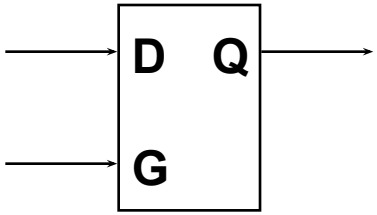
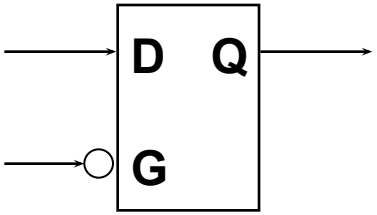
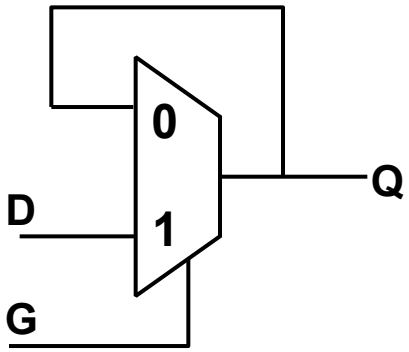
Latches: Digital storage Elements

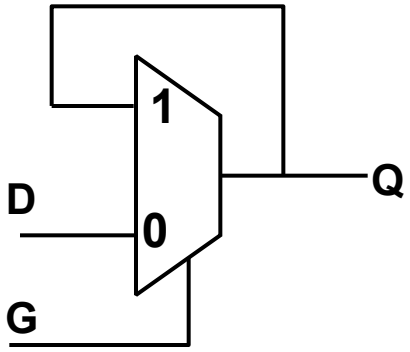


Positive latch



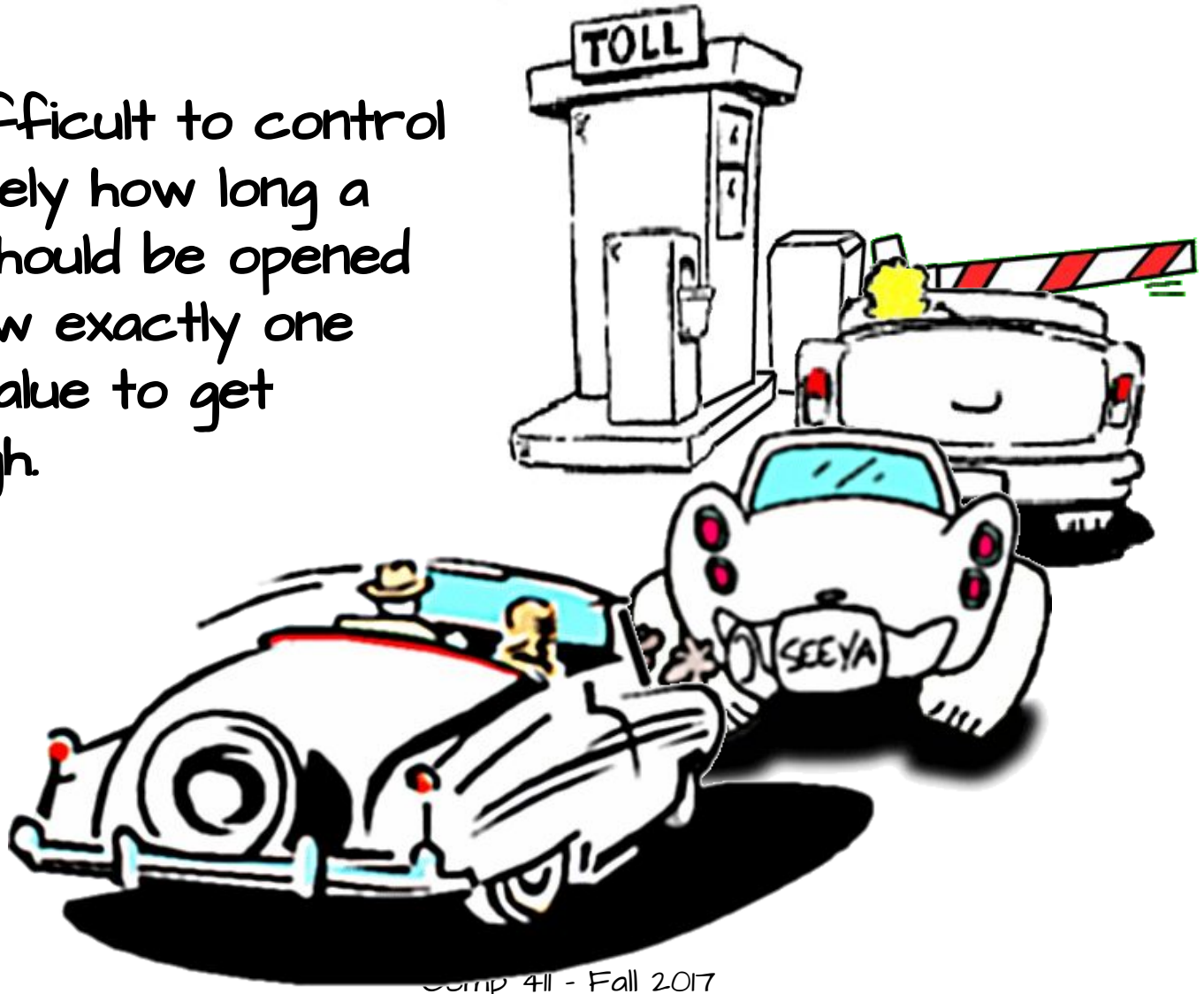
Negative latch





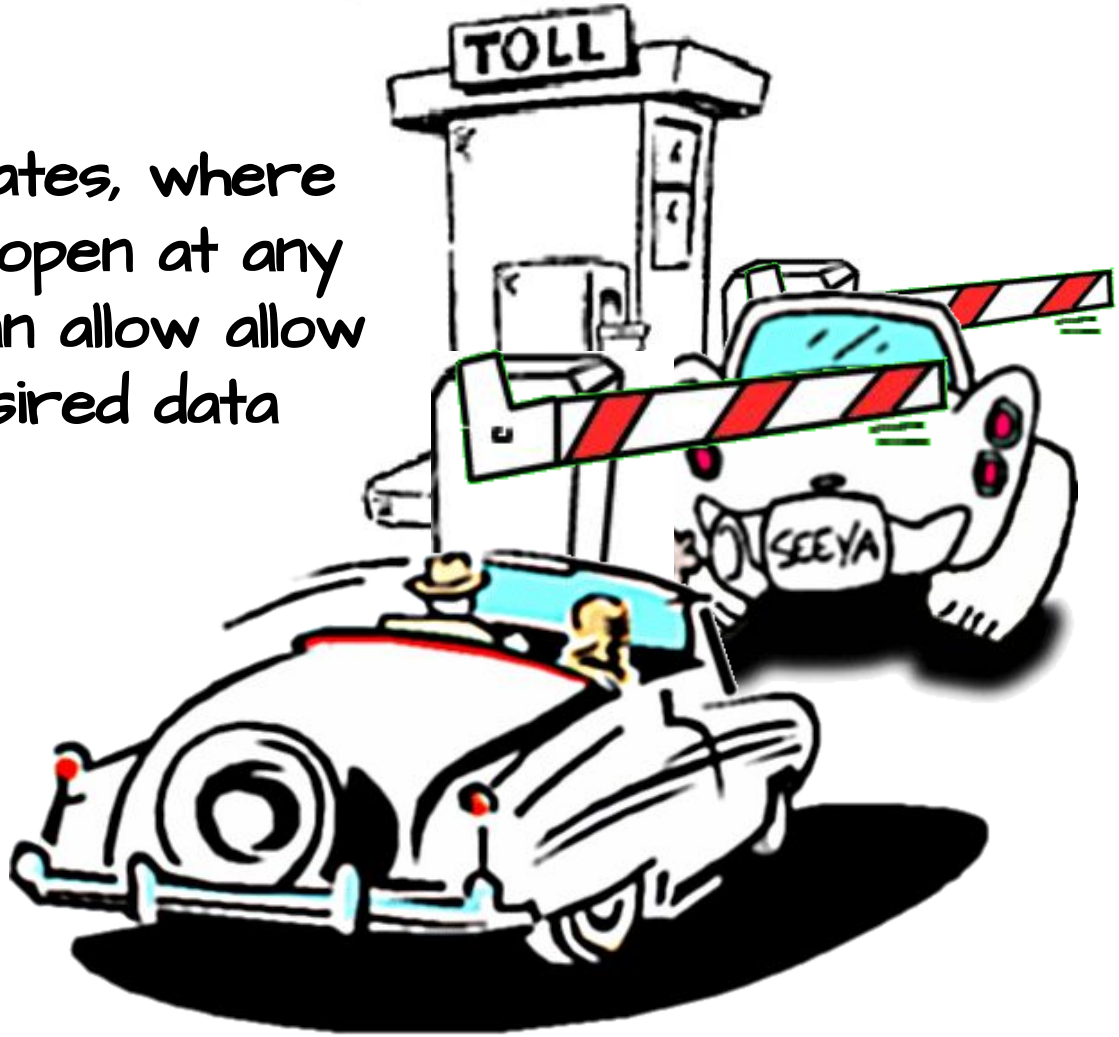
Latches are a digital "memory", they "remember" one-bit while their "gate", G, is closed. When open, their output follows the input D. The last value of D is remembered when the gate is closed.

One gate is not enough



It is difficult to control precisely how long a gate should be opened to allow exactly one data value to get through.

Two gates provide precise control

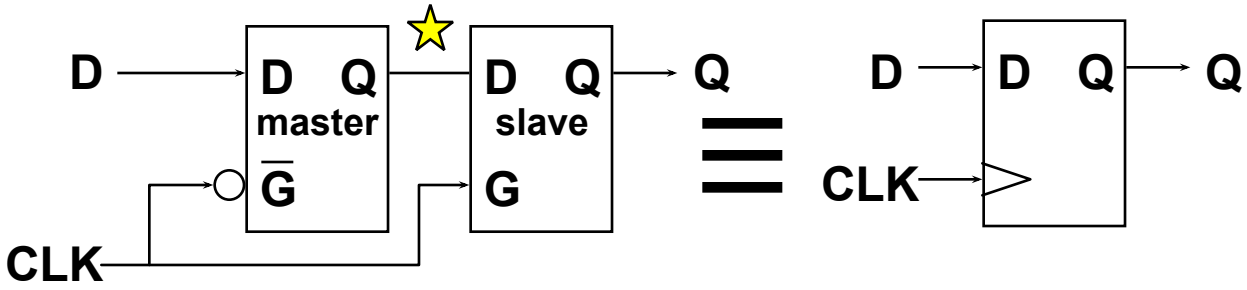


The Solution:

With two gates, where only one is open at any time, we can allow allow only the desired data through

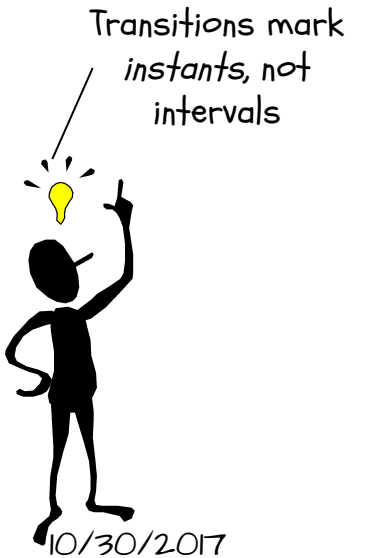
Edge-Triggered Flip Flop

Logical "Escapement"

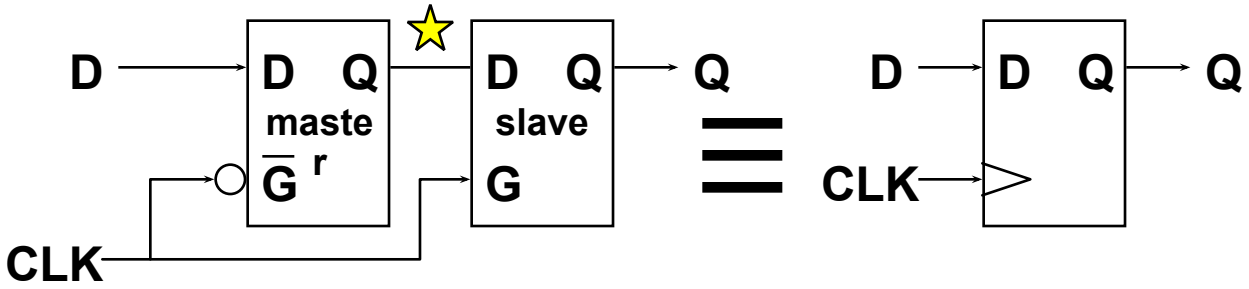


Observations:

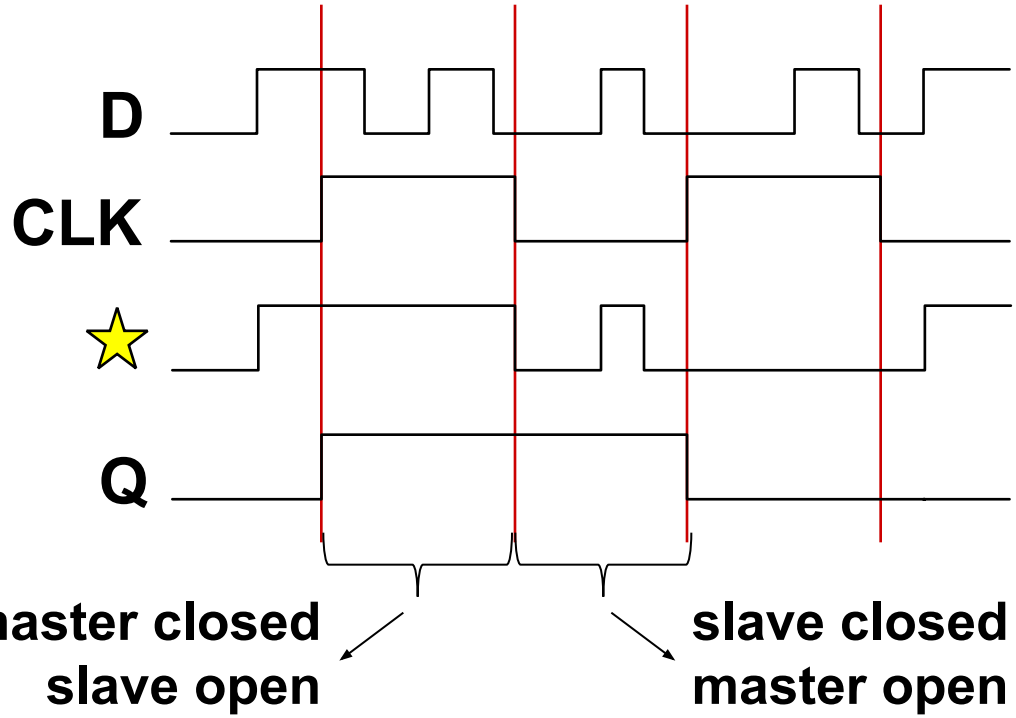
- only one latch's gate is open at any time
- master closed when slave is open (CLK is high)
- slave closed when master is open (CLK is low)
- no combinational path through flip flop
- Q only changes shortly after 0→1 transition of CLK, so a "flip flop" appears to be "triggered" by rising edge of CLK



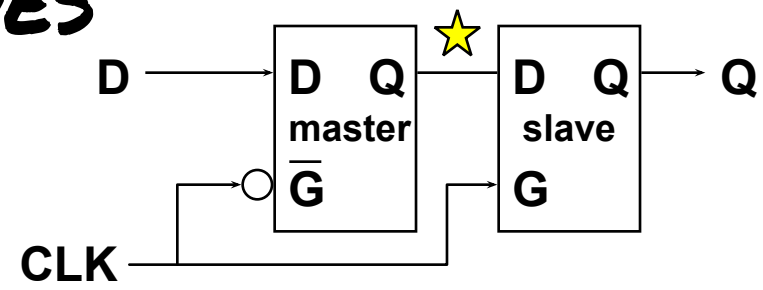
Flip-flop Timing



The output of a flip-flop only changes after the rising edge of its clock

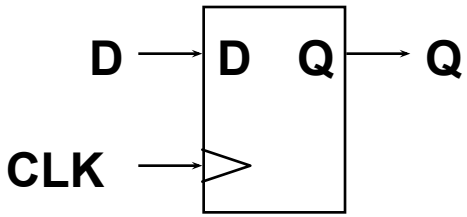


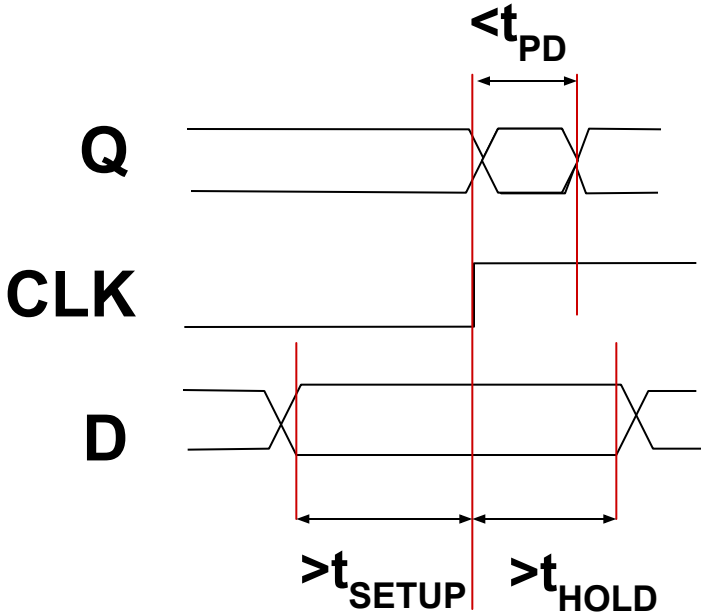
Two Issues



- Must allow time for the input's value to propagate to the Master's output while CLK is LOW.
  - This is called "SET-UP" time (How long a D input must valid before the clock rises)
- Must keep the input stable, just after CLK transitions to HIGH. This is insurance in case the SLAVE's gate opens just before the MASTER's gate closes.
  - This is called "HOLD-TIME" (How long a D input must "remain" valid after the clock rises)
  - Can be zero (or even negative!)
- Assuring "set-up" and "hold" times is what limits a computer's performance

Flip-Flop Timing Specs





t_{PD}: maximum propagation delay, CLK →Q

t_{SETUP}: setup time

guarantee that D has propagated through feedback path before master closes

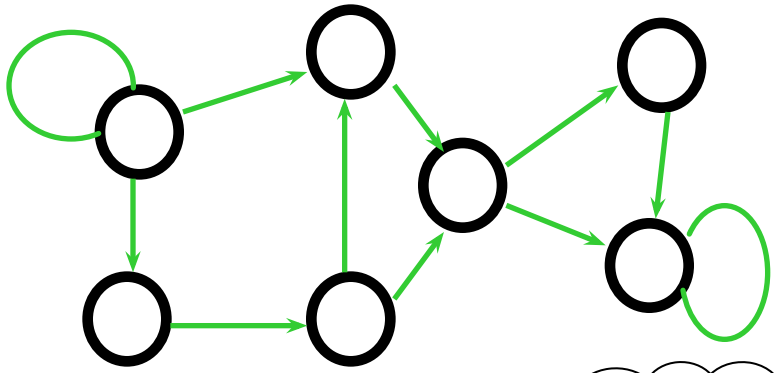
t_{HOLD}: hold time

guarantee master is closed and data is stable before allowing D to change

Summary

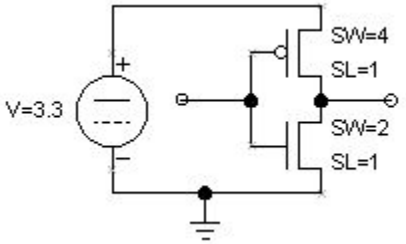
- Regular Arrays can be used to implement arbitrary logic functions
- ROMs decode every input combination (fixed-AND array) and compute the output for it (customized-OR array)
- Memories
  - ROMs are HARDWIRED memories
  - RAMs include storage elements at each WORD-line and BIT-line intersection
    - dynamic memory: compact, only reliable short-term
    - static memory: controlled use of positive feedback
- Level-sensitive D-latches for static storage
- Dynamic discipline (setup and hold times)

Synchronous Logic

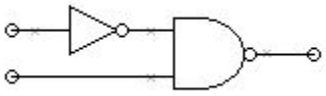


1) Sequential Logic
2) Synchronous Design
3) Synchronous Timing Analysis
4) Single Clock Design
5) Finite State Machines
6) Mealy and Moore
7) State Transition Diagrams

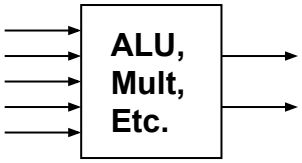
Road Traveled So Far...



FETs & voltages



Logic gates



Combinational logic circuits

Sequential Logic

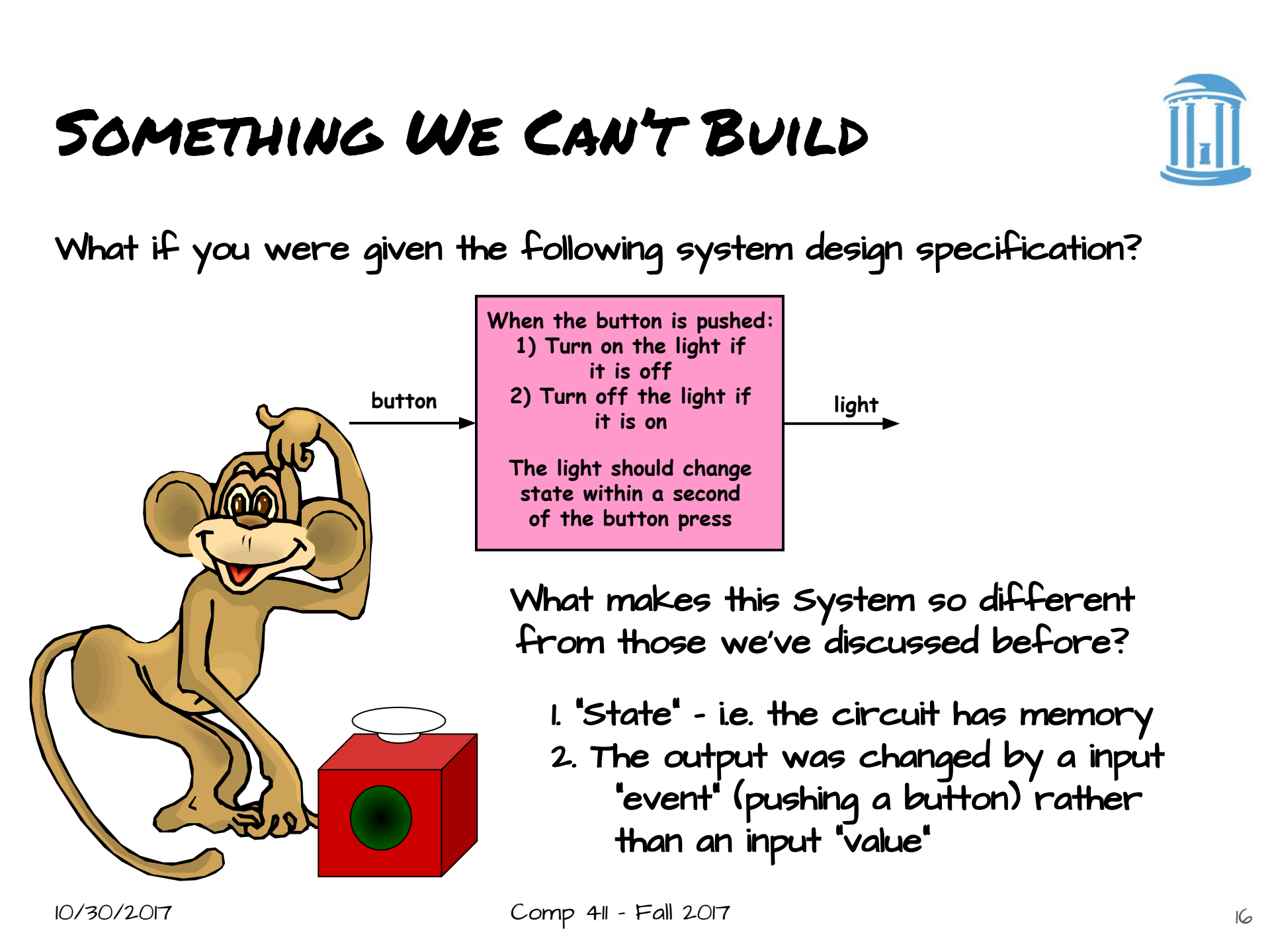
Combinational contract:
- Voltage-based "bits"
- 1-bit per wire
- Generate quality outputs, tolerate inferior inputs
- Combinational contract
- Complete in/out/timing spec

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

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

Our motto: Sweat the details once, and then put a box around it!

Something We Can't Build

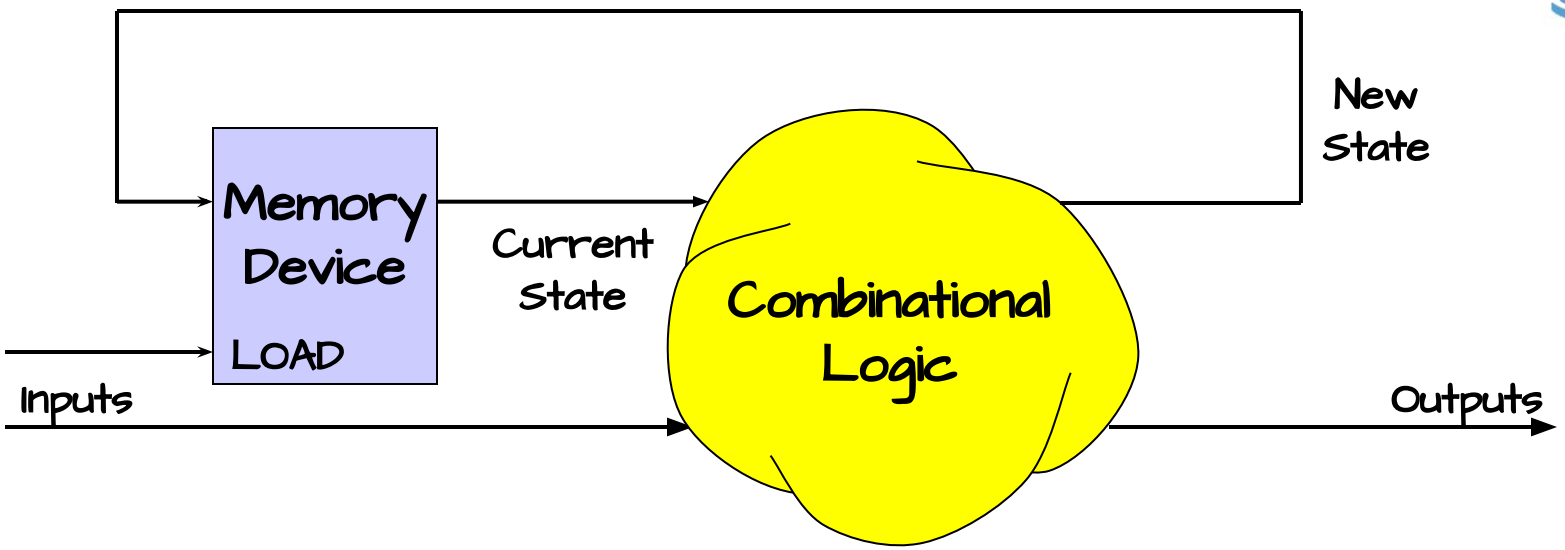


What if you were given the following system design specification?

What makes this System so different from those we've discussed before?

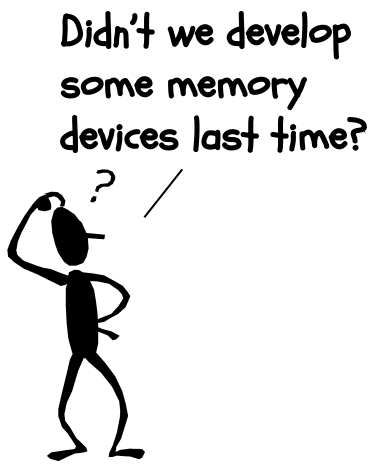
1. "State" - i.e. the circuit has memory
2. The output was changed by a input "event" (pushing a button) rather than an input "value"

"Sequential" = Stateful

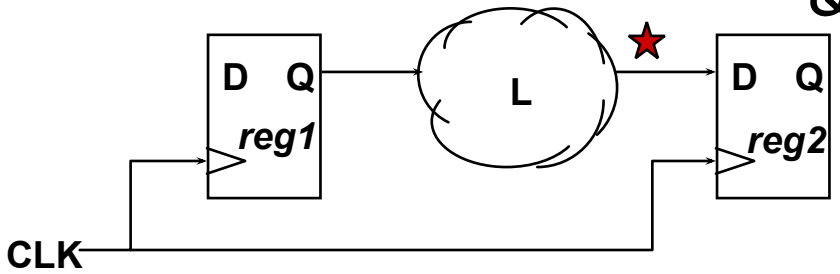


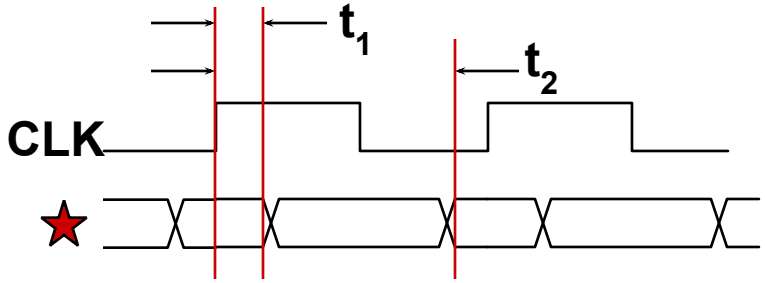
Plan: Build a Sequential Circuit with stored digital STATE -

- MEMORY stores CURRENT state
- Combinational Logic computes
  - the NEXT state (Based on inputs & current state)
  - the OUTPUTs (Based on inputs and/or current state)
  - State changes on LOAD control input



"Synchronous" Single-Clock Logic





Questions for register-based designs:

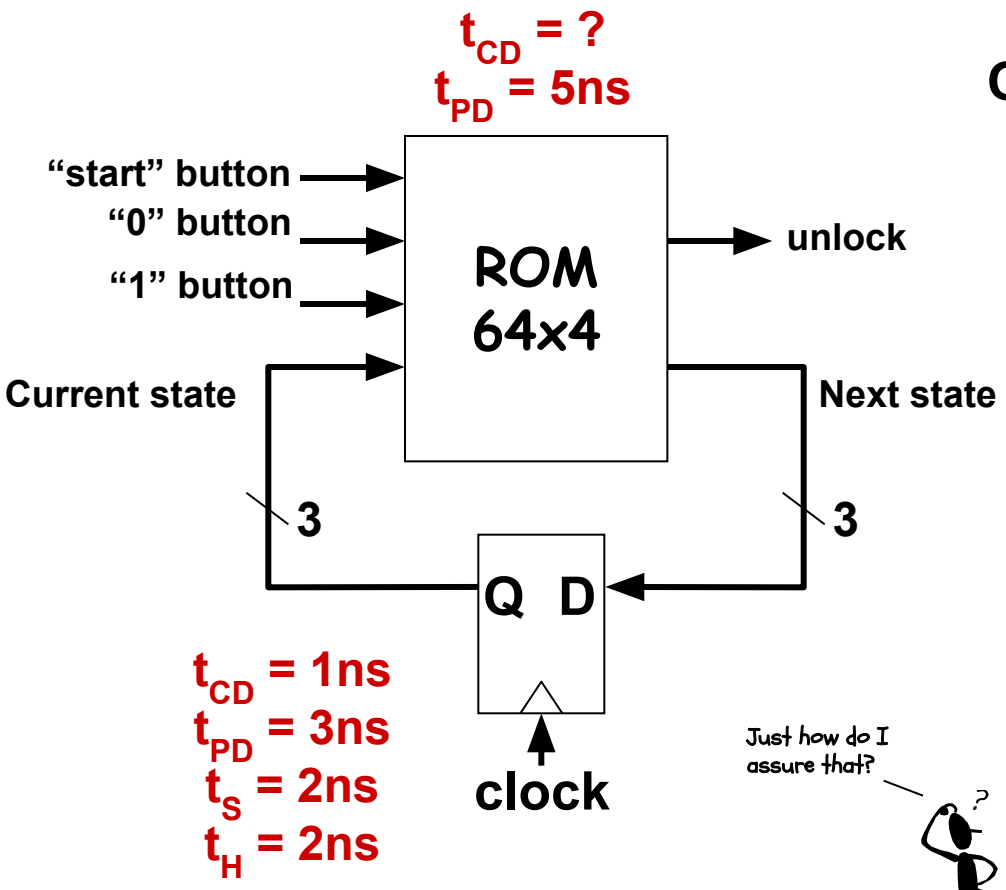
- How much time for useful work (i.e. for combinational logic delay)?
- Does it help to guarantee a minimum t_{CD}? How 'bout designing registers so that $t_{CD,reg} \geq t_{HOLD,reg}$?
- What happens if CLK signal doesn't arrive at the two registers at exactly the same time (a phenomenon known as "clock skew")?

$t_1 = t_{CD,reg1} + t_{CD,L} \geq t_{HOLD,reg2}$

$t_2 = t_{PD,reg1} + t_{PD,L} \leq t_{CLK} - t_{SETUP,reg2}$

Minimum Clock Period : $t_{CLK} \geq t_{PD,reg1} + t_{PD,L} + t_{SETUP,reg2}$

Example: Synchronous Timing



Questions:

1. t_{CD} for the ROM?

   $t_{CD,REG} + t_{CD,ROM} > t_{H,REG}$
   $1\text{ ns} + t_{CD,ROM} > 2\text{ nS}$
   $t_{CD,ROM} > 1\text{ nS}$

2. Min. clock period?

   $t_{CLK} > t_{PD,REG} + t_{PD,ROM} + t_{S,REG}$
   $t_{CLK} > 3\text{ ns} + 5\text{ ns} + 2\text{ nS}$
   $t_{CLK} > 10\text{ nS}$

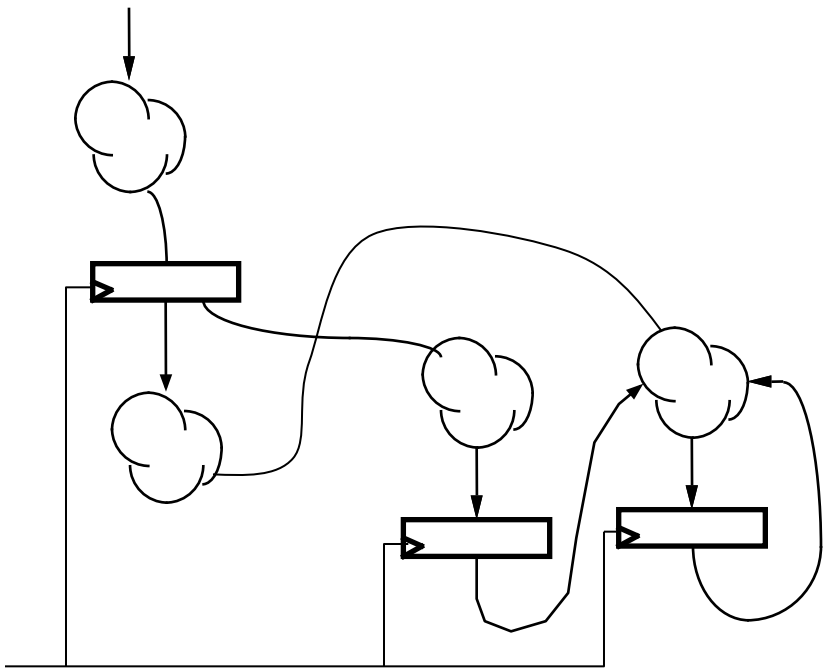
3. Constraints on inputs?

"start" , "0", and "1" must be valid $t_{PD,ROM} + t_{S,REG} = 5 + 2 = 7$ ns before the clock and held $t_{H,REG} - t_{CD,ROM} = 2 - 1 = 1$ ns after it.

Synchronous Single-Clock Design

Sequential ≠ Synchronous

However, Synchronous = A recipe for robust sequential circuits:



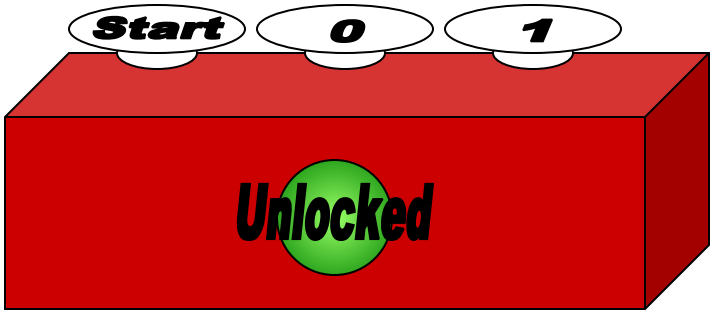
- No combinational cycles
  (other than those already inside the registers)
- Only cares about values of combinational circuits just before rising edge of clock
- Clock period greater than every combinational delay
- Changes state after all logic transitions have stopped!

Designing Sequential Logic

Sequential logic is used when the solution to some design problem involves 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"

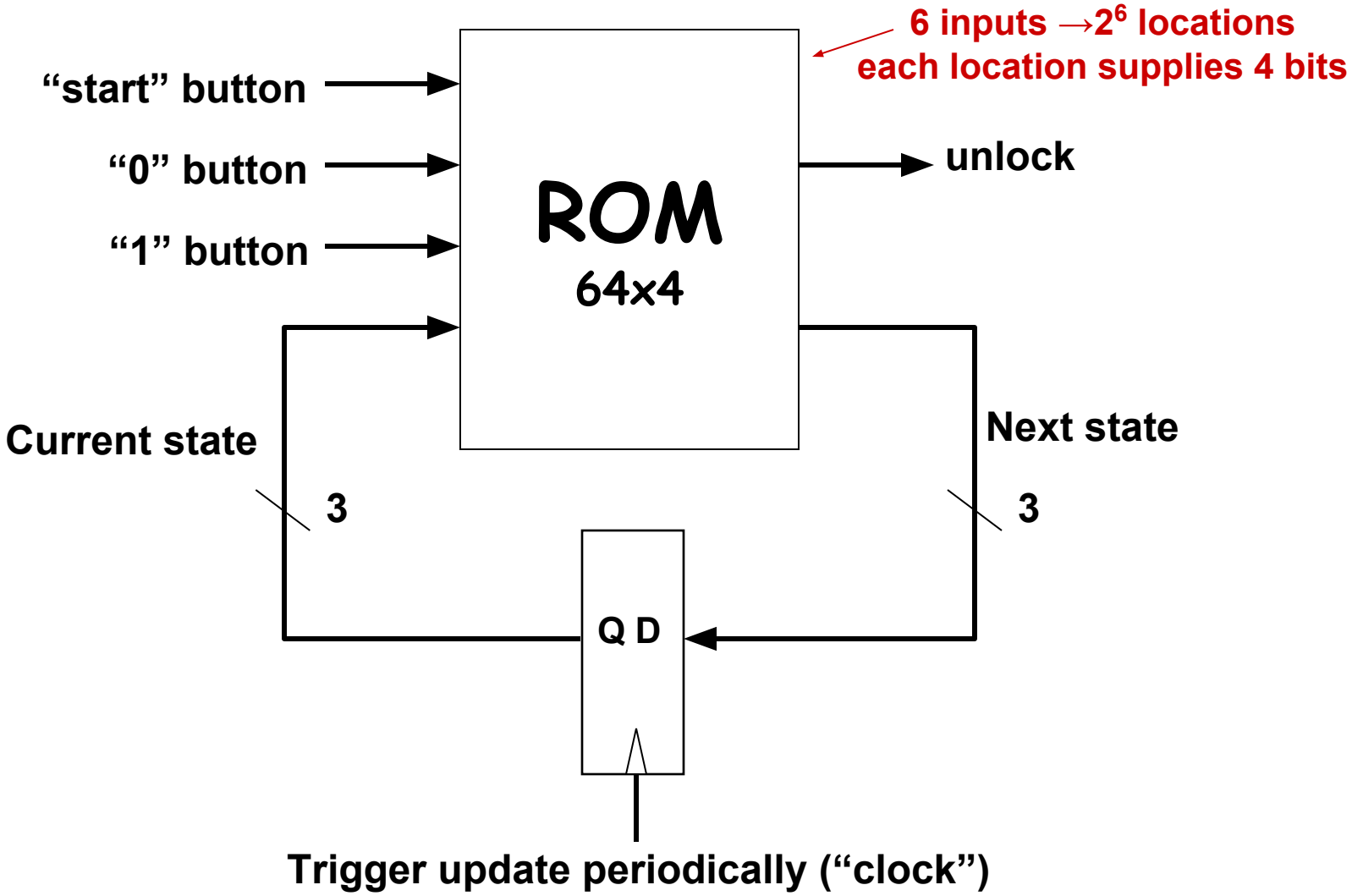
This flavor of "truth-table" is called a "state-transition table"

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

This is starting to look like a PROGRAM

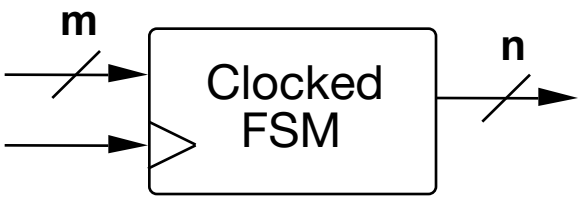
6 different states → encode using 3 bits

Now, We Do It With Hardware!



Abstraction du jour:

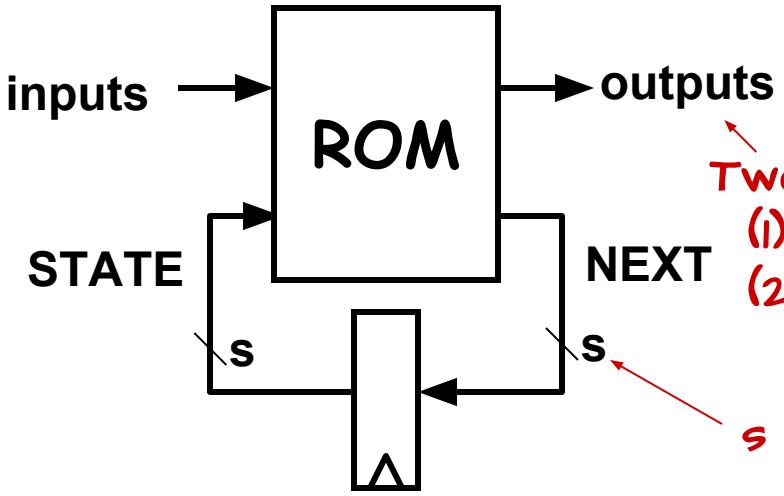
A Finite State Machine



A Finite State Machine has:

- k States S_1, S_2, … S_k (one is the "initial" state)
- m inputs I_1, I_2, … I_m
- n outputs O_1, O_2, … O_n
- Transition Rules, $S'(S_i, I_1, I_2, \ldots I_m)$ for each state and input combination
- Output Rules, $O(S_i)$ for each state

Discrete State, Discrete Time



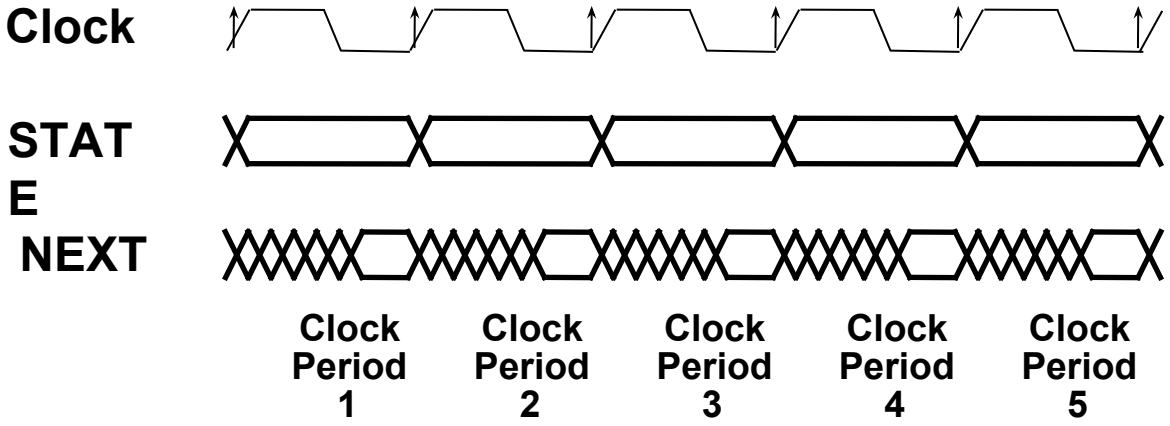
While a ROM is shown here in the feedback path any form of combinational logic can be used to construct a state machine.

Two design choices:

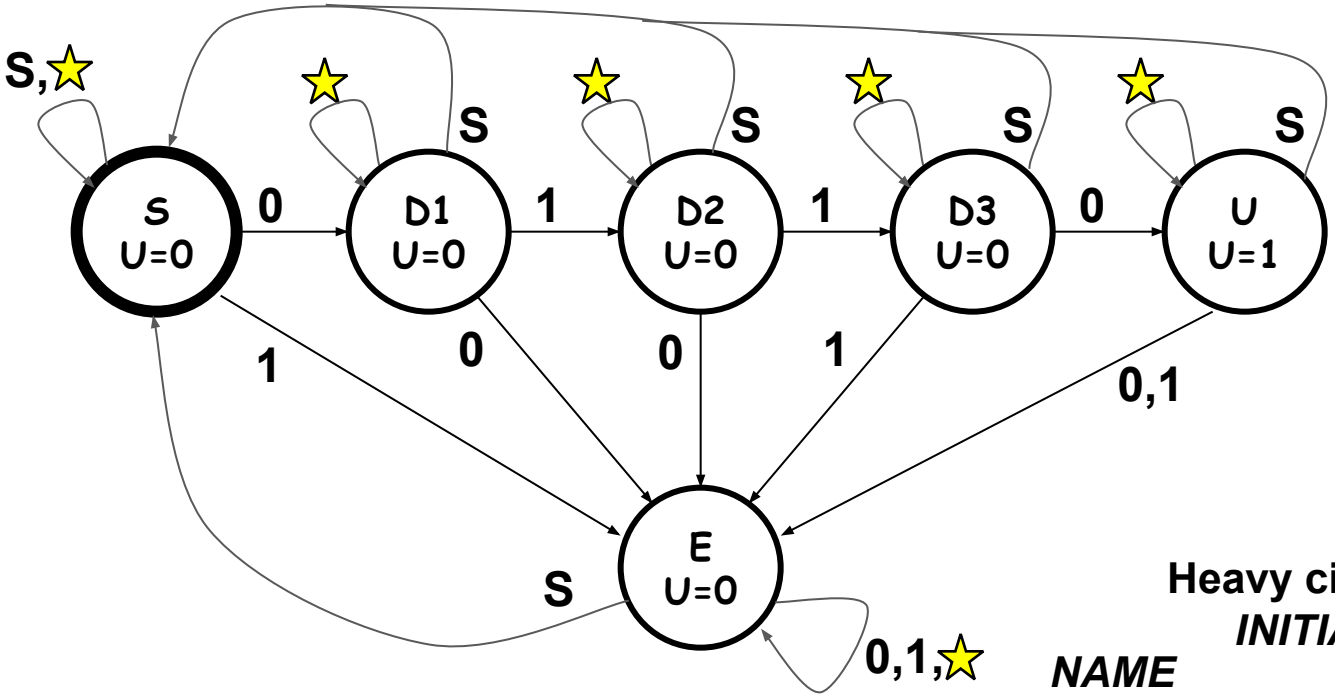
(1) outputs *only* depend on state (Moore)

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

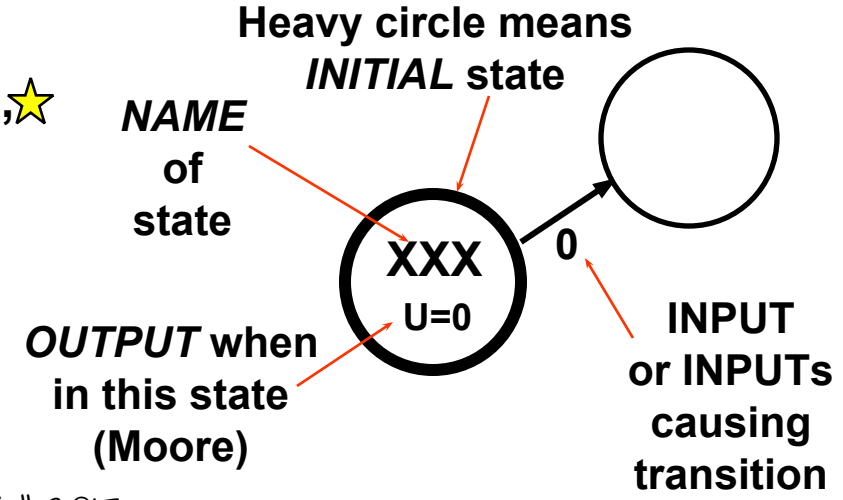
s state bits → 2^s possible states



State Transition Diagrams

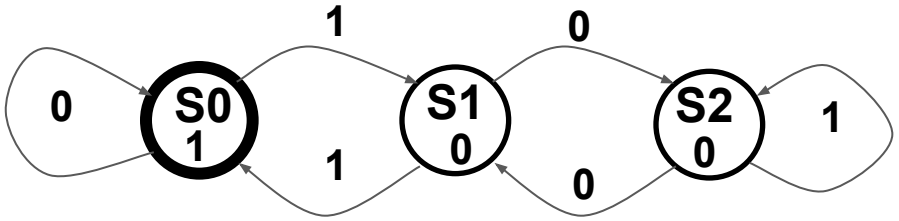


A state transition diagram is an abstract "graph" representation of a "state transition table", where each state is represented as a node and each transition is represented as a as an arc. It represents the machine's behavior not its implementation!

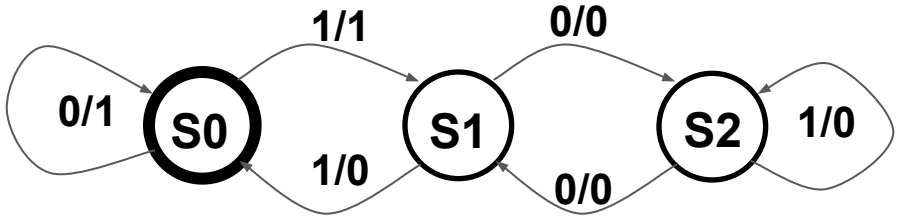


= no buttons pressed

Example State Diagrams



MOORE Machine:
Outputs on States



MEALY Machine:
Outputs on Transitions

Arcs leaving a state must be:

(1) mutually exclusive

can only have one choice for any given input value

(2) collectively exhaustive

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

Next Time

Counting state machines