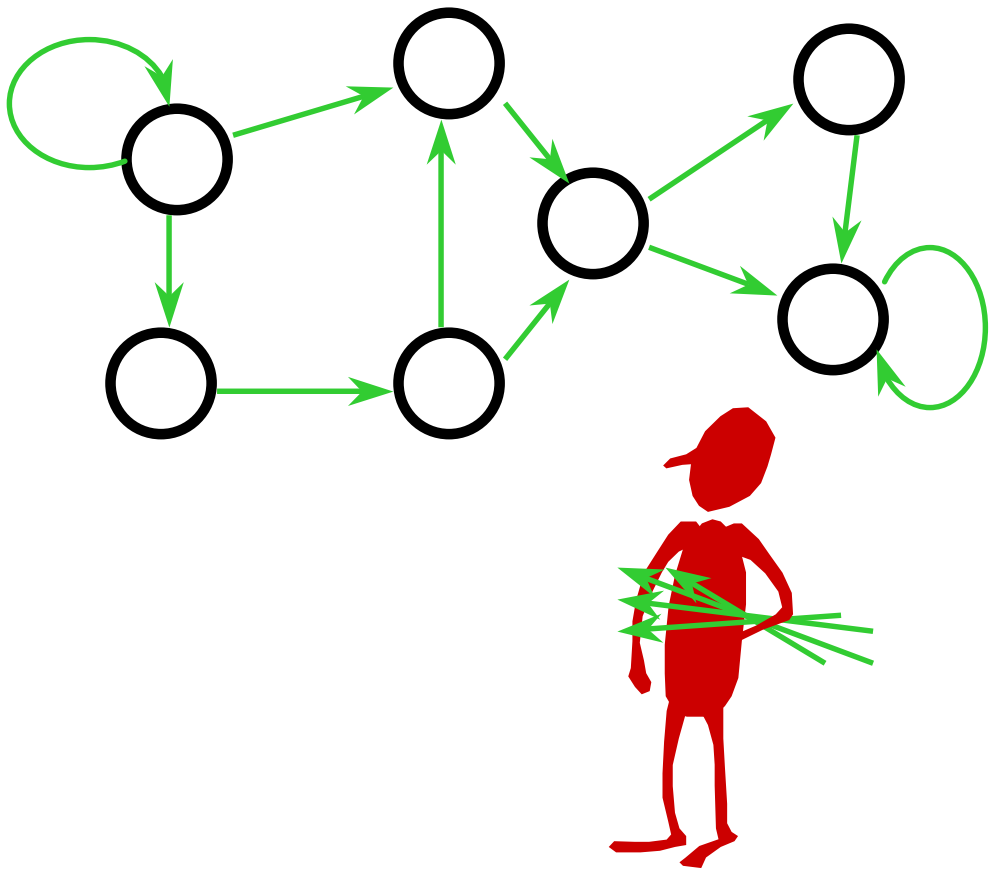
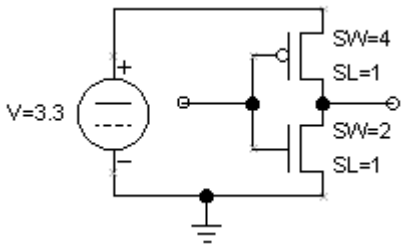
Sequential Logic

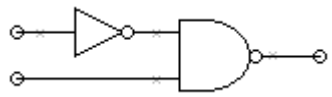


Handouts: Lecture Slides

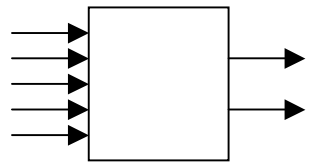
Roadmap so far…



Fets & voltages



Logic gates



Combinational logic circuits

Sequential Logic

Voltage-based encoding
- V_{OL}, V_{IL}, V_{IH}, V_{OH}

Combinational contract:
- discrete-valued inputs
- complete in/out spec.
- static discipline
  t_{CD} and t_{PD}

Acyclic connections
Composable blocks
Design:
- truth tables
- sum-of-products
- simplification
- muxes, ROMs, PLAs

Storage & state
Dynamic discipline
Finite-state machines
Metastability
Throughput & latency
Pipelining

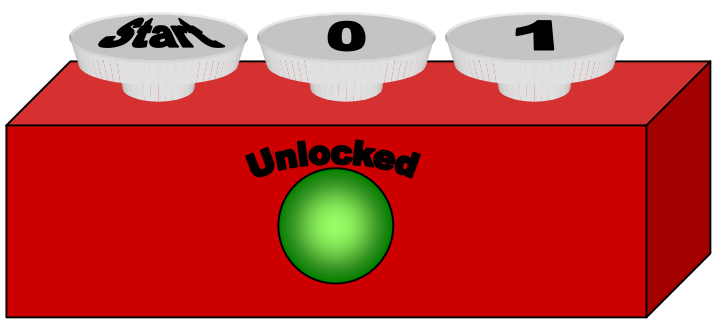
Our motto: Spare me the details!

Why Sequential Logic?

We'll use sequential logic when we need organize the solution to some design problem as a *sequence* of steps:

How to open digital combination lock w/ 3 buttons ("start", "0" and "1"):

Step 1: press "start" button
Step 2: press "0" button
Step 3: press "1" button
Step 4: press "1" button
Step 5: press "0" button



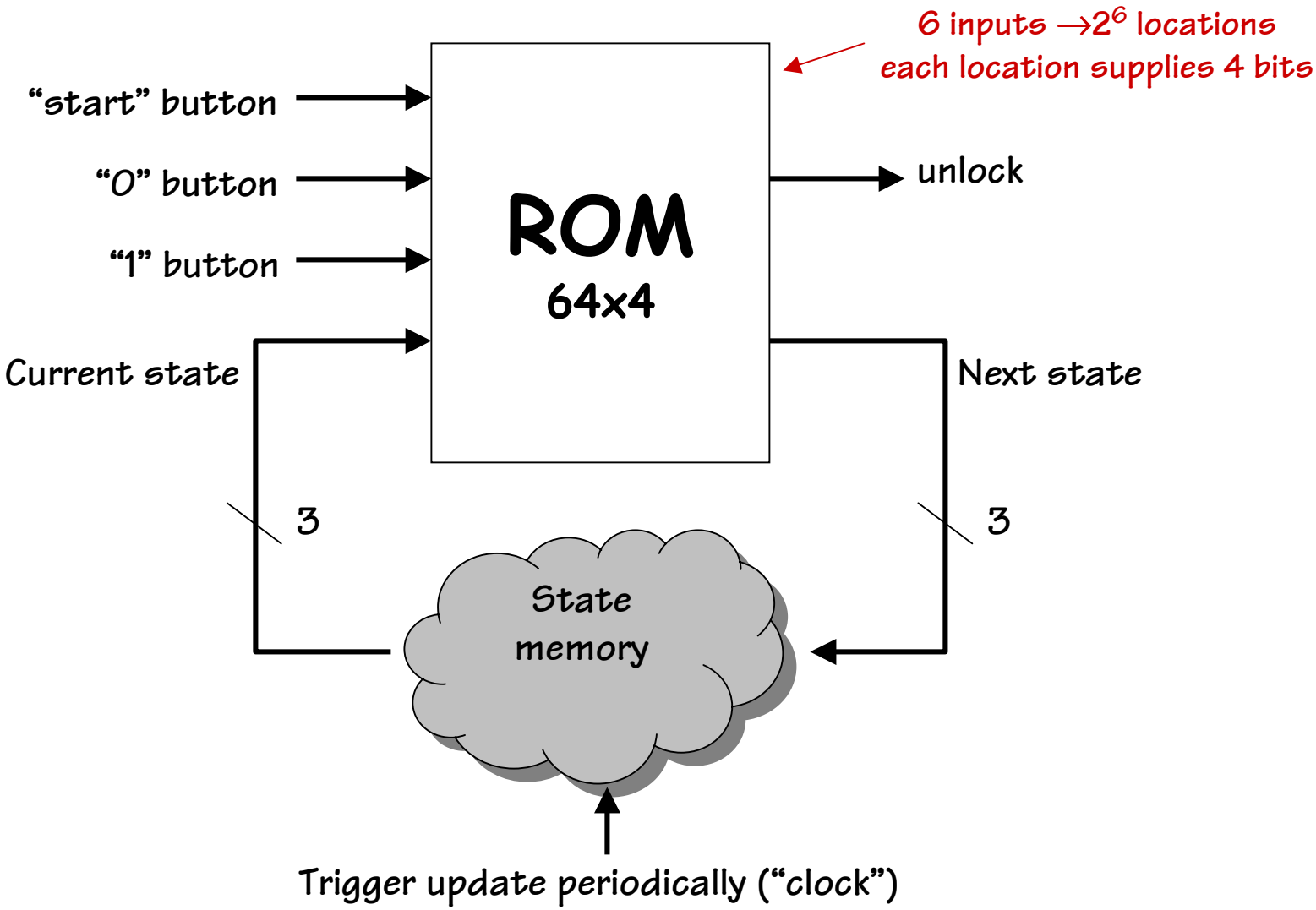
Information remembered between steps is called state. Might be just what step we're on, or might include results from earlier steps we'll need to complete a later step.

Implementing a "State Machine"

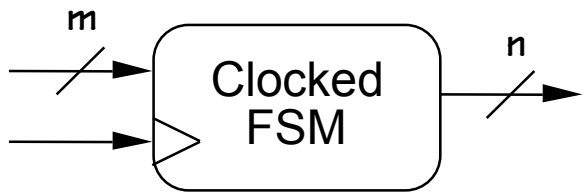
Current state	"start"	"1"	"0"	Next state	unlock
---	1	---	---	start 000	0
start 000	0	0	1	digit1 001	0
start 000	0	1	0	error 101	0
start 000	0	0	0	start 000	0
digit1 001	0	1	0	digit2 010	0
digit1 001	0	0	1	error 101	0
digit1 001	0	0	0	digit1 001	0
digit2 010	0	1	0	digit3 011	0
...					
digit3 011	0	0	1	unlock 100	0
...					
unlock 100	0	1	0	error 101	1
unlock 100	0	0	1	error 101	1
unlock 100	0	0	0	unlock 100	1
error 101	0	---	---	error 101	0

6 different states → encode using 3 bits

Now do it in Hardware!



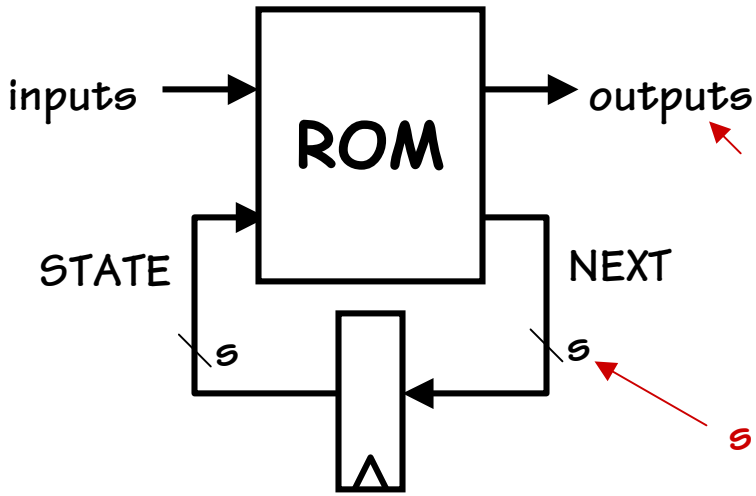
Abstraction du jour: Finite State Machines



A FINITE STATE MACHINE has

- k STATES S_1 ... S_k (one is "initial" state)
- m INPUTS I_1 ... I_m
- n OUTPUTS O_1 ... O_n
- Transition Rules S'(s,i) for each state s and input i
- Output Rules Out(s) for each state s

Discrete State, Time

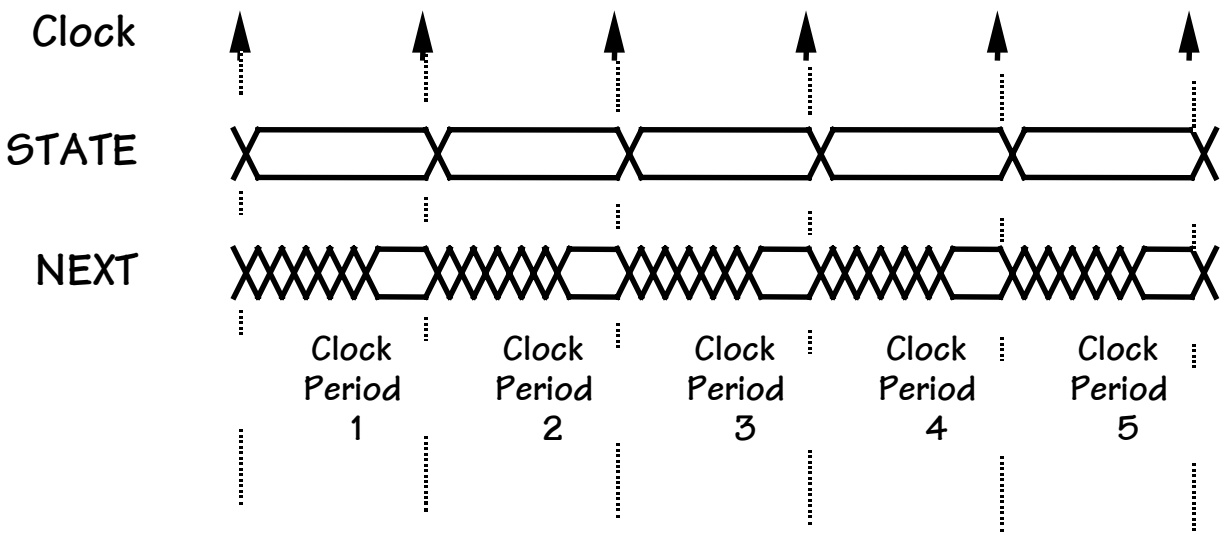


Two design choices:

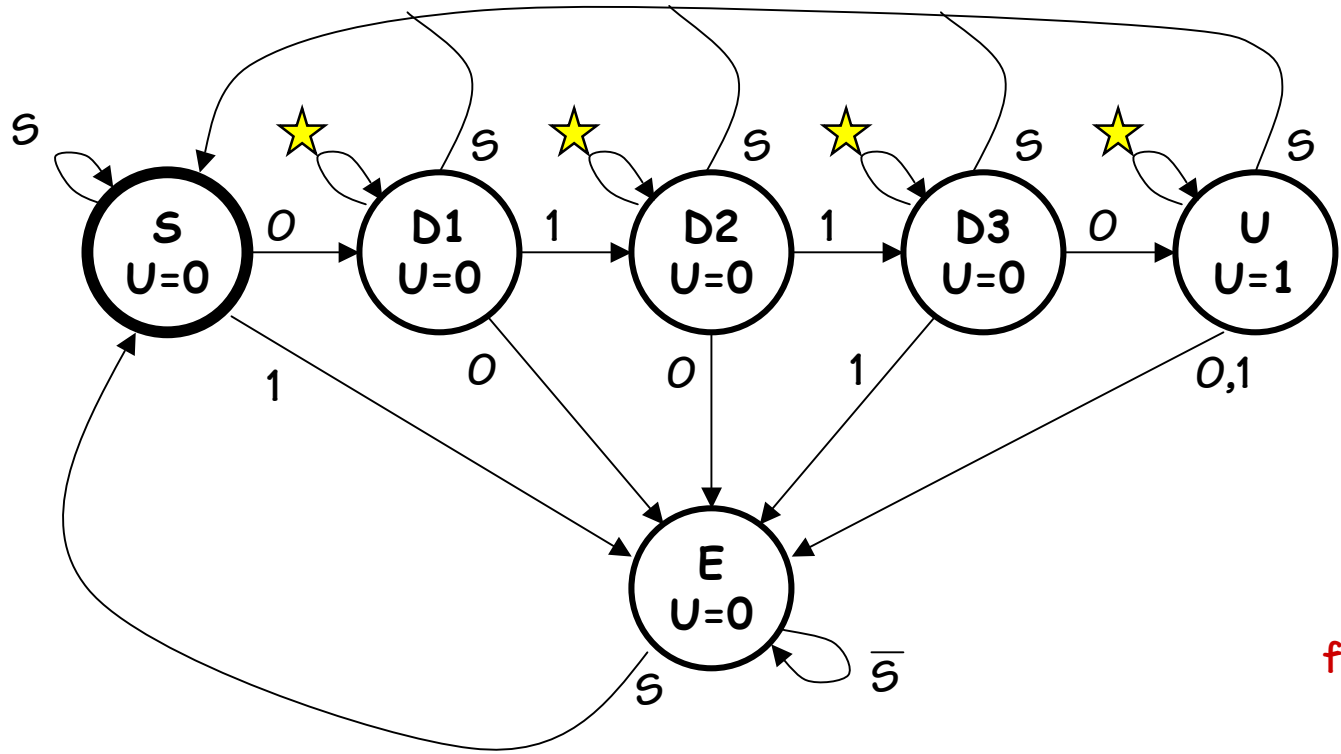
(1) outputs only depend on state (Moore)

(2) outputs depend on inputs + state (Mealy)

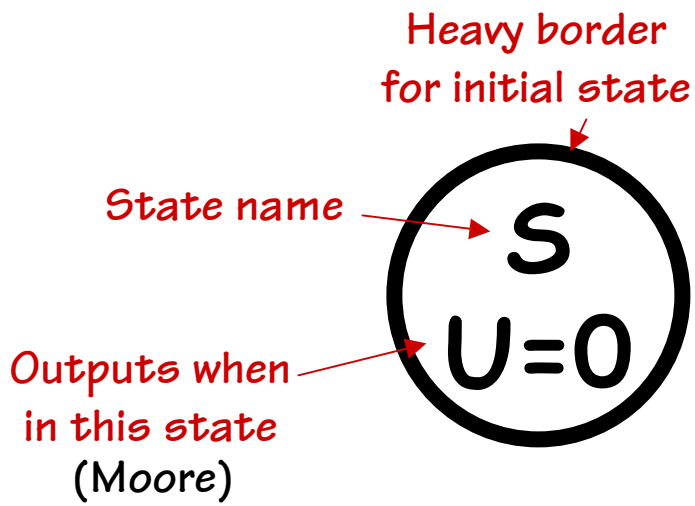
s state bits → 2^s possible states



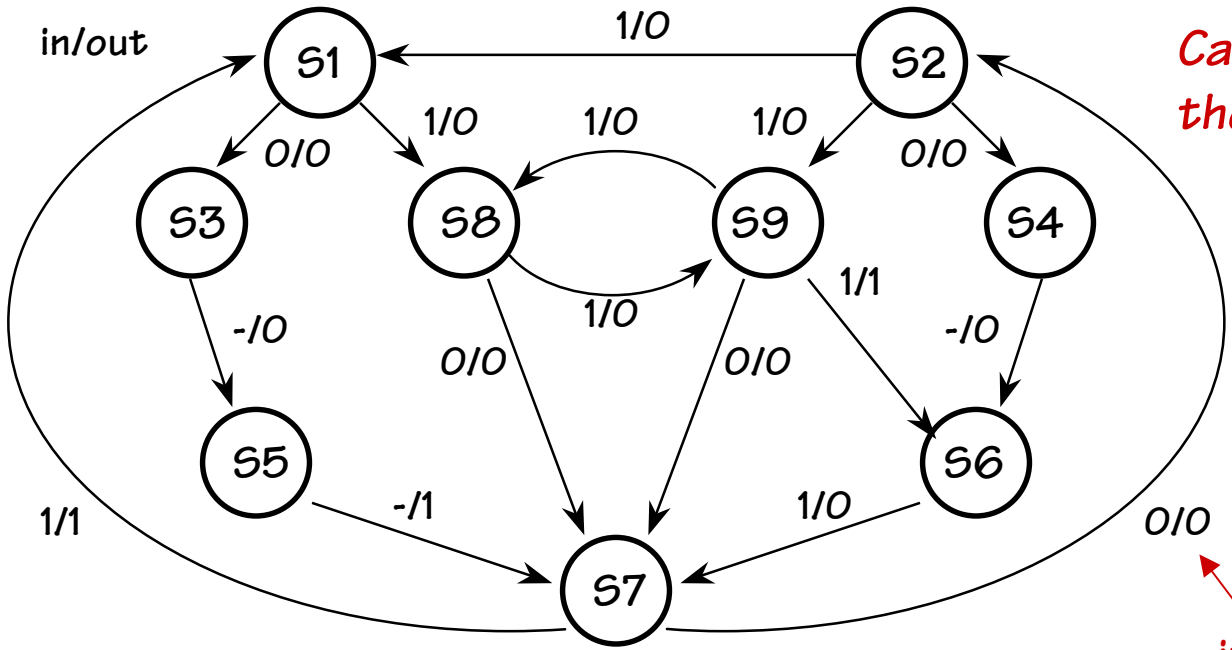
State Transition Diagrams



☆ = no buttons pressed



Valid State Diagrams



Can you spot the problems?

input/output
(Mealy)

Arcs leaving a state must be:

(1) mutually exclusive

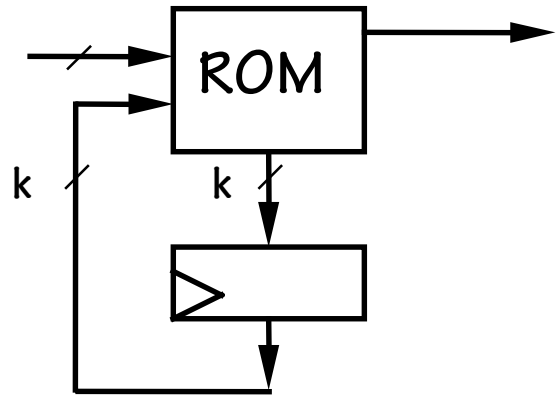
can't have two choices for a given input value

(2) collectively exhaustive

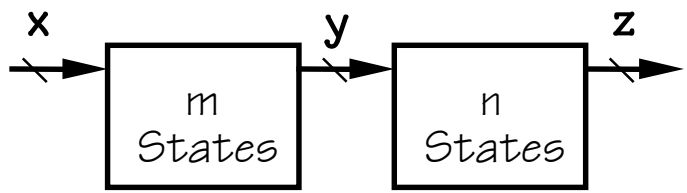
every state must specify what happens for each possible input combination. "Nothing happens" means arc back to itself.

FSM Party Games

1. What can you say about the number of states?

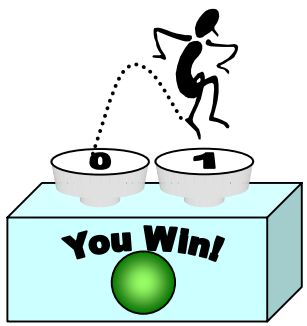


2. Same question:



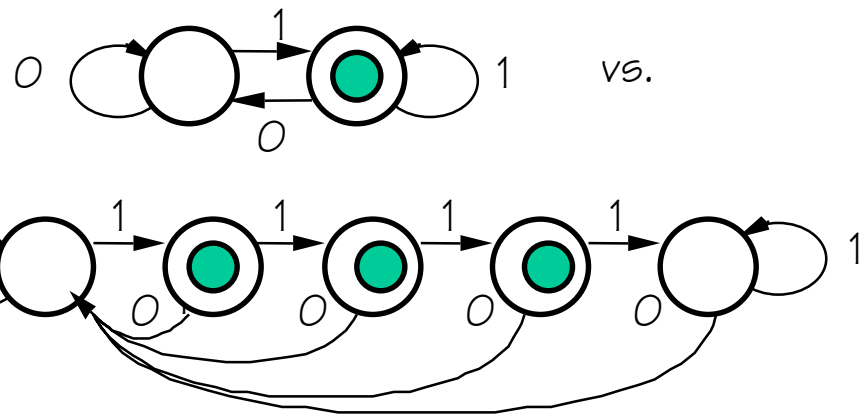
3. Here's an FSM. Can you discover its rules?

What's My Transition Diagram?



0=OFF,
1=ON?

"1111" =
Surprise!

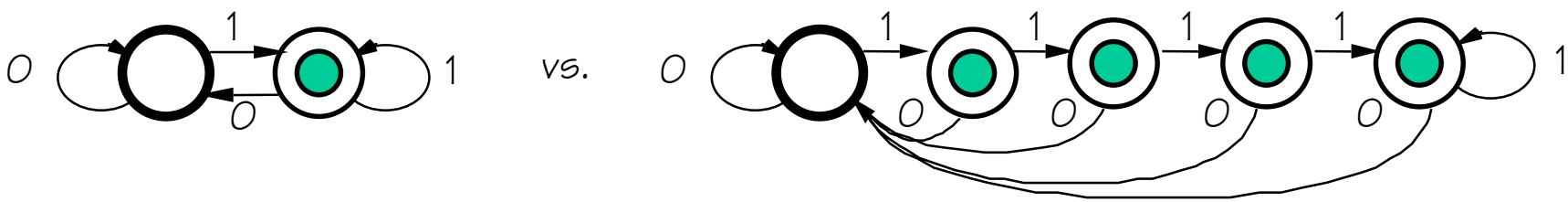


- If you know NOTHING about the FSM, you're never sure!
- If you have a BOUND on the number of states, you can discover its behavior:

  K-state FSM: Every (reachable) state can be reached in < k steps.

BUT ... states may be equivalent!

FSM Equivalence



ARE THEY DIFFERENT?

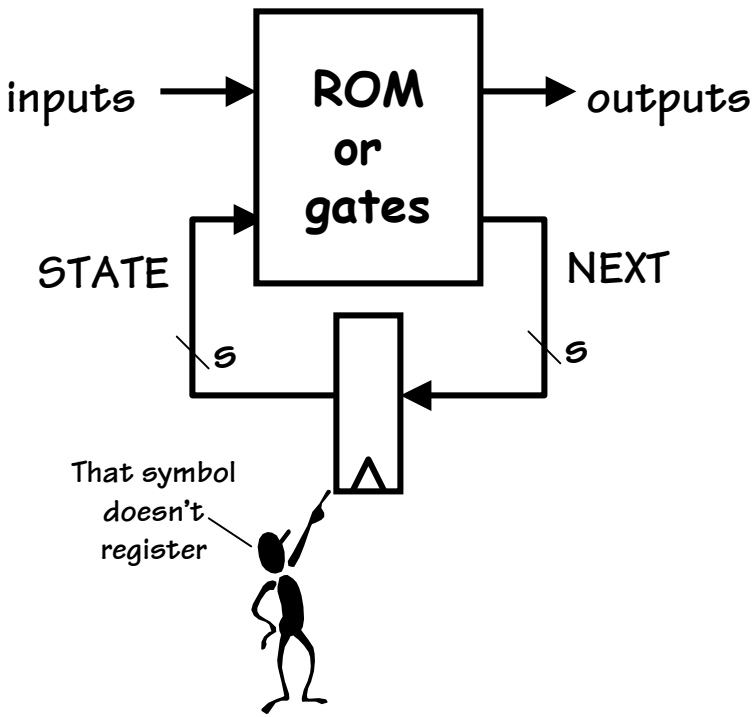
NOT in any practical sense! They are EXTERNALLY INDISTINGUISHABLE, hence interchangeable.

> FSMs *EQUIVALENT* iff every input sequence yields identical output sequences.

ENGINEERING GOAL:

- HAVE an FSM which *works...*
- WANT <u>simplest</u> (*ergo* cheapest) equivalent FSM.

Housekeeping issues...



1. Initialization? Clear the memory?

2. Unused state encodings?
   - waste ROM (use PLA or gates)
   - meaning?

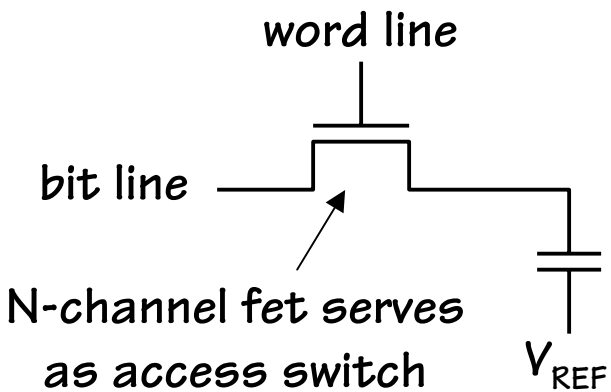
3. Synchronizing input changes with state update?

4. Choosing encoding for state?

5. How do we implement memory?

Storage: Using Capacitors

We've chosen to encode information using voltages and we know from 6.002 that we can "store" a voltage as charge on a capacitor:



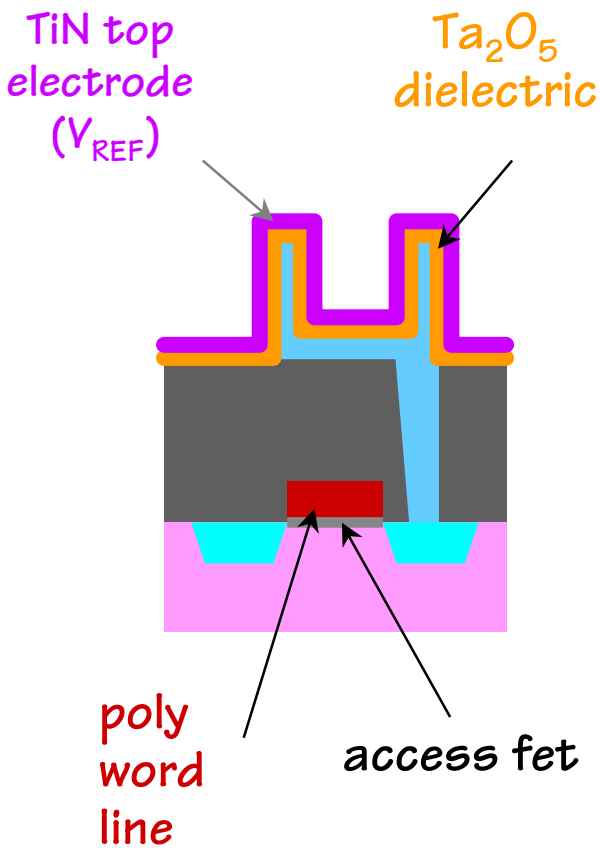
To write:
Drive bit line, turn on access fet, force storage cap to new voltage

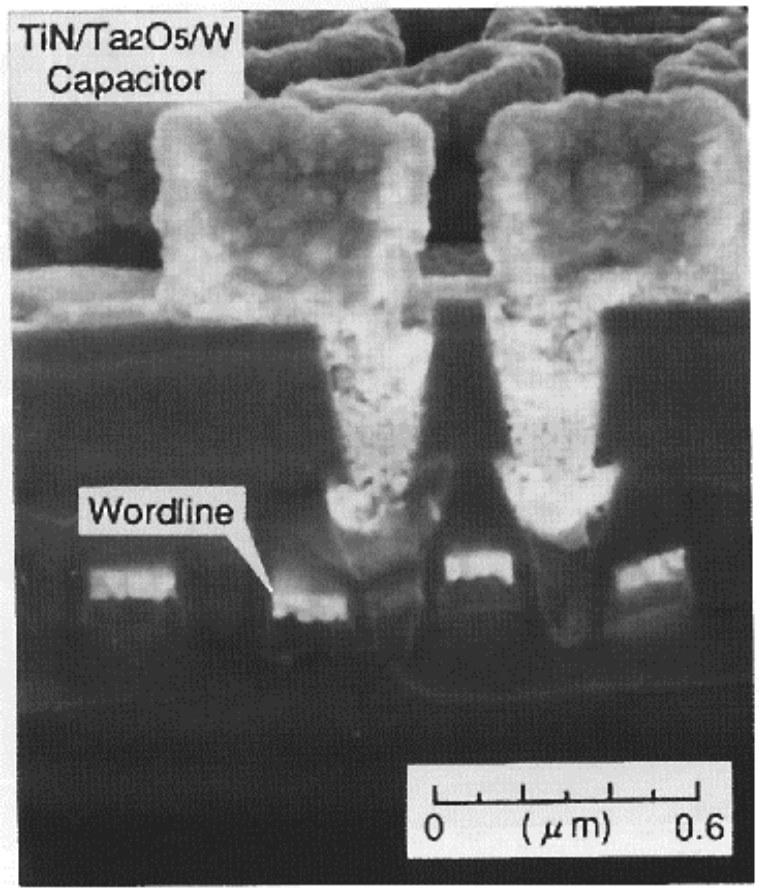
To read:
precharge bit line, turn on access fet, detect (small) change in bit line voltage

Pros:
- compact!

Cons:
- it leaks! ⇒ refresh
- complex interface
- stable? (noise, ...)

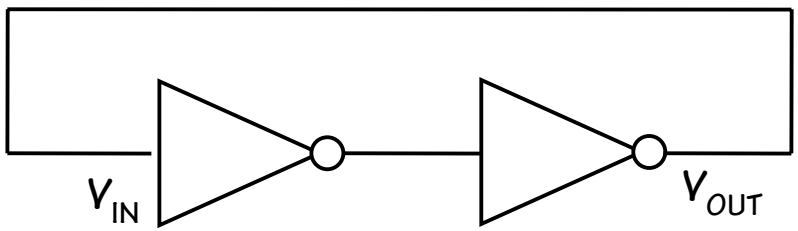
Dynamic Memory



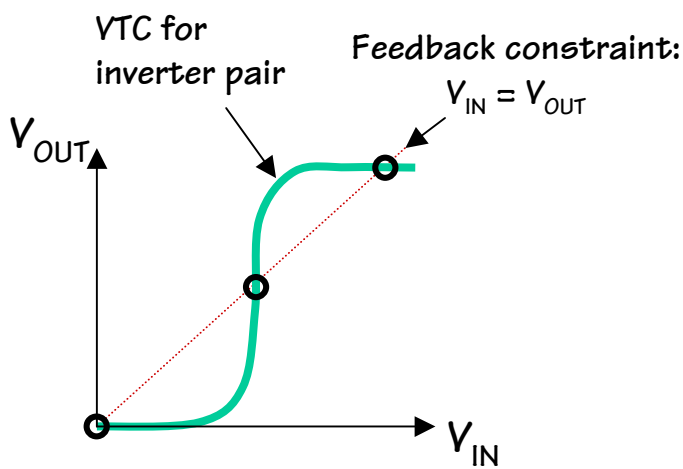


Storage: Using Feedback

BIG IDEA: use positive feedback to maintain storage indefinitely. Our logic gates are built to restore marginal signal levels, so noise shouldn't be a problem!



Result: a bistable storage element



Not affected by noise

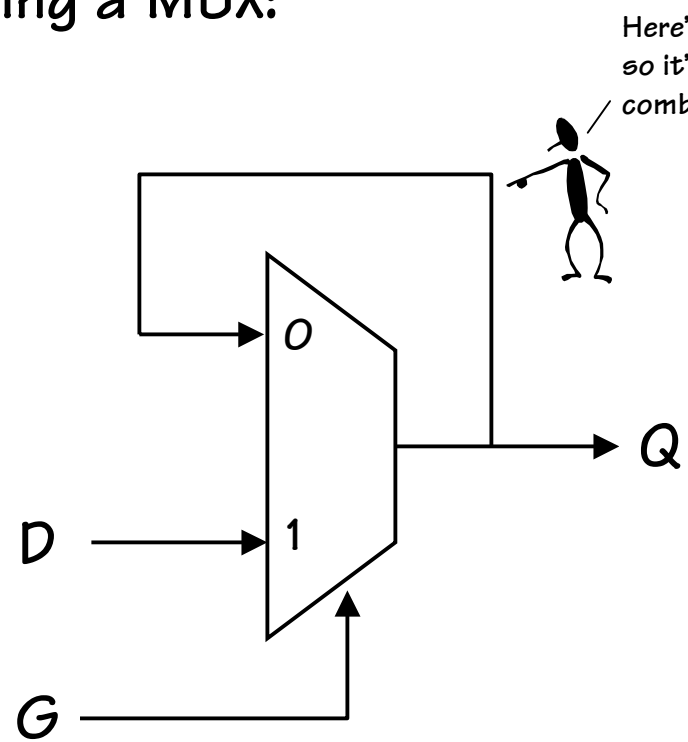
Three solutions:

- two end-points are stable
- middle point is unstable

We'll get back to this!

Settable storage element

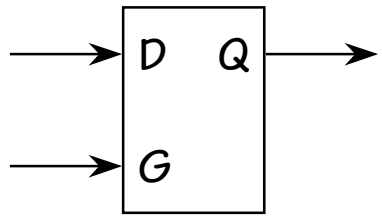
It's easy to build a settable storage element (called a latch) using a MUX:



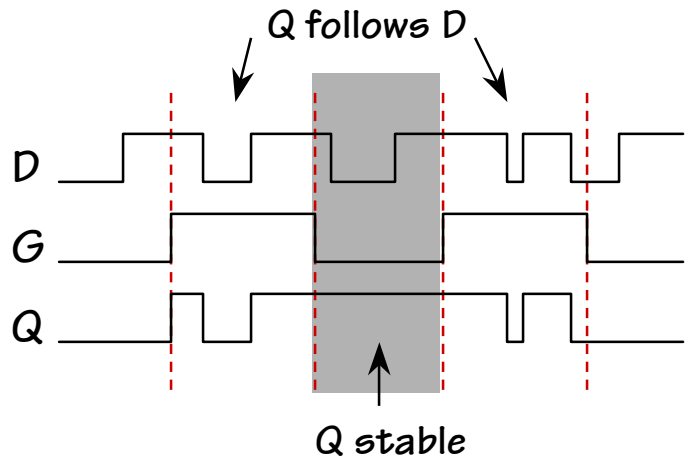
"state" signal appears as both input and output

G	D	Q_{IN}	Q_{OUT}	
0	--	0	0	Q stable
0	--	1	1	
1	0	--	0	Q follows D
1	1	--	1	

Static D Latch

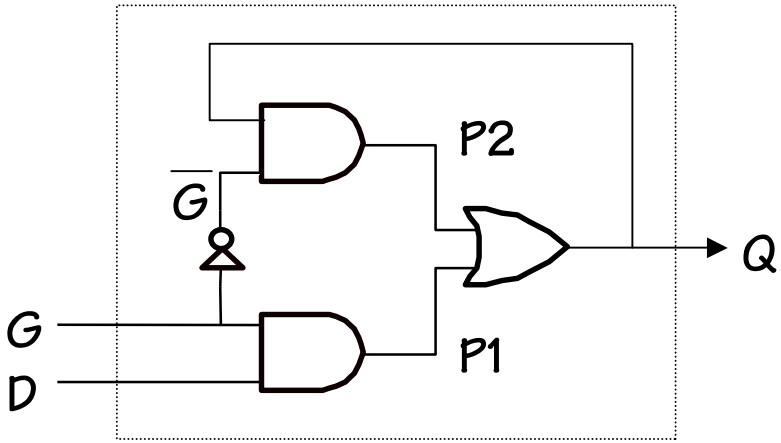


Positive latch



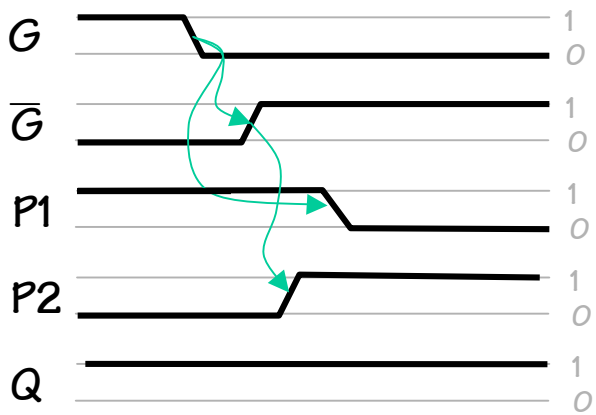
"static" means latch will hold data (i.e., value of Q) while G is inactive, however long that may be. Q: Does this take static power?

Potential "hazard"

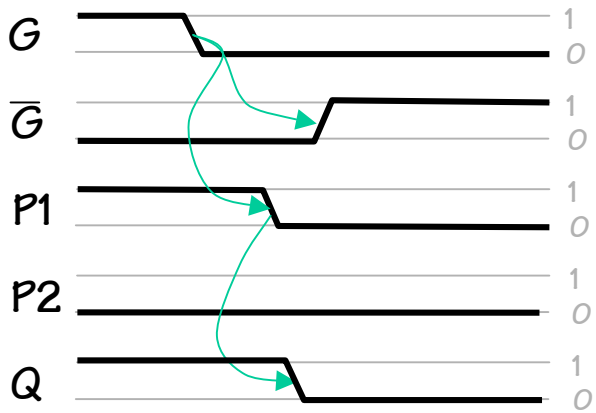


Let D = 1 and consider G: 1→0

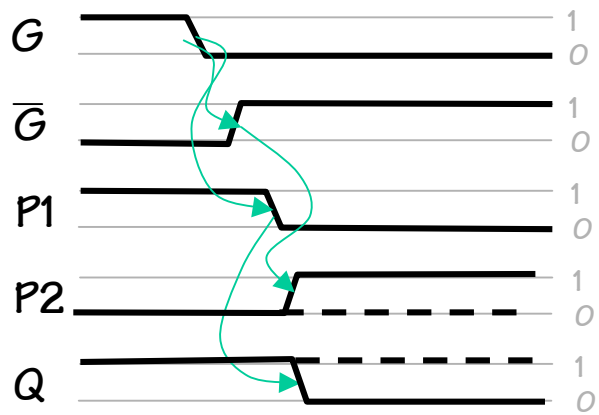
- P1 = 1
- P2 = 0
- Q = 1



Scenario 1:
- fast inverter
- slow AND, OR



Scenario 2:
- slow inverter
- fast AND, OR

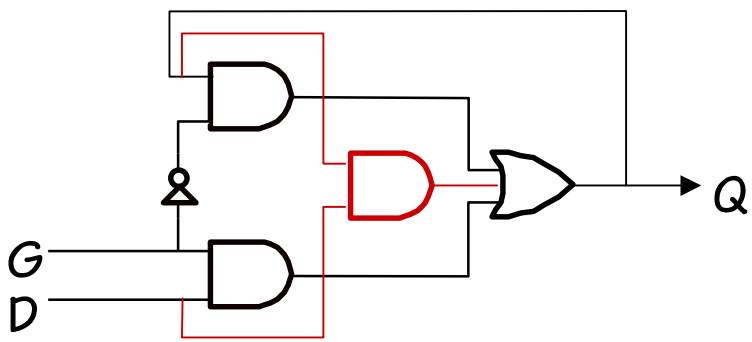


Scenario 3:
- fast inverter
- fast AND, OR

Hazards in perspective

Two possible fixes:

1. Adjust gate timings
2. Add AND gate to cover P1:P2 transition



Q: When should we worry about hazards?

A. Whenever we build combinational circuits.

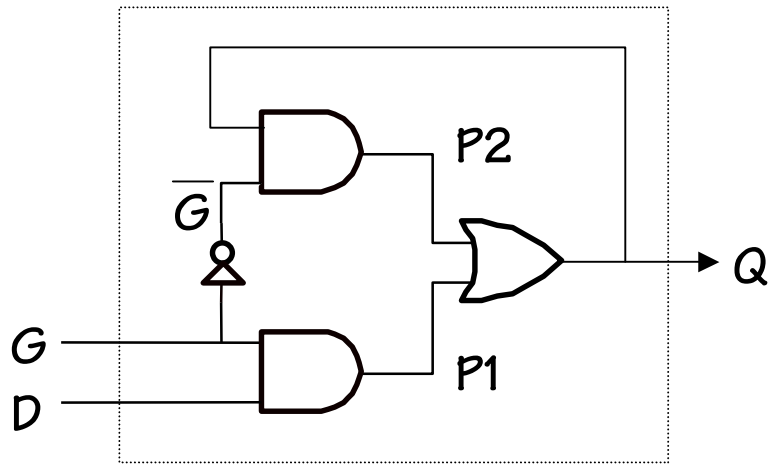
B. When driving in Cambridge

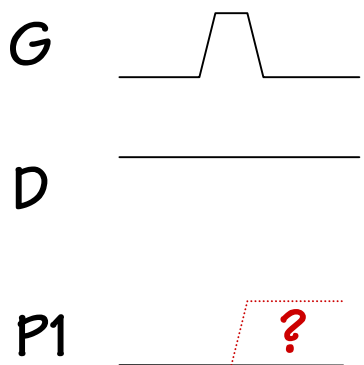
C. On 6.004 quizzes

D. Only in special circumstances, eg, when using feedback to build static storage elements.

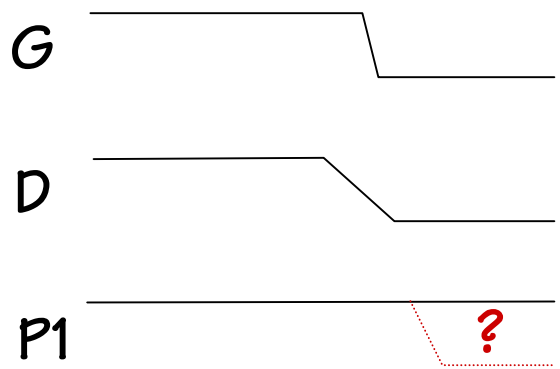
A. "D": Common digital engineering approaches do not depend on combinational outputs during propagation, hence tolerate combinational hazards.

More serious problems





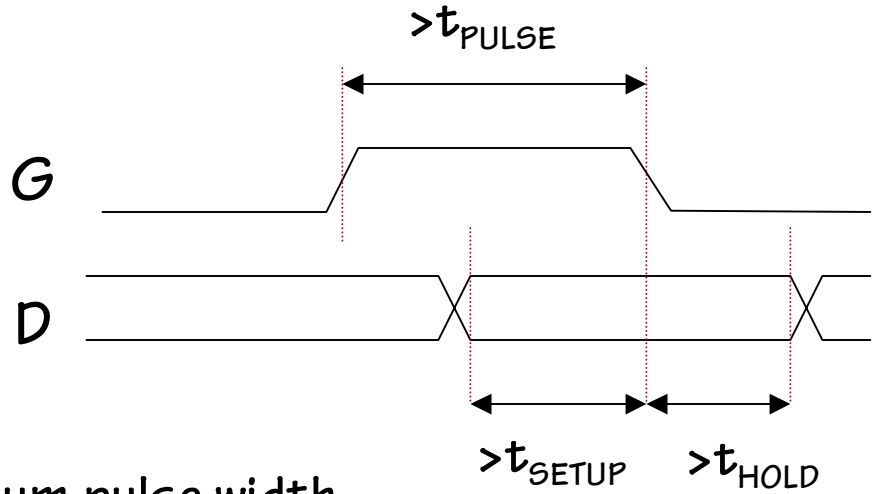
Short pulse on G: how long is enough to set the latch?



D changes about the same time as latch closes. What value is saved?

A Dynamic Discipline

Design of sequential circuits MUST guarantee that inputs to sequential devices are valid and stable during periods when they may influence state changes.



t_{PULSE}: minimum pulse width

guarantee G is active for long enough for latch to capture data

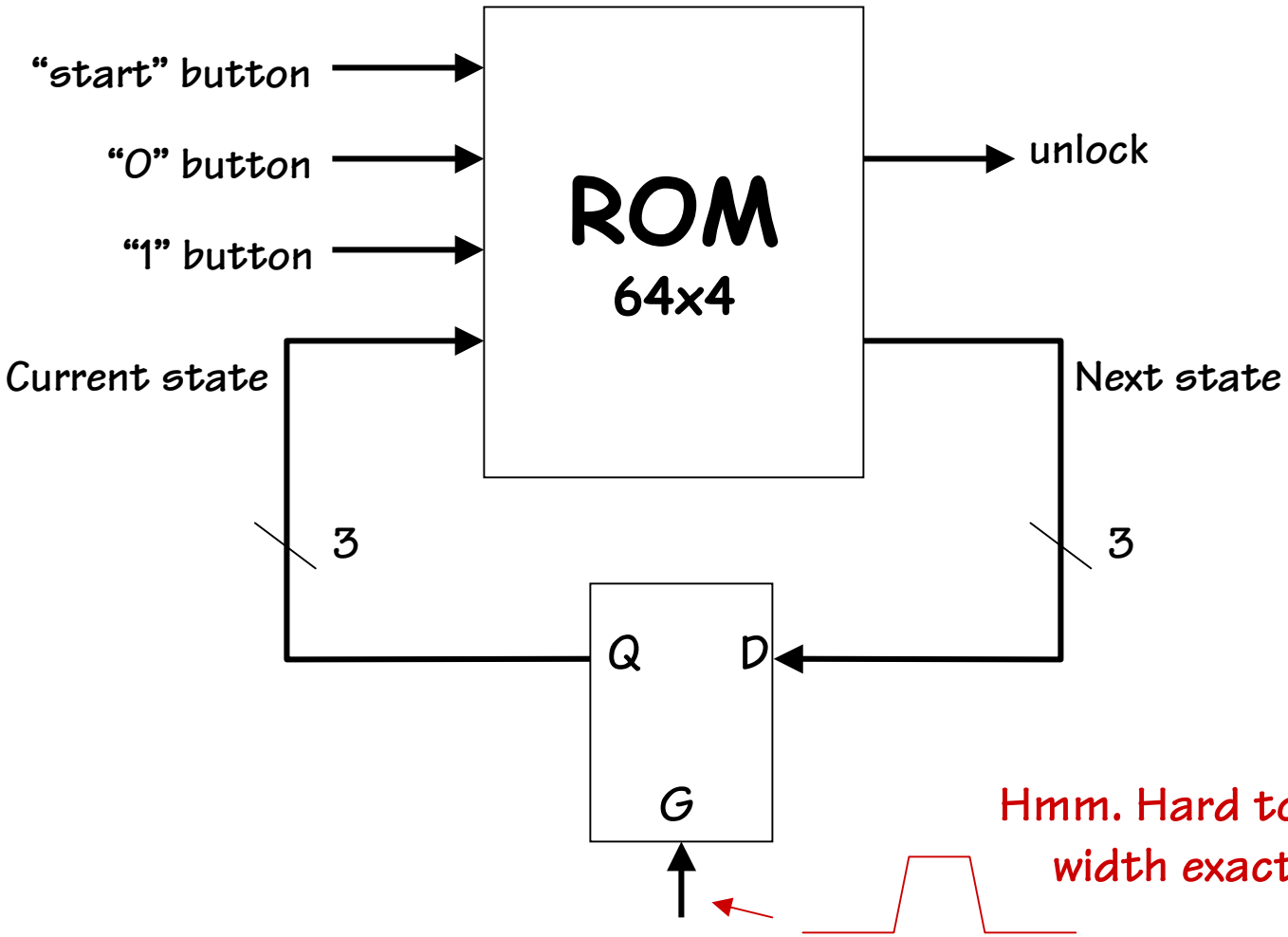
t_{SETUP}: setup time

guarantee that D value has propagated through feedback path before latch closes

t_{HOLD}: hold time

guarantee latch is closed and Q is stable before allowing D to change

Does this work yet?



Summary

- sequential logic = combinational logic + memory
- new abstraction: finite state machines
  - transition diagrams showing states, outputs, and
  - transition arcs: mutually exclusive, collectively exhaustive
- memory elements
  - dynamic memory: compact, only reliable short-term
  - static memory: controlled use of positive feedback
- level-sensitive D-latches
- dynamic discipline (setup and hold times)

To do list:

- how to generate correct size pulse for G input to latch?
- what about 3rd solution point in bistable storage element?
- input synchronization to avoid violating dynamic discipline?