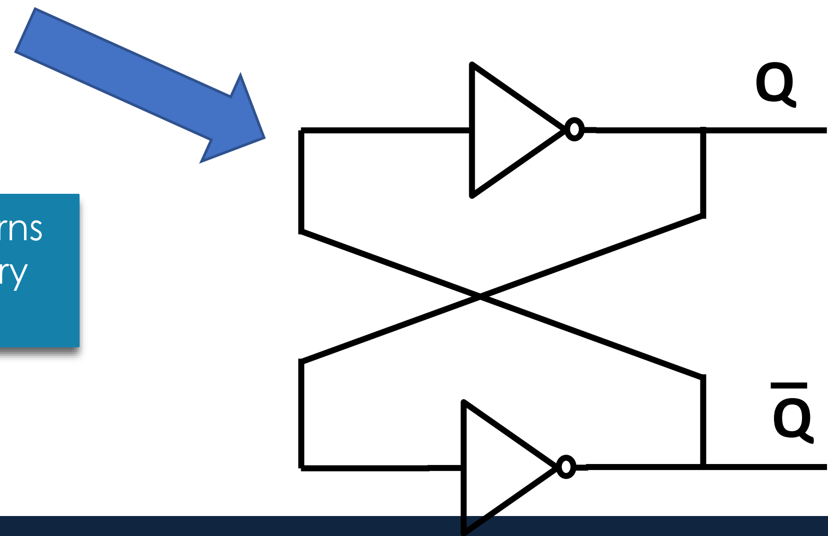
Useful identify: NOR(0,X)= NOT(X)



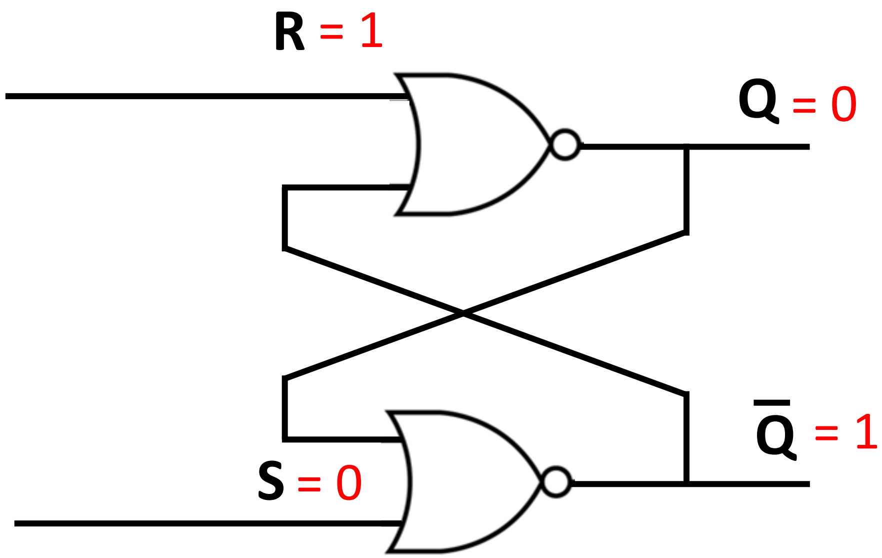


S	R	Q	Q̄
0	0	Q	Q̄

When R=S=0, turns into our memory element!



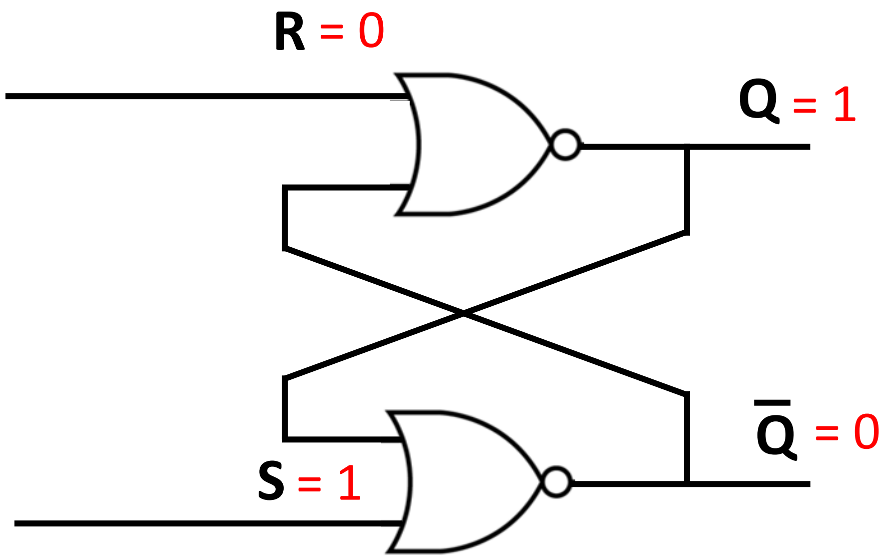
Let's look at the following circuit



S	R	Q	$\overline{Q}$
0	0	Q	Q
0	1	0	1

What is the value of Q if R is 1 and S is 0?

Let's look at the following circuit



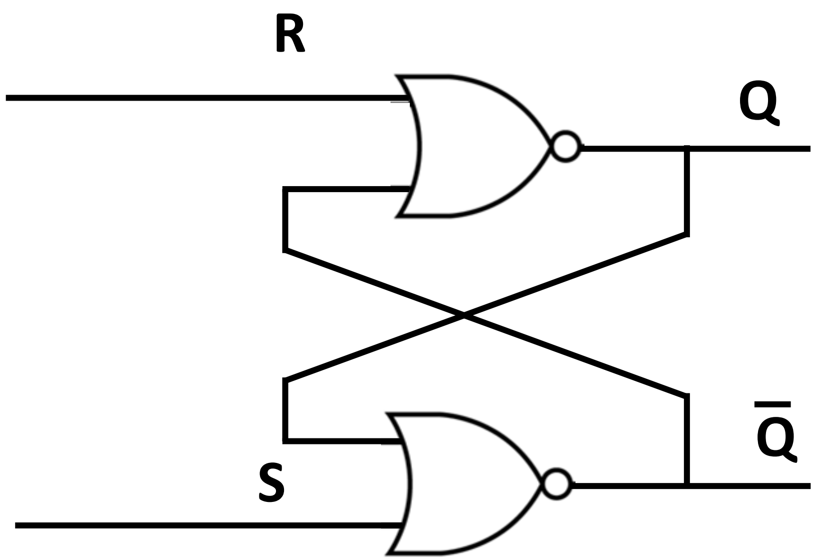
S	R	Q	$\bar{Q}$
0	0	Q	Q
0	1	0	1
1	0	1	0

What is the value of Q if R is 0 and S is 1?

Poll

SR Latch

- So this circuit (an SR latch):
  - "Sets" Q to 1 when S=1 R=0
  - "Resets" Q to 0 when S=0 R=1
  - "Latches" Q when S=0 R=0
  - What about when S=1 R=1 ?

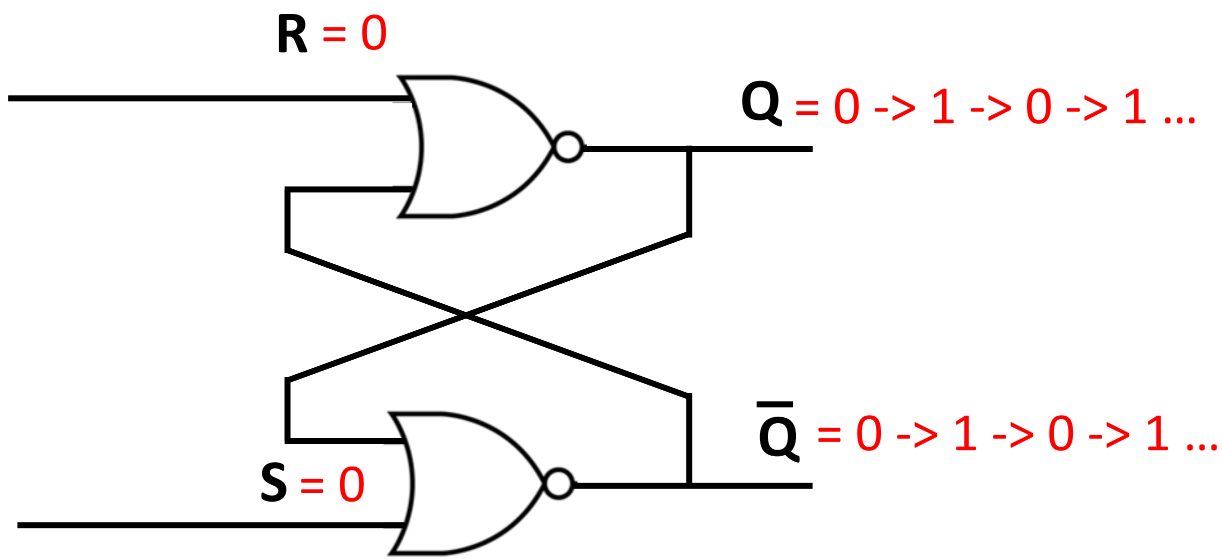


S	R	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
0	1	0	1
1	0	1	0
1	1	0	0

BAD! Why?

SR Latch – Undefined behavior

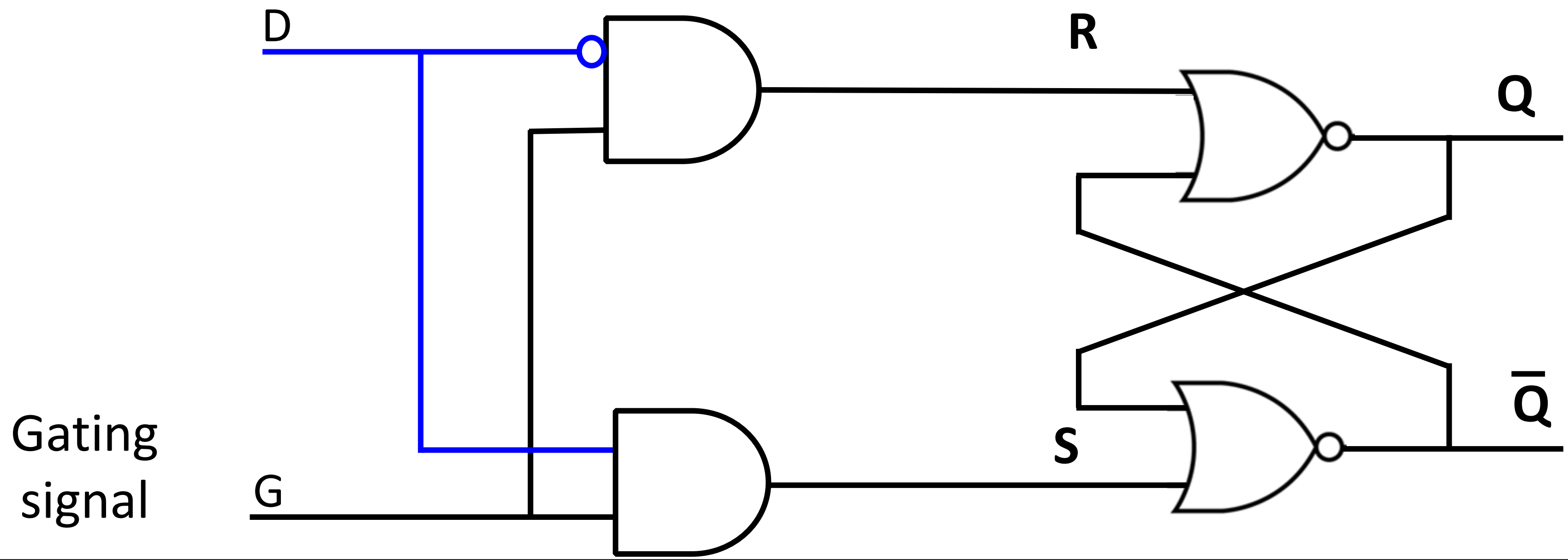
- If S=1, R=1, then Q and it's inverse are both 0
- If inputs then change to S=0, R=0, we get this circuit



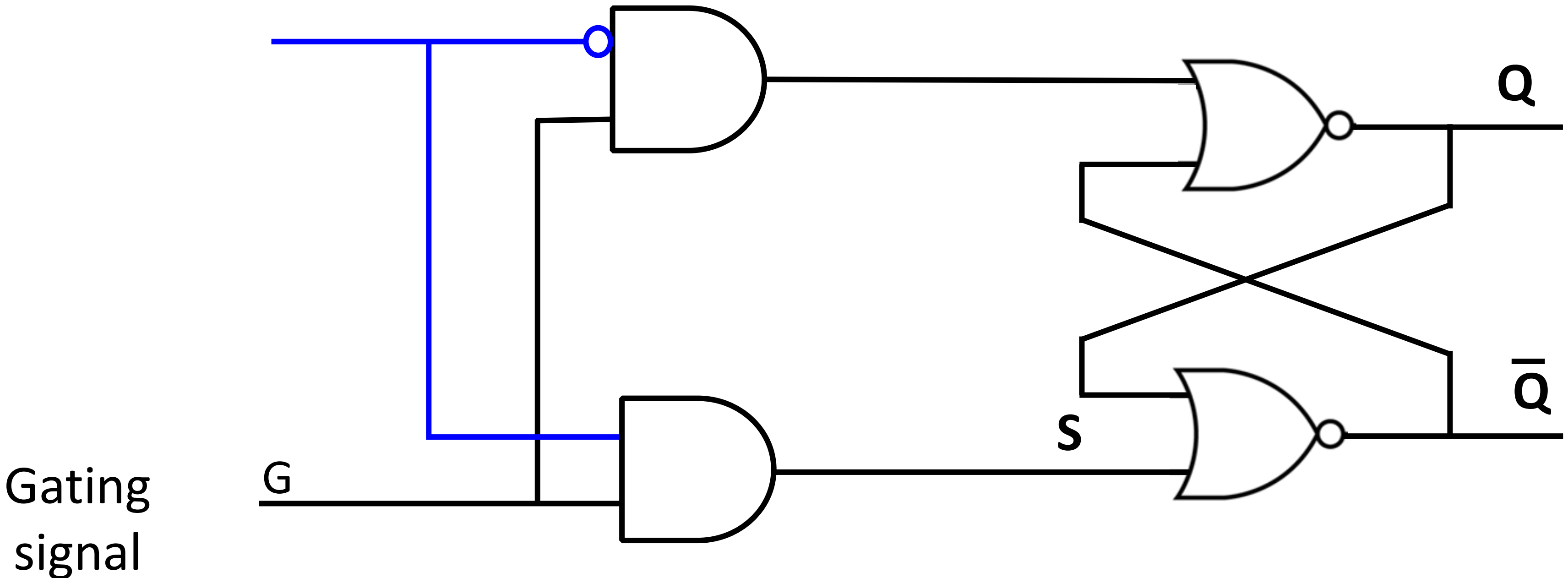
- This is unstable! Output rapidly oscillates between 0 and 1

Improving SR Latch

- SR Latch works great at saving a bit of data…
  - Unless S=R=1, even for a fraction of a second
- Idea: let's prevent that from happening by adding AND gates in front of each input
  - Impossible for R and S to both be 1



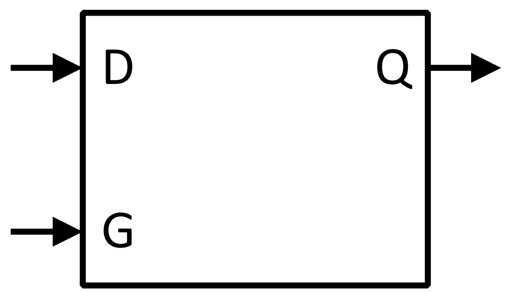
Transparent D Latch



D	G	Q	$\bar{Q}$
0	0	Q	$\bar{Q}$
0	1	0	1
1	0	Q	$\bar{Q}$
1	1	1	0

Next state is set

Set state is retained when gate is low



No invalid states ☺

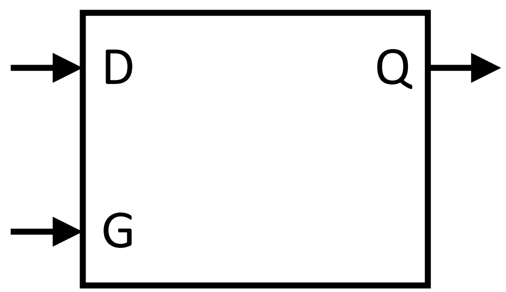
Transparent D Latch

- When G ("gate") is high, Q=D (the latch is "transparent")
- When G is low, Q "latches" to the value of D at that instant and remembers, even if D changes later

D	G	Q	$\bar{Q}$
0	0	Q	$\bar{Q}$
0	1	0	1
1	0	Q	$\bar{Q}$
1	1	1	0

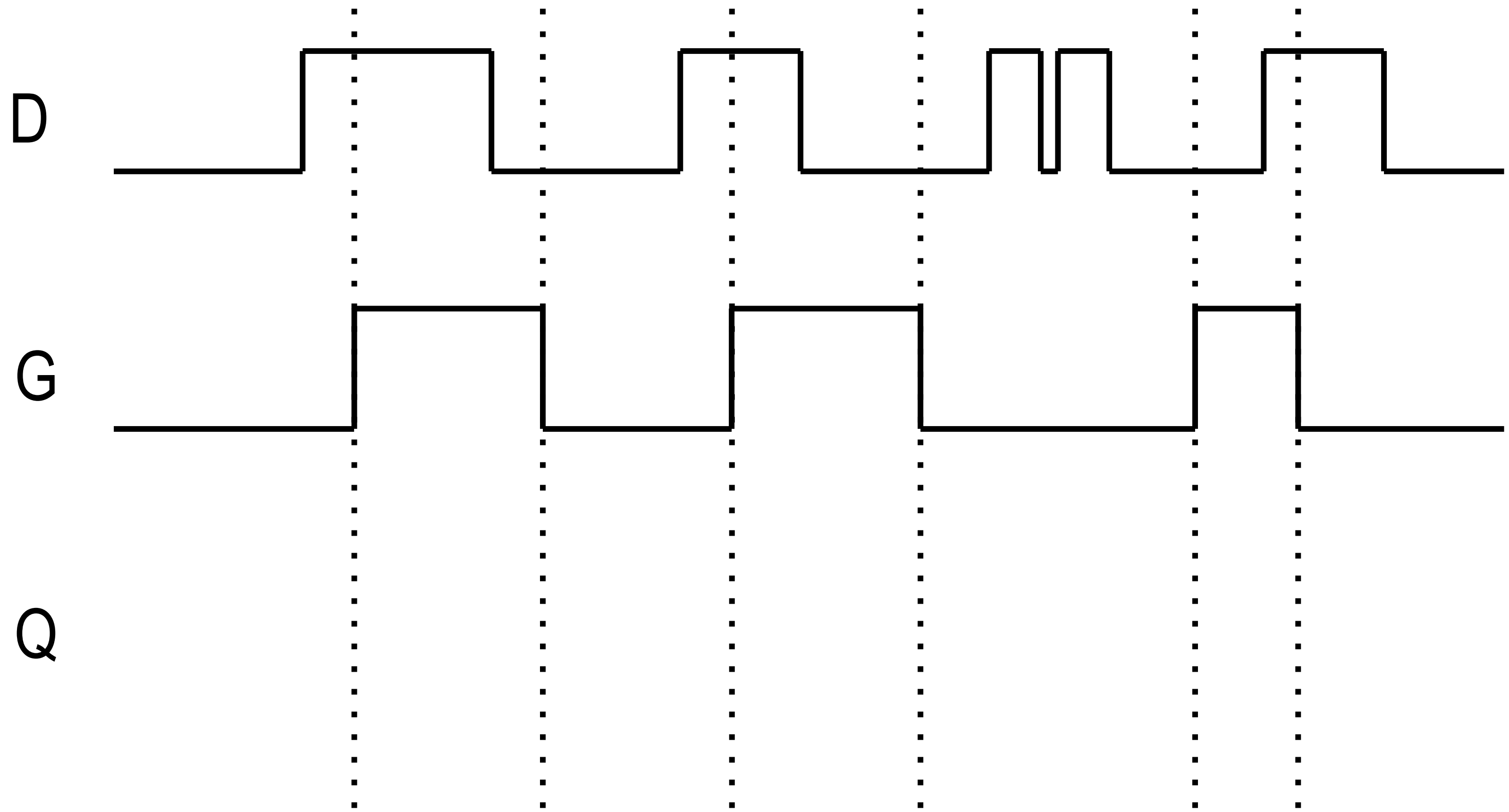
Next state is set

Set state is retained when gate is low

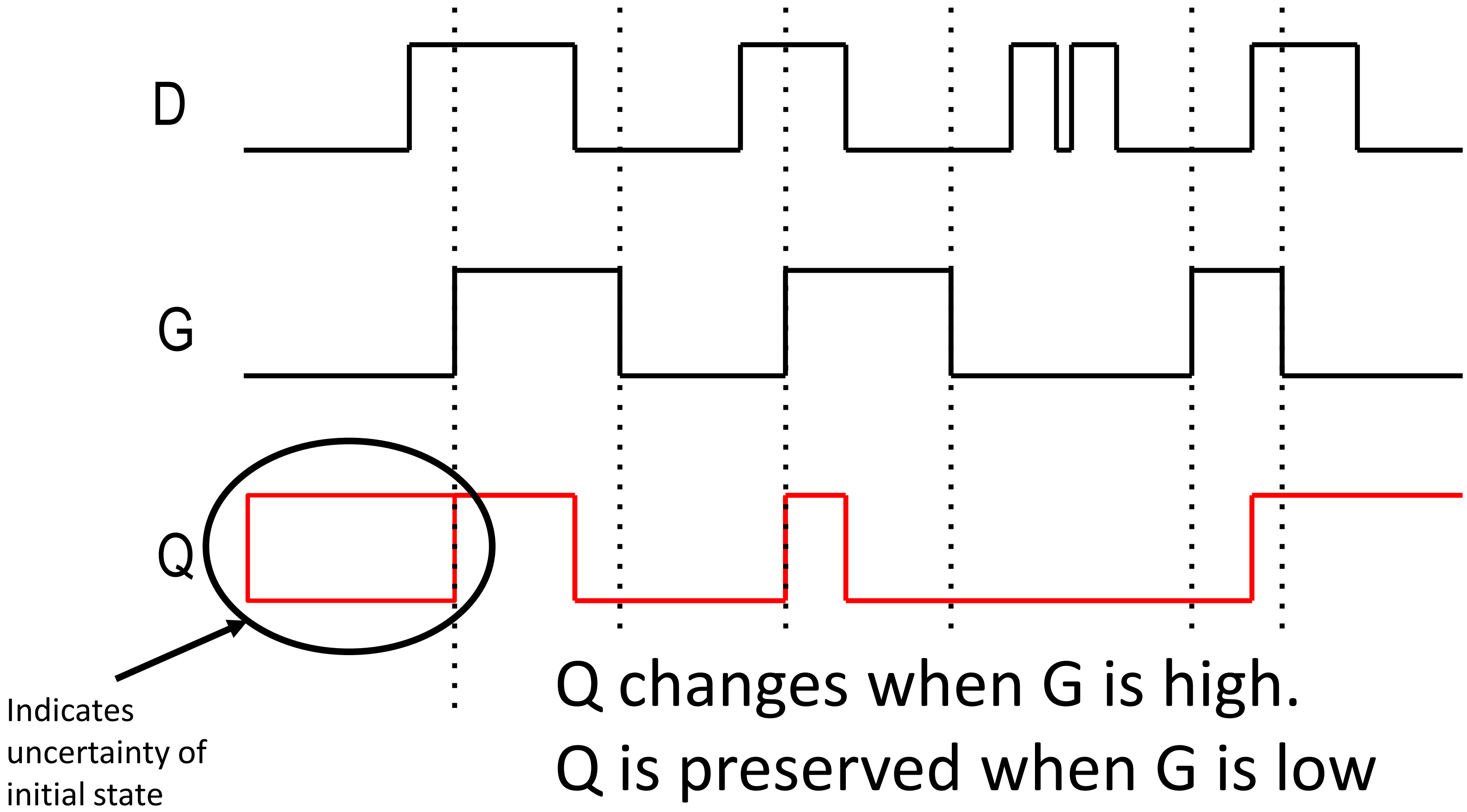


No invalid states ☺

D-Latch Timing Diagram



D-Latch Timing Diagram

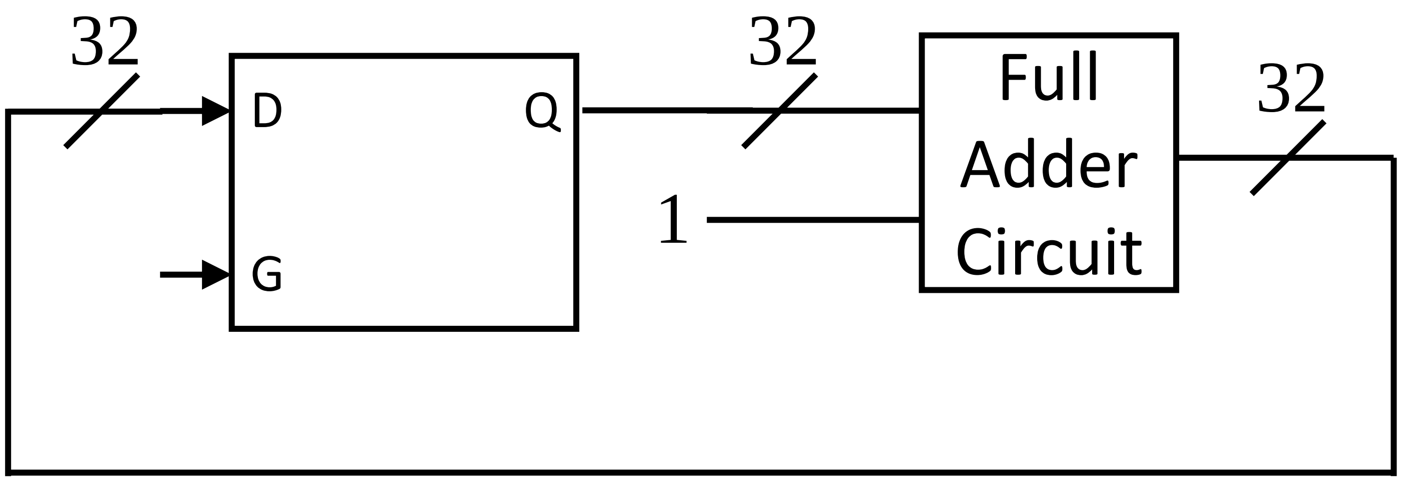


Q changes when G is high.

Q is preserved when G is low

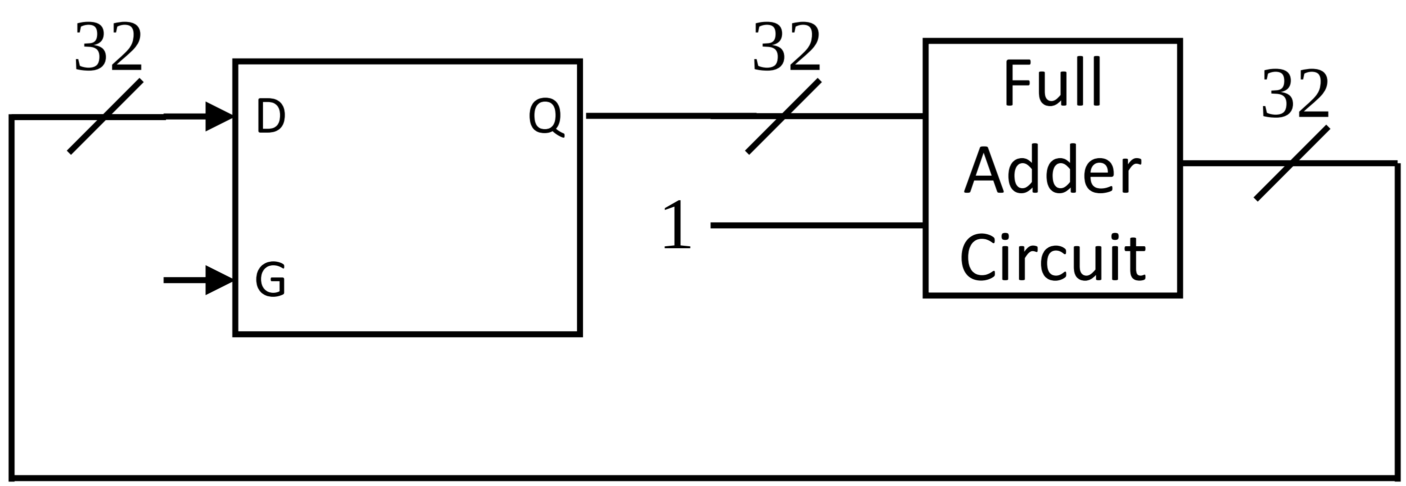
Is D-Latch Sufficient?

- Can we use D-latches to build our PC logic?
- Idea:
  - Use 32 latches to hold current PC, send output Q to memory
  - Also pass output Q into 32-bit adder to increment by 1 (for word-addressable system)
  - Wrap sum around back into D as "next PC"
  - Once ready to execute next instruction, set G high to update



Shortcoming of D-Latch

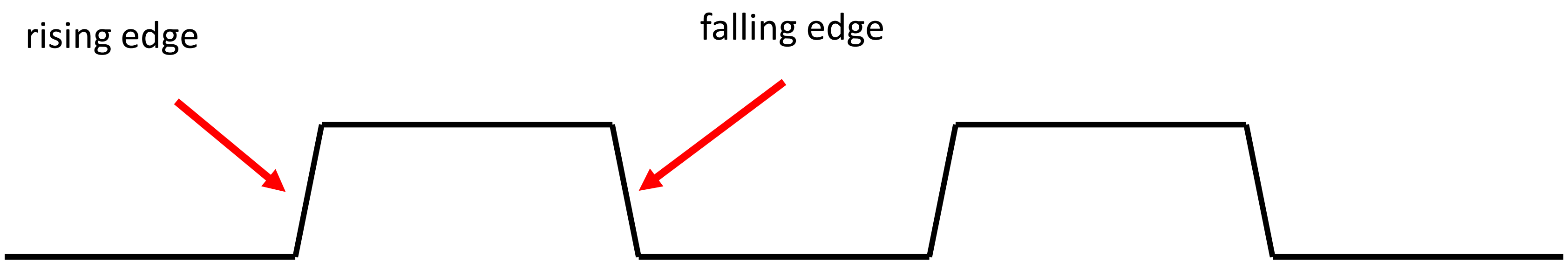
- Problem: G must be set very precisely
  - Set high for too short: latch doesn't have enough time for feedback to stabilize
  - Set high for too long: Signal may propagate round twice and increment PC by 2 (or more)
- Challenging to design circuits with exactly the right durations



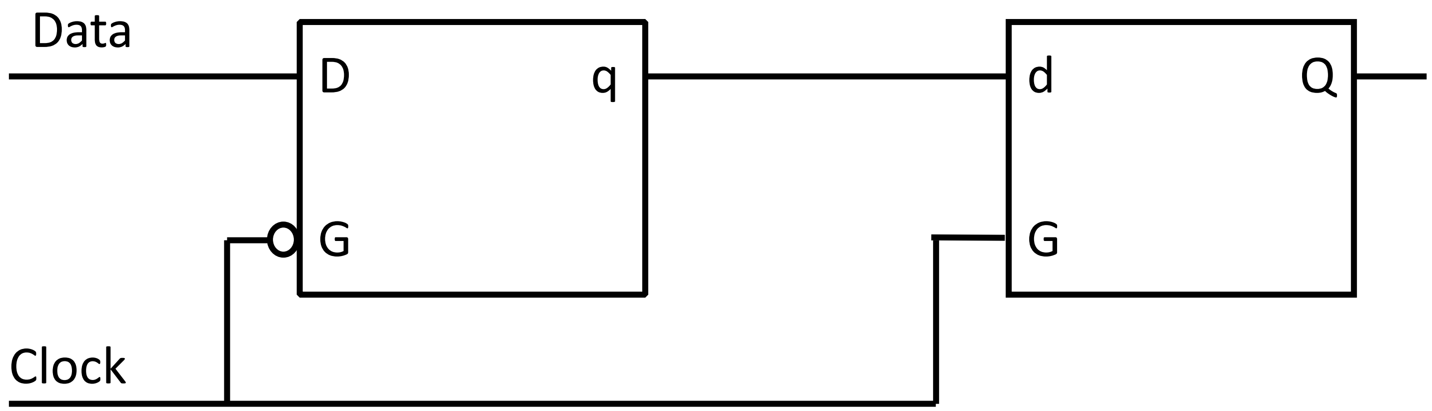
Not just a problem for PC, much of our processor will involve logic like this

Adding a Clock to the Mix

- We can solve this if we introduce a **clock**
  - Alternating signal that switches between 0 and 1 states at a fixed frequency (e.g., 1 GHz)
  - Only store the value the **instant** the clock changes (i.e. the **edge**)



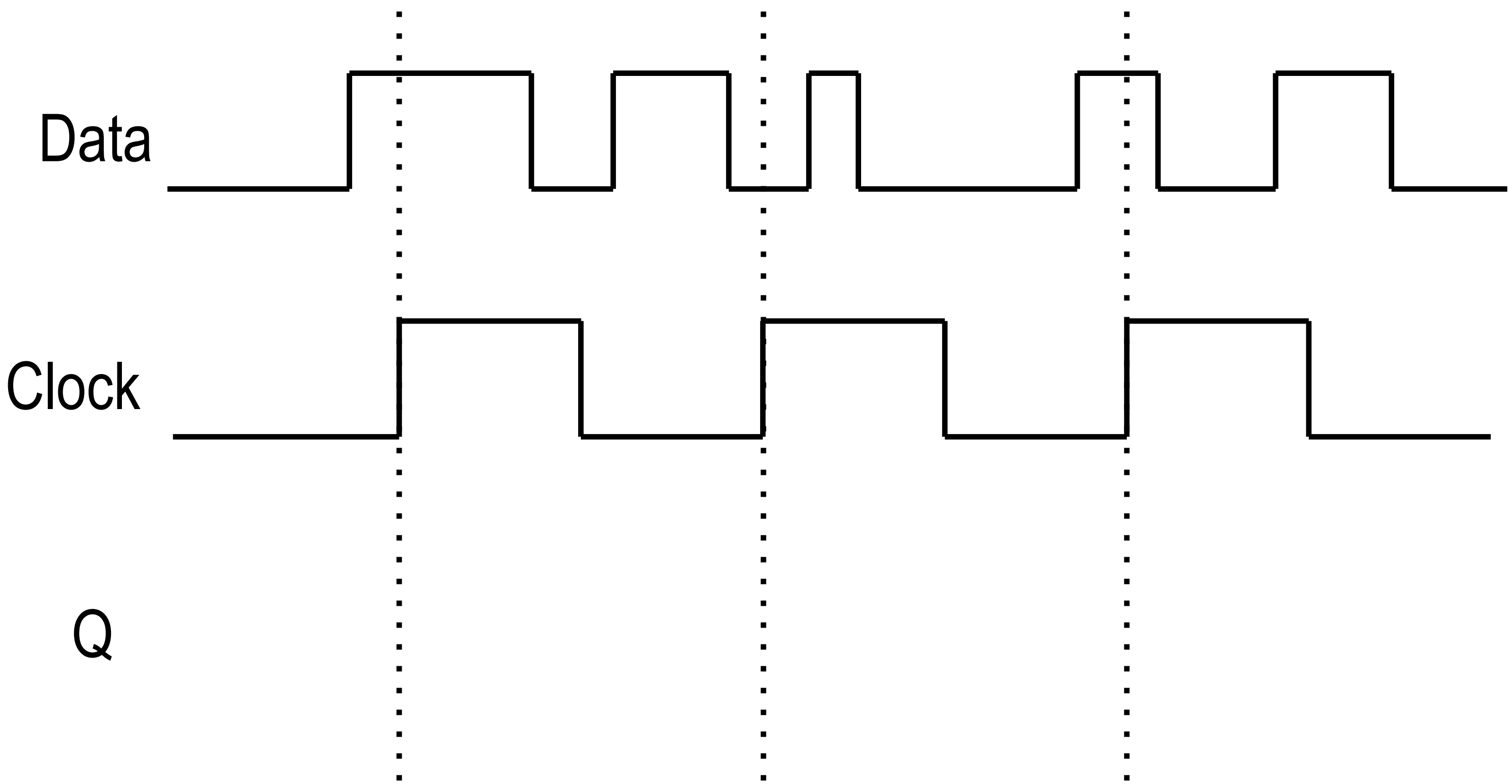
Adding a Clock to the Mix



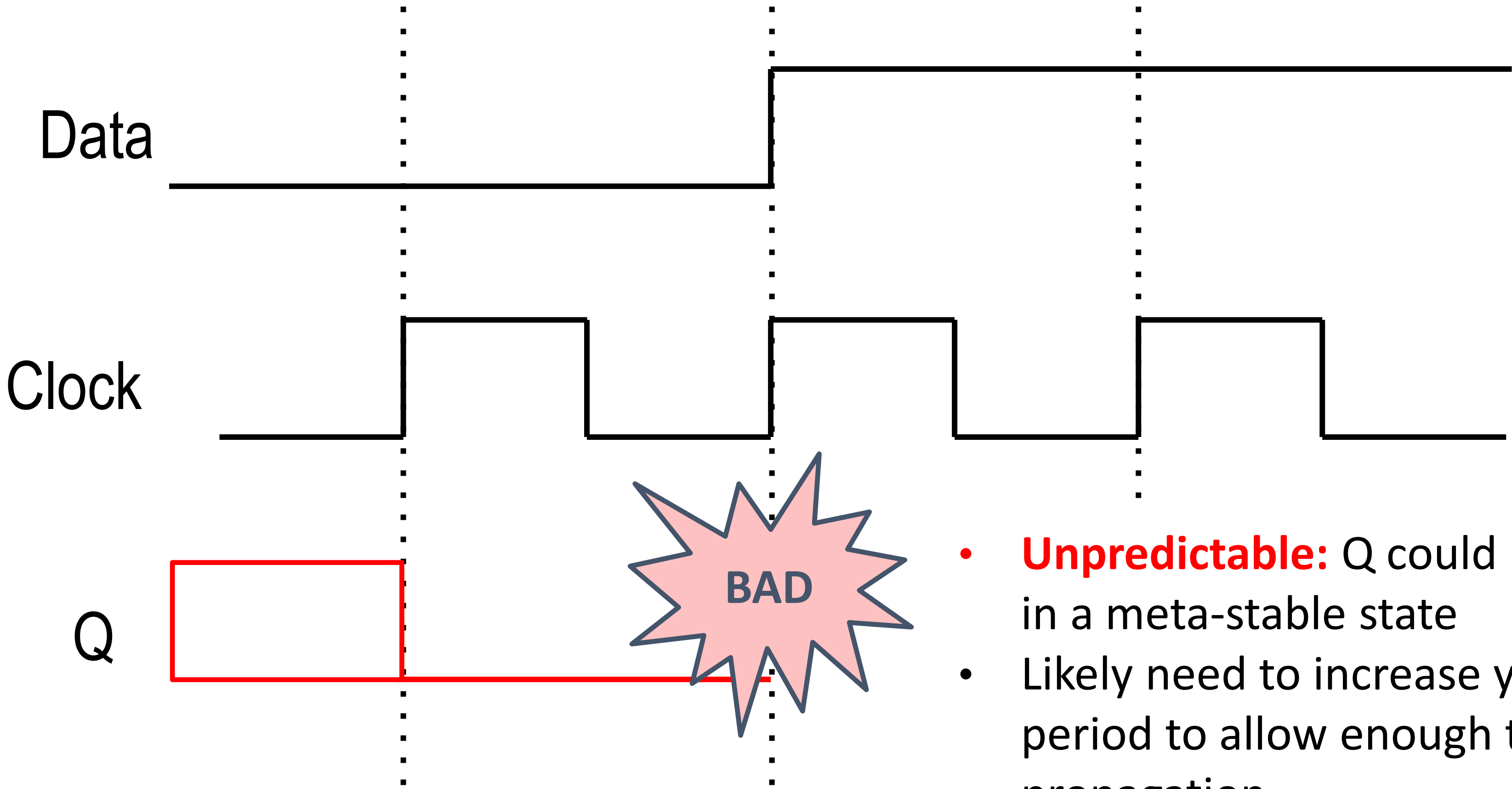
We won't discuss it further here, but this circuit sets Q=D **ONLY** when clock transitions from 0 -> 1

Intuitively, the design works by inverting the Gate signals, so only one passes at a time (like a double set of sliding doors)

D Flip-Flop Timing Diagram

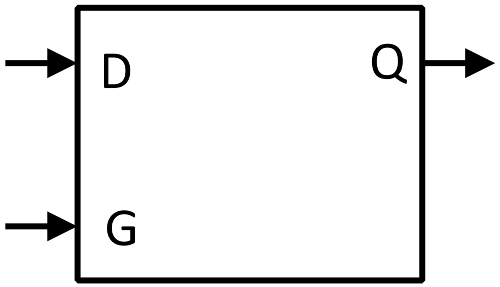


What happens if Data changes on clock edge?

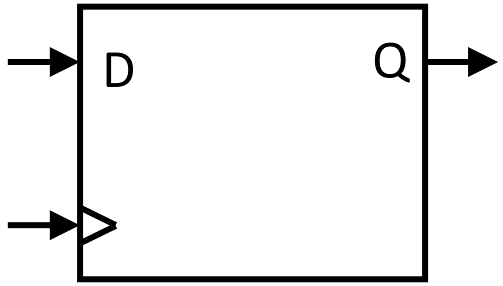


- **Unpredictable:** Q could be high, low or in a meta-stable state
- Likely need to increase your clock period to allow enough time for signal propagation

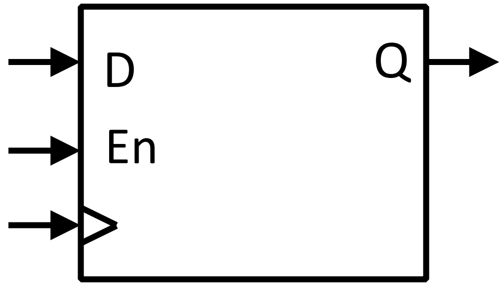
Latches vs Flip-flops



D Latch



D Flip-flop



Enabled D Flip-flop (only updates on clock edge if 'en' is high)

Finite State Machines

- So far we can do two things with gates:
  1. Combinational Logic: implement Boolean expressions
     - Adder, MUX, Decoder, logical operations etc
  2. Sequential Logic: store state
     - Latch, Flip-Flops
- How do we combine them to do something interesting?
  - Let's take a look at implementing the logic needed for a vending machine
  - Discrete states needed: remember how much money was input
    - Store sequentially
  - Transitions between states: money inserted, drink selected, etc
    - Calculate combinationally or with a control ROM *(more on this later)*

Input and Output

- Inputs:
  - Coin trigger
  - Refund button
  - 10 drink selectors
  - 10 pressure sensors
    - Detect if there are still drinks left
- Outputs:
  - 10 drink release latches
  - Coin refund latch

Operation of Machine

- Accepts quarters only
- All drinks are $0.75
- Once we get the money, a drink can be selected
- If they want a refund, release any coins inserted

- No free drinks!
- No stealing money.

Building the controller

- Finite State Machine
  - An abstract model describing how the machine should be have under a fixed set of circumstances (i.e. finite states)
  - Remember how many coins have been put in the machine and what inputs are acceptable

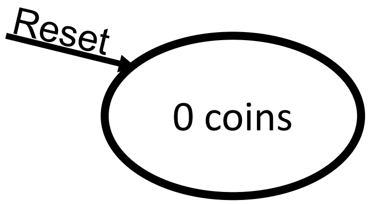
- Read-Only Memory (ROM)
  - A cheaper way of implementing combinational logic
  - Define the outputs and state transitions

- Custom combinational circuits
  - Reduce the size (and therefore cost) of the controller

Finite State Machines

- A Finite State Machine (FSM) consists of:
  - K states: S = {s1, s2, … ,sk}, s1 is initial state
  - N inputs: I = {i1, i2, … ,in}
  - M outputs: O = {o1, o2, … ,om}
  - Transition function T(S,I) mapping each current state and input to next state
  - Output Function P(S) or P(S,I) specifies output
    - P(S) is a Moore Machine
    - P(S,I) is a Mealy Machine

FSM for Vending Machine



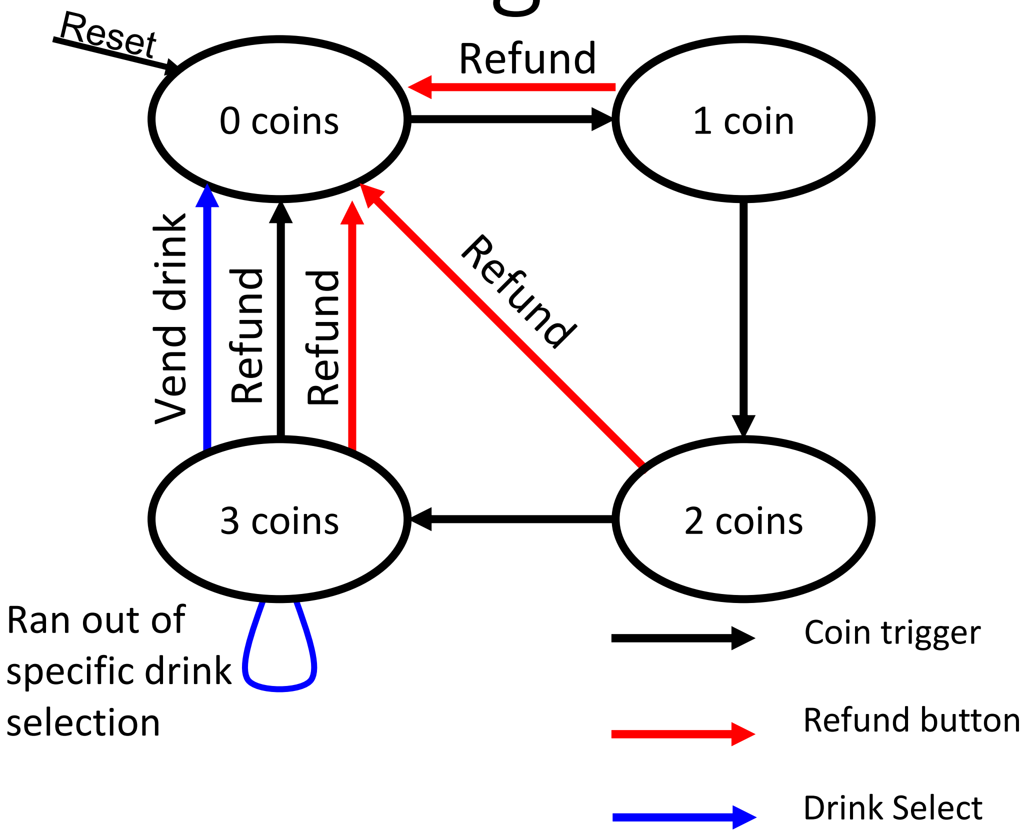
Coin trigger

Refund button

Drink Select

FSM for Vending Machine

Poll: How many flip-flops would we need to remember which state we're in?



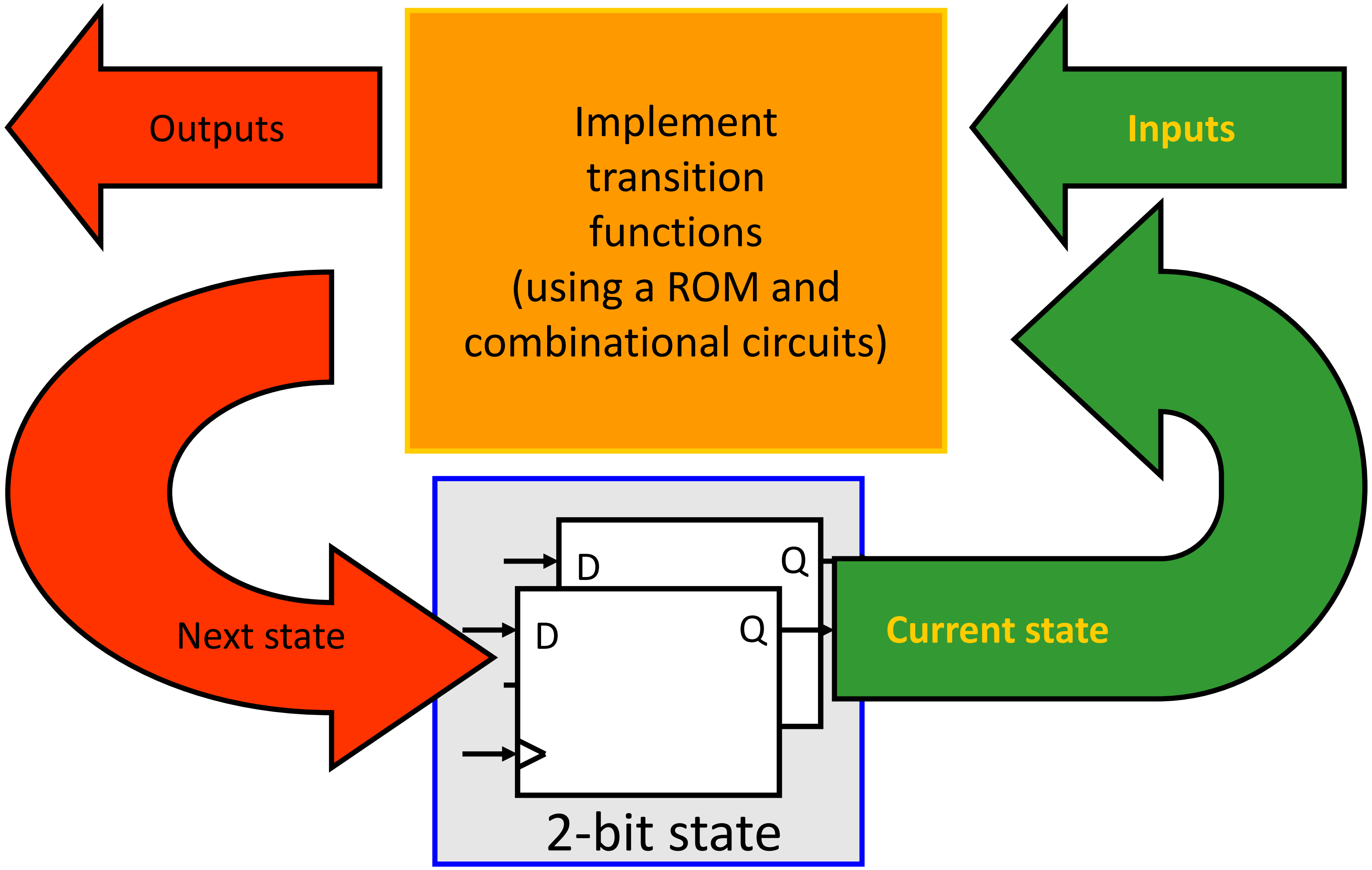
Is this a Mealy or Moore Machine?

This is Mealy: Mealy output is based on current state *AND* input

Poll: **Mealy or Moore?**

Implementing an FSM

Poll: How cheaply do you think we can build one of these controllers?



Implementing an FSM

- Let's see how cheap we can build this vending machine controller!
- Jameco.com sells electronic chips we can use
  - D-Flip-flops: $3, includes several in one package

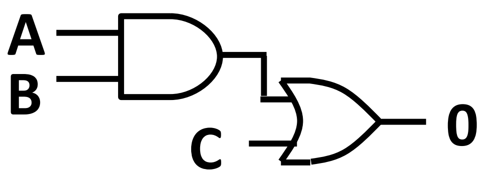
- For custom combinational circuits, would need to design and send to a fabrication facility
  - Thousands or millions of dollars!!
  - Alternative?

Implementing Combinational Logic

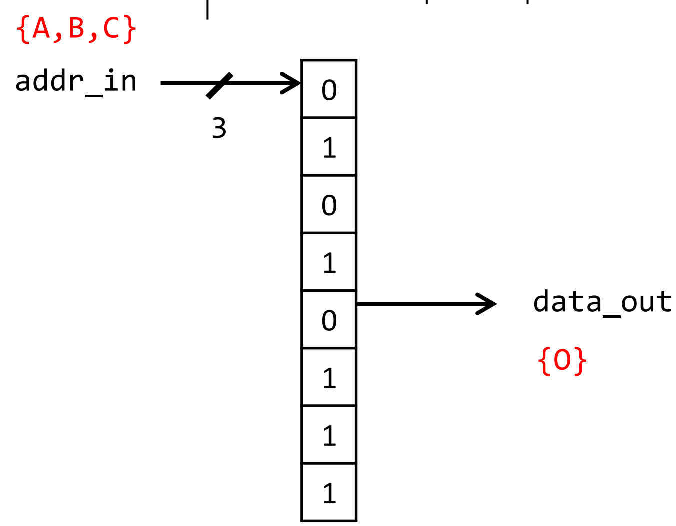
A	B	C	O
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

If I have a truth table:

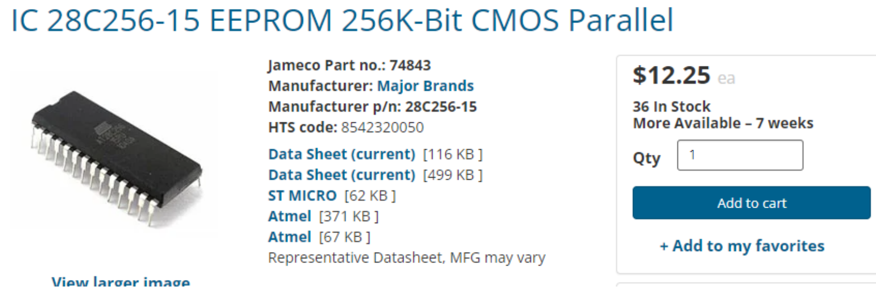
- Either implement this using combinational logic:



- ...or literally just store the entire truth table in a memory and "index" it by treating the input as a number!
  - Can be implemented cheaply using "Read Only Memories", or "ROMS"

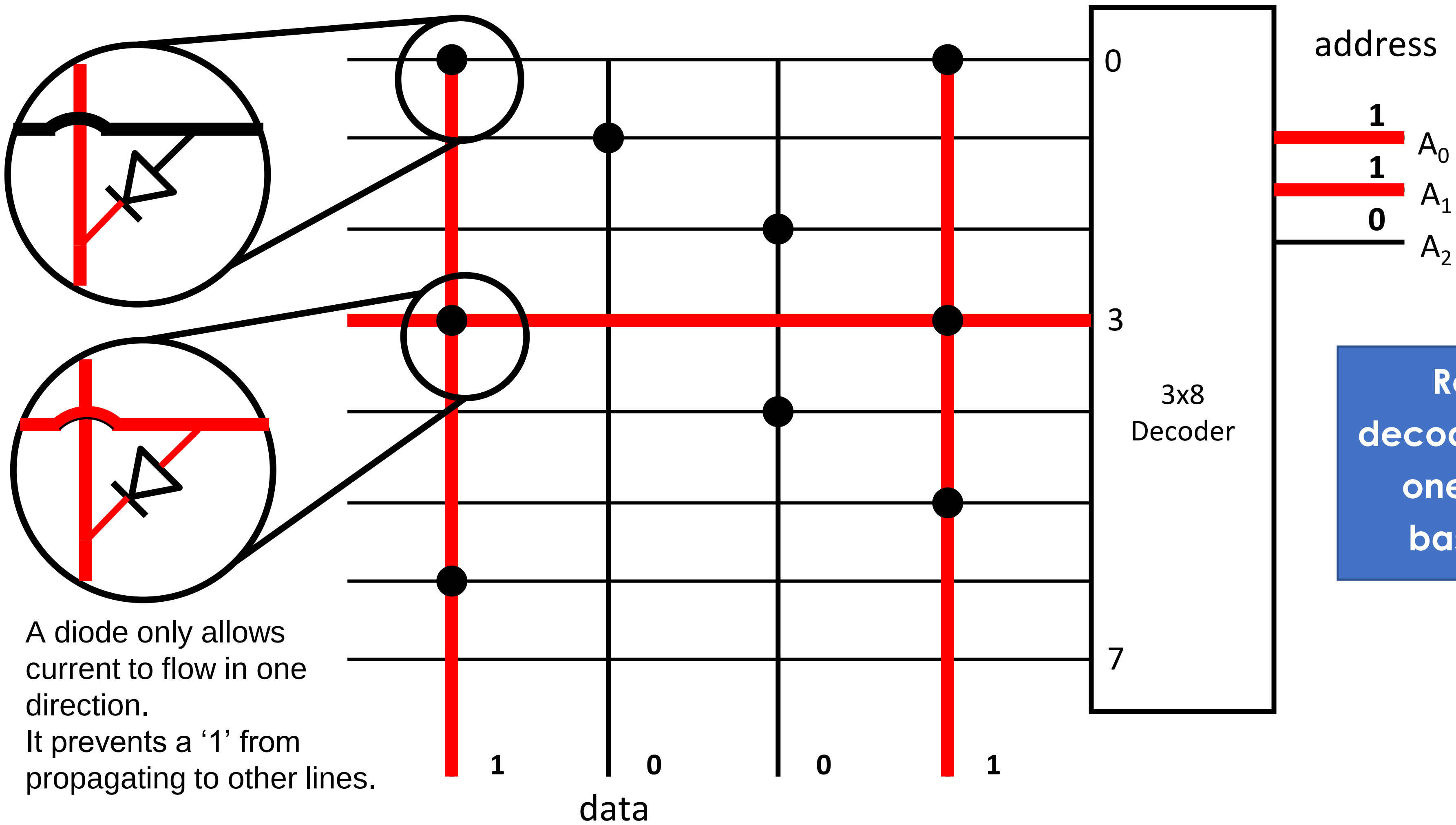


ROMs and PROMs



- Read Only Memory (ROM)
  - Array of memory values that are constant
  - Non-volatile (doesn’t need constant power to save values)
- Programmable Read Only Memory
  - Array of memory values that can be written exactly once
- Electronically Erasable PROM (EEPROM)
  - Can write to memory, deploy in field
  - Use special hardware to reset bits if need to update
- 256 KBs of EEPROM costs ~$10 on Jameco
  - Much better then spending thousands on design costs unless we're gonna make **tons** of these

8-entry 4-bit ROM



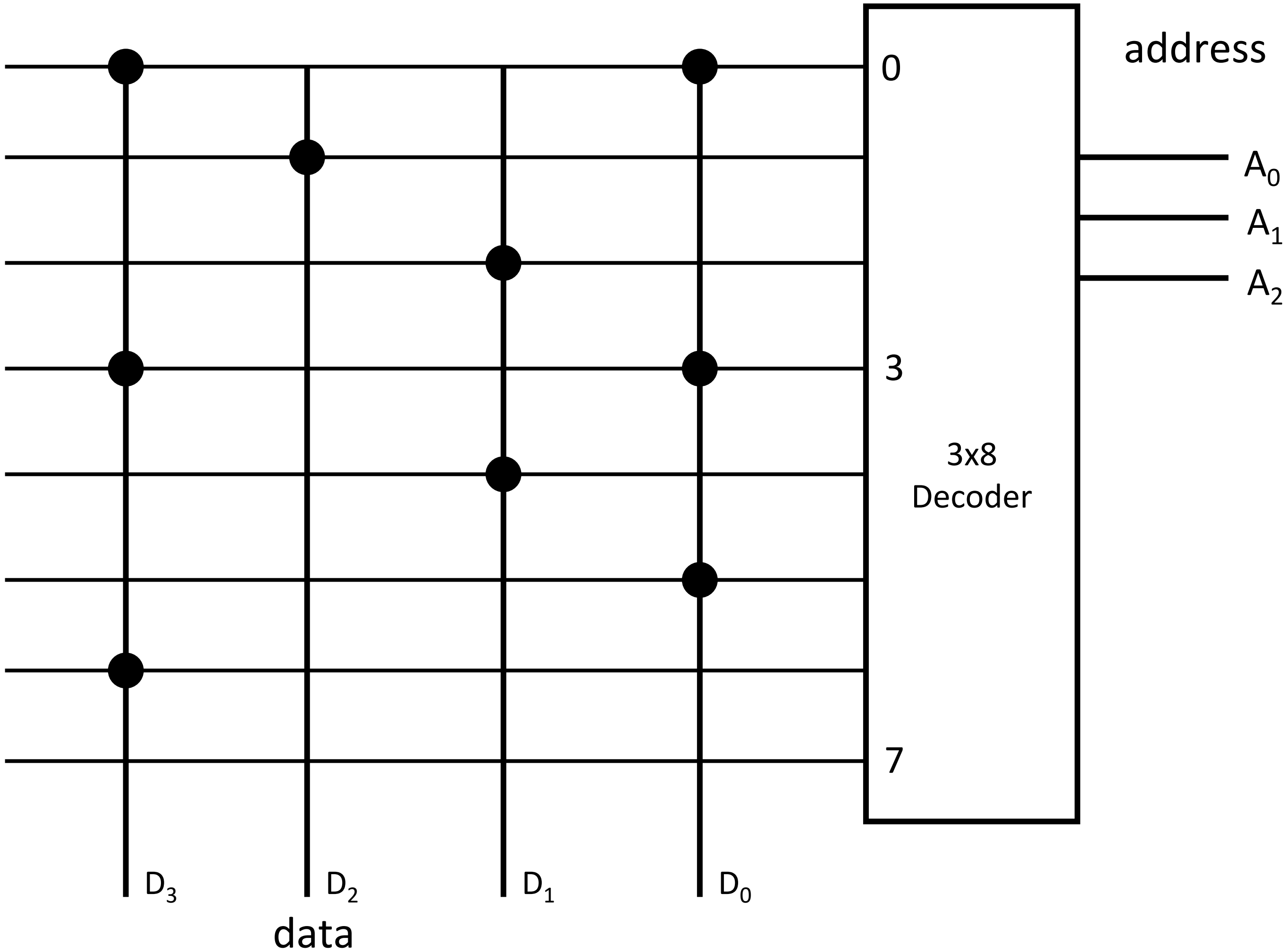
- A diode only allows current to flow in one direction.
- It prevents a ‘1’ from propagating to other lines.

Reminder: A decoder sets exactly one output high based on input

8-entry 4-bit ROM

Input	Output
000	
001	
010	
011	
100	
101	
110	
111	

This ROM corresponds to this truth table

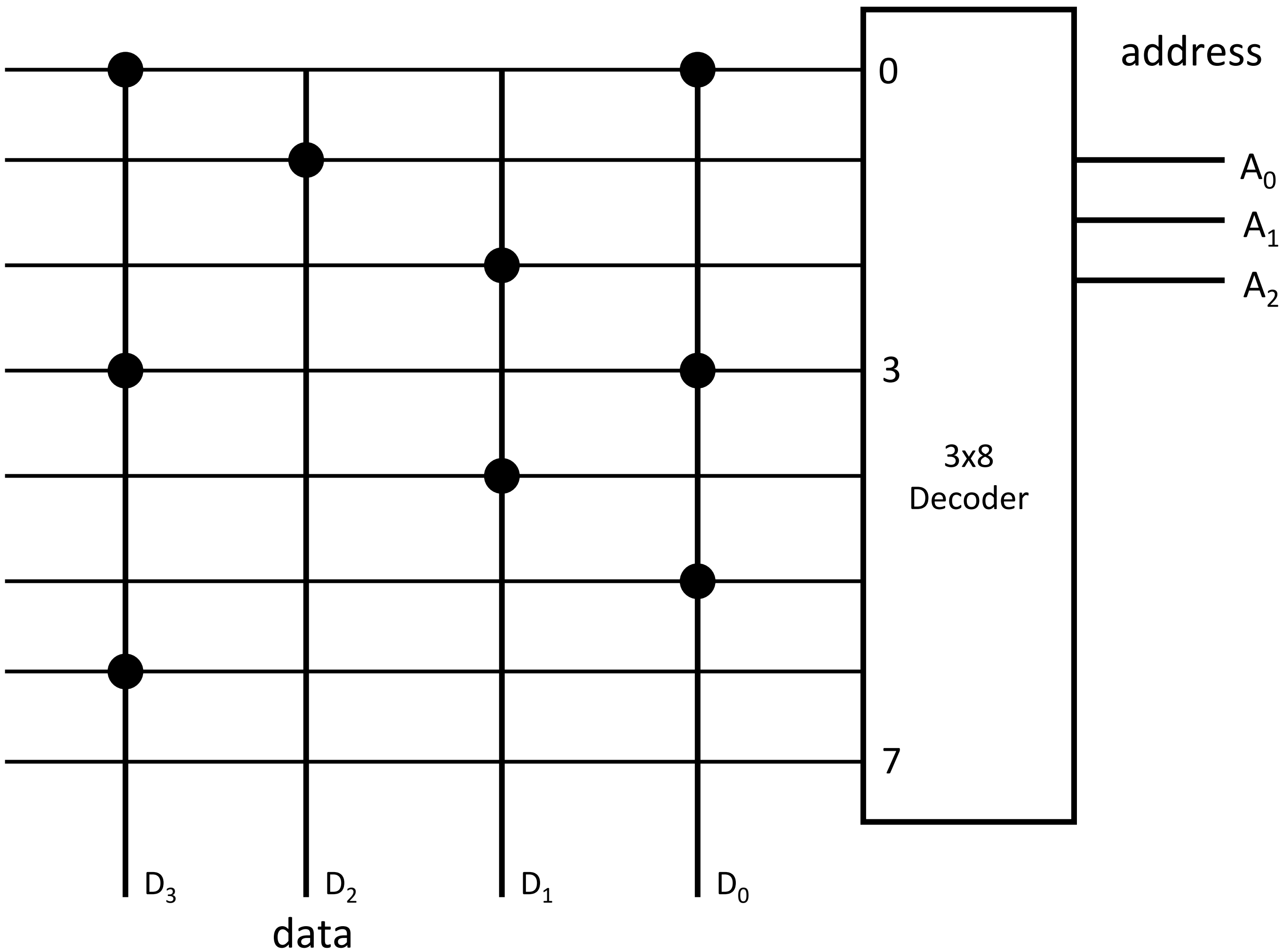


8-entry 4-bit ROM

Poll: What's the formula for size of ROM needed?

Input	Output
000	1001
001	0100
010	0010
011	1001
100	0010
101	0001
110	1000
111	0000

This ROM corresponds to this truth table



Aside: Other Memories

- Static RAM (random access memory)
  - Built from sequential circuits
    - Takes 4-6 transistors to store 1 bit
    - Fast access (< 1 ns access possible)

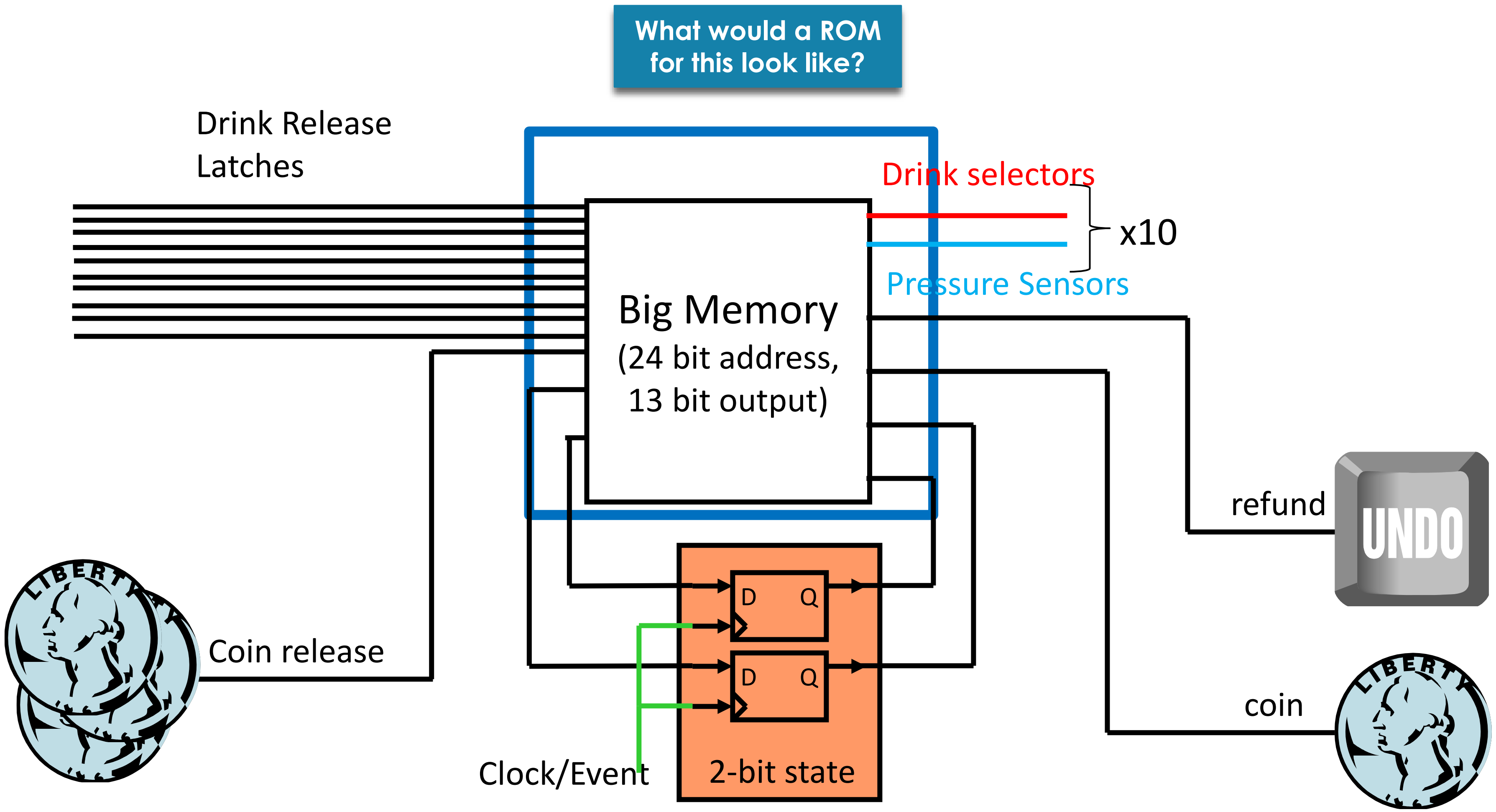
- Dynamic RAM
  - Built using a single transistor and a capacitor
    - 1’s must be refreshed often to retain value
    - Slower access than static RAM
    - Much more dense layout than static RAM

- Both require constant power, or they will lose their data (i.e. are **volatile**)
- These will be used to build computer memory hierarchies (later in class)

Implementing Combinational Logic

- Custom logic
  - Pros:
    - Can optimize the number of gates used
  - Cons:
    - Can be expensive / time consuming to make custom logic circuits
- Lookup table:
  - Pros:
    - Programmable ROMs (Read-Only Memories) are very cheap and can be programmed very quickly
  - Cons:
    - Size requirement grows exponentially with number of inputs (adding one just more bit **doubles** the storage requirements!)

Controller Design So far



ROM for Vending Machine

Size of ROM is (# of ROM entries * size of each entry)

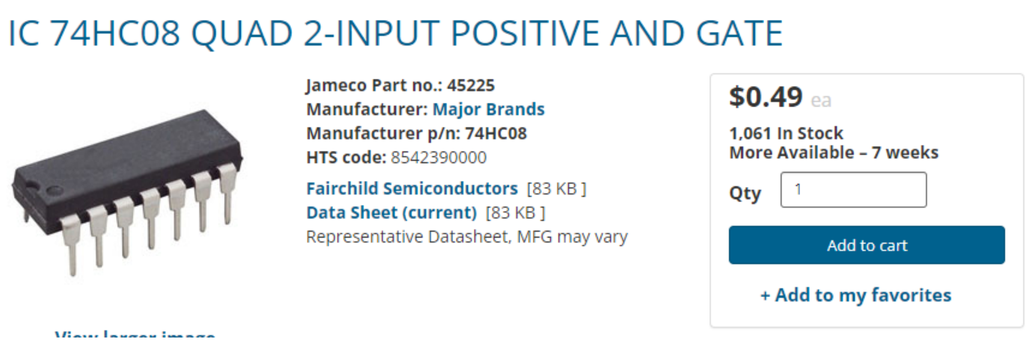
- # of ROM entries = $2^{\text{input\_size}} = 2^{24}$
- Size of each entry = output size = 13 bits

We need 2^{24} entry, 13 bit ROM memories

- **218,103,808 bits of ROM (26 MB)**
- Biggest ROM I could find on Jameco was 4 MB @ $6
  - Need 7 of these at $42??
- Let's see if we can do better

Reducing the ROM needed

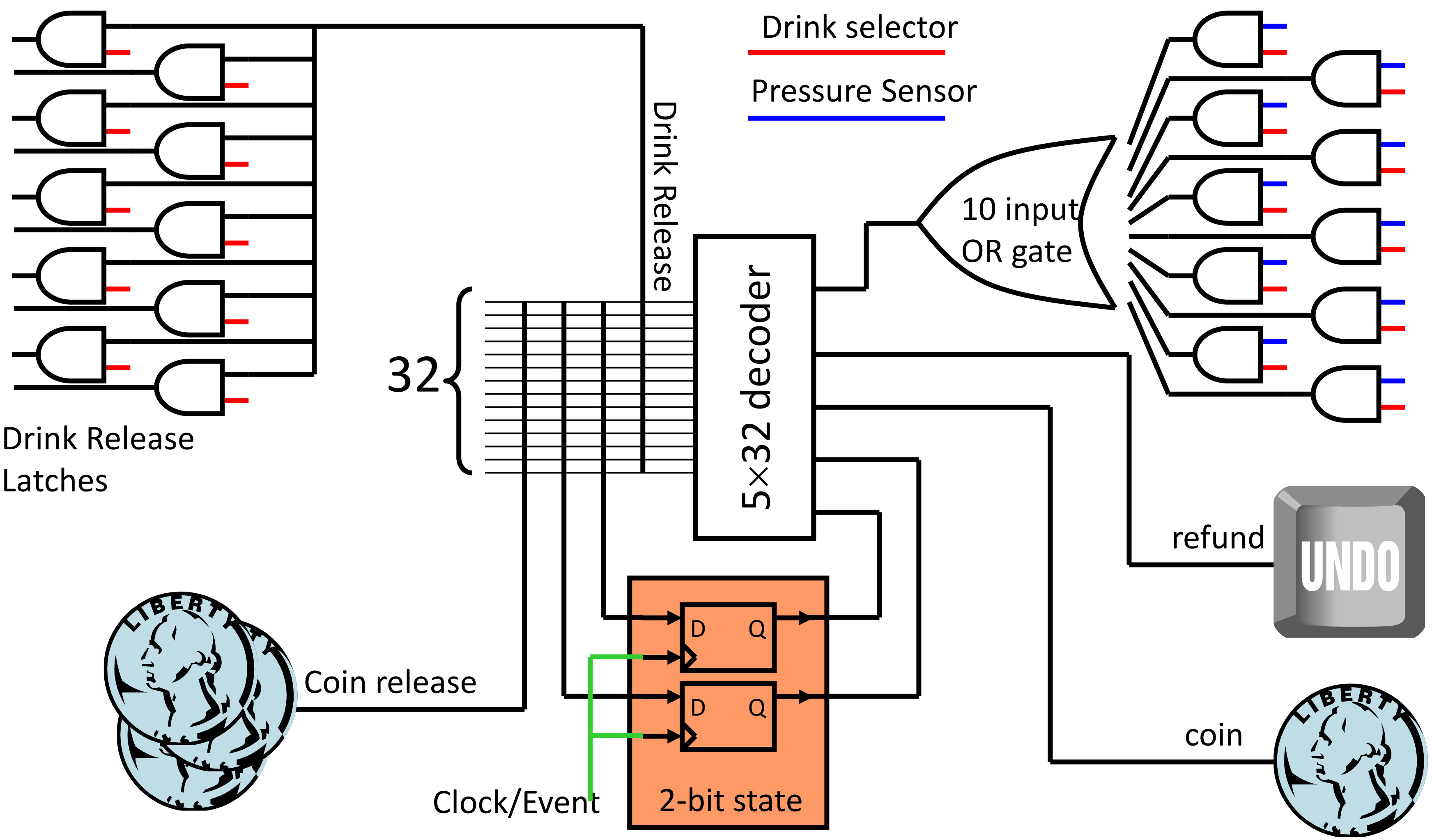
- Idea: let's do a hybrid between combinational logic and a lookup table
  - Use basic hardware (AND / OR) gates where we can, and a ROM for everything more complicated
  - AND / OR gates are mass producible & cheap!
    - ~$0.15 each on Jameco



Reducing the ROM needed

- Observation: overall logic doesn't really need to distinguish between **which** button was pressed
  - That's only relevant for choosing **which** latch is released, but overall logic is the same
- Replace 10 selector inputs and 10 pressure inputs with a **single** bit input (drink selected)
  - Use drink selection input to specify which drink release latch to activate
  - Only allow trigger if pressure sensor indicates that there is a bottle in that selection. (10 2-bit ANDs)

Putting it all together



Total cost of our controller

- Now:
  - 2 current state bits + 3 input bits (5 bit ROM address)
  - 2 next state bits + 2 control trigger bits (4 bit memory)
  - $2^5 \times 4 = 128$ bit ROM
    - 1-millionth size of our 26 MB ROM 😬

- Total cost on Jameco:
  - Flip-flops to store state: $3
  - ROM to implement logic: $3
  - AND/OR gates: $5
  - **Total: $11**
- Could probably do a lot cheaper if we buy in bulk

Next Time

- Introduce first processor implementation
- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah