
EE 40 Digital Circuits

Reading Material:
None

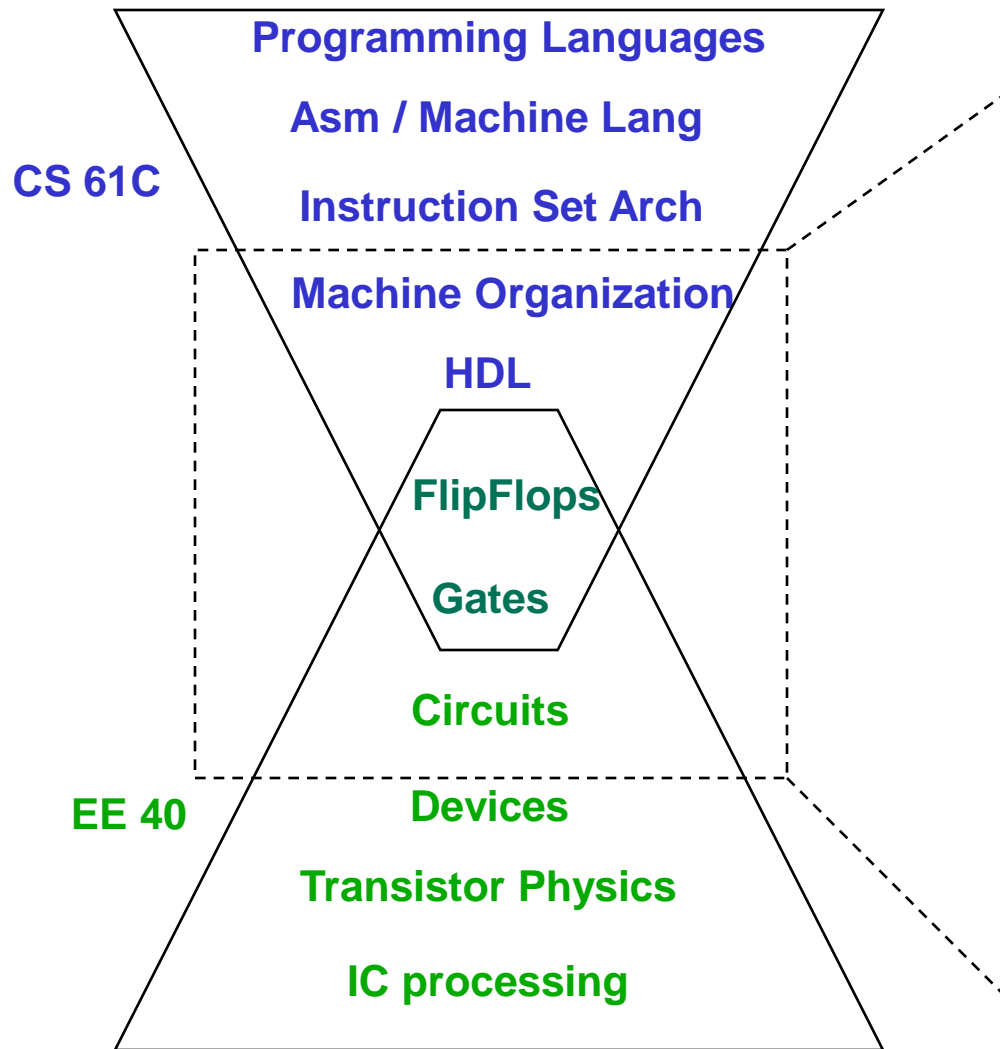
A quick history lesson

- 1850: George Boole invents Boolean algebra
- 1938: Claude Shannon links Boolean algebra to switches
- 1945: John von Neumann develops first stored program computer
 - Its switching elements are vacuum tubes (a big advance from relays)
- 1946: ENIAC--world's first all electronic computer
 - 18,000 vacuum tubes
 - Several hundred multiplications per minute
- 1947: Shockley, Brittain, and Bardeen invent the transistor

What is digital hardware?

- Collection of devices that sense and/or control wires carrying a digital value (i.e., a physical quantity interpreted as a “0” or “1”)
 - logic where voltage $< 0.8V$ is “0” and $> 2.0V$ is “1”
 - e.g., orientation of magnetization signifies “0” or “1”
- Primitive digital hardware devices
 - Logic computation devices (sense and drive)
 - two wires both “1” - make another be “1” (AND)
 - at least one of two wires “1” - make another be “1” (OR)
 - a wire “1” - then make another be “0” (NOT)
 - Memory devices (store)
 - store a value
 - recall a value previously stored

40 – 61C – 150



Deep Digital Design Experience

- Fundamentals of Boolean Logic
- Synchronous Circuits
- Finite State Machines
- Timing & Clocking
- Device Technology & Implications
- Controller Design
- Arithmetic Units
- Encoding, Framing
- Testing, Debugging
- Hardware Architecture
- Hardware Design Language (HDL)
- Design Flow (CAD)

Different layers in the stack

High-level Organization : Hardware Architectures

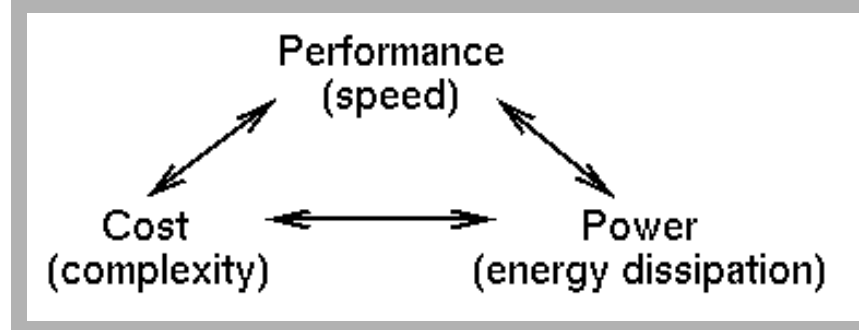
System Building Blocks : Arithmetic units, controllers

Circuit Elements : Memories, logic blocks

Transistor-level circuit implementations

Circuit primitives : Transistors, wires

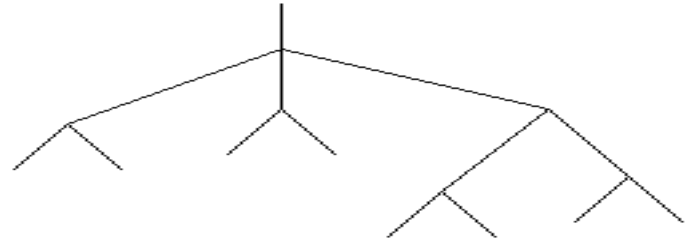
Basic Design Tradeoffs



- You can improve on one at the expense of worsening one or both of the others.
- These tradeoffs exist at every level in the system design - every sub-piece and component.
- Design Specification -
 - Functional Description.
 - Performance, cost, power constraints.
- As a designer you must make the tradeoffs necessary to achieve the function within the constraints.

Hierarchy in Designs

- Helps control complexity -
 - by hiding details and reducing the total number of things to handle at any time.
- Modularizes the design -
 - divide and conquer
 - simplifies implementation and debugging
- Top-Down Design
 - Starts at the top (root) and works down by successive refinement.
- Bottom-up Design
 - Starts at the leaves & puts pieces together to build up the design.
- Which is better?
 - In practice both are needed & used.
 - Need top-down divide and conquer to handle the complexity.
 - Need bottom-up because in a well designed system, the structure is influence by what primitives are available.



Digital Design

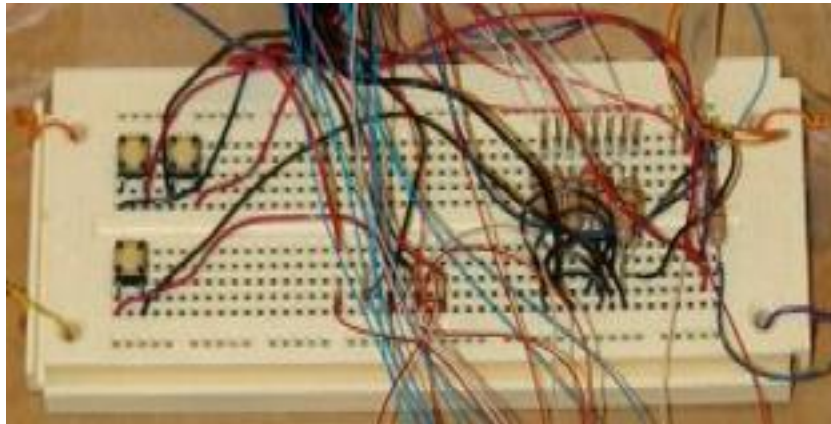
Given a functional description and performance, cost, & power constraints, come up with an implementation using a set of primitives.

- How do we learn how to do this?
 1. Learn about the primitives and how to use them.
 2. Learn about design representations.
 3. Learn formal methods to optimally manipulate the representations.
 4. Look at design examples.
 5. Use trial and error - CAD tools and prototyping. Practice!
- Digital design is in some ways more an art than a science. The creative spirit is critical in combining primitive elements & other components in new ways to achieve a desired function.
- However, unlike art, we have objective measures of a design:

Performance Cost Power

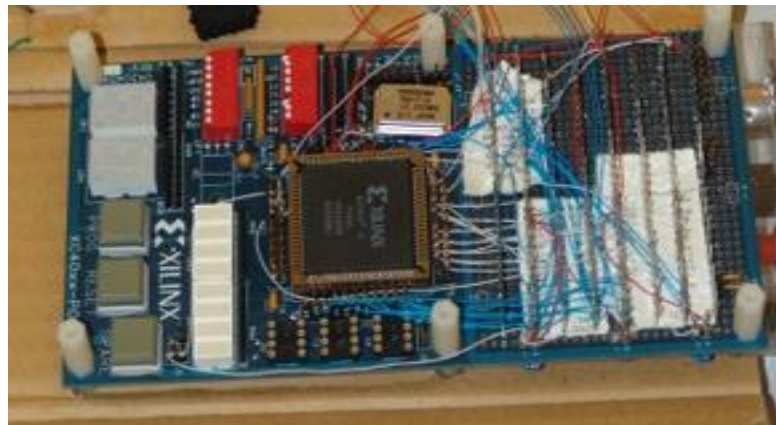
FPGA Evolution

- Final project circa 1980:
 - Example project: pong game with buttons for paddle and LEDs for output.
 - Few 10's of logic gates
 - Gates hand-wired together on “bread-board” (protoboard).
 - No computer-aided design tools
 - Debugged with oscilloscope and logic analyzer

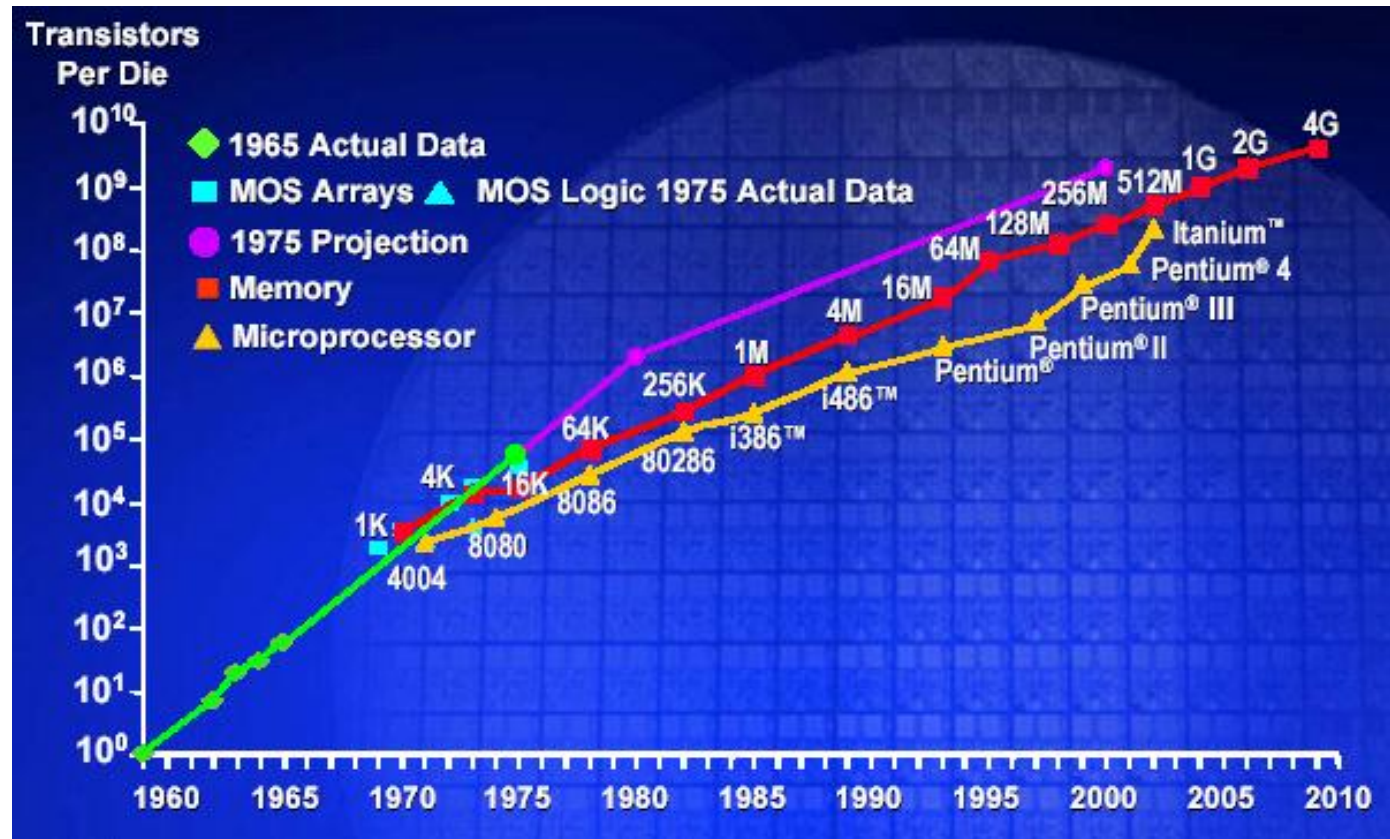


FPGA Evolution

- Final project circa 1995:
 - Example project: MIDI music synthesizer
 - Few 1000's of logic gates
 - Gates wired together internally on field programmable gate array (FPGA) development board with some external components.
 - Circuit designed “by-hand”, computer-aided design tools to help map the design to the hardware.
 - Debugged with circuit simulation, oscilloscope and logic analyzer



Moore's Law – 2x stuff per 1-2 yr

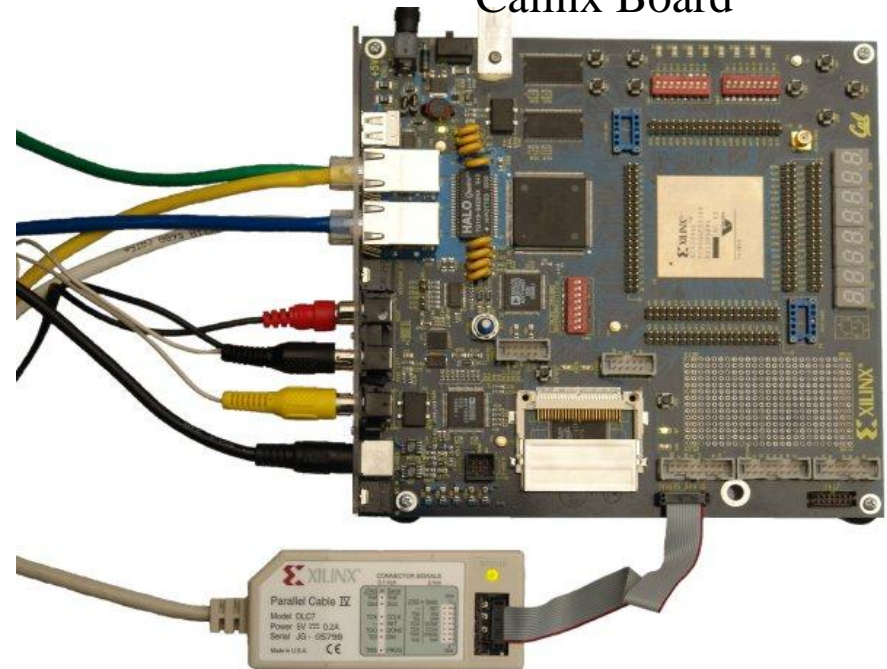


FPGA Evolution

- Final project circa 2000-2008:
 - Example project: eTV - streaming video broadcast over Ethernet, student project decodes and displays video
 - Few 10,000's of logic gates
 - Gates wired together internally on FPGA development board and communicate with standard external components.
 - Circuit designed with logic-synthesis tools, computer-aided design tools to help map the design to the hardware.
 - Debugged with circuit simulation, logic analyzer, and in-system debugging tools.

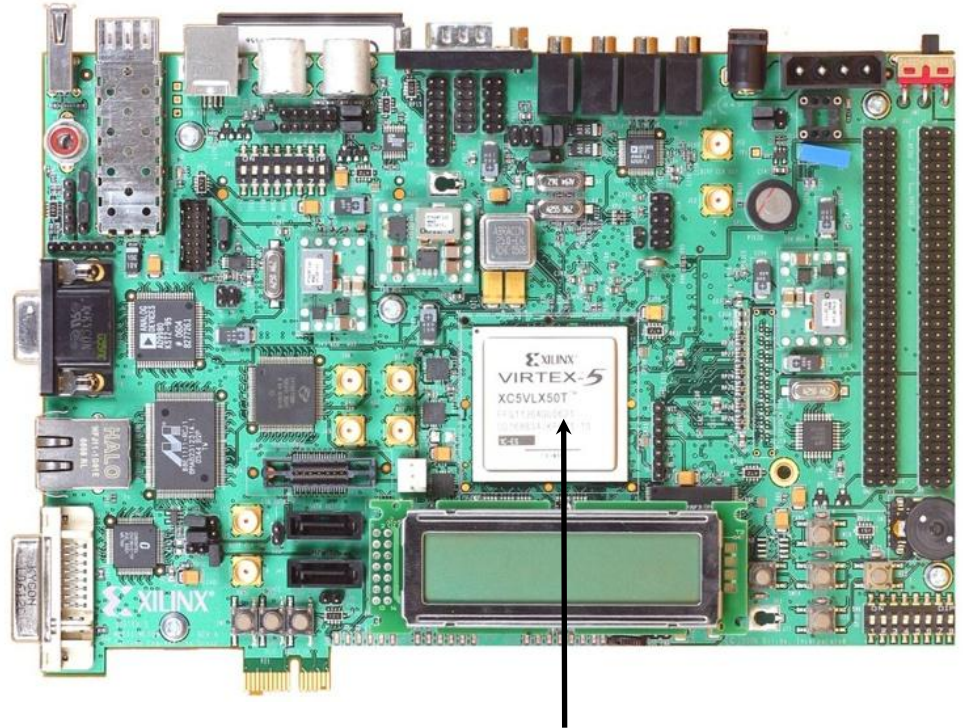


Calinx Board



FPGA Evolution

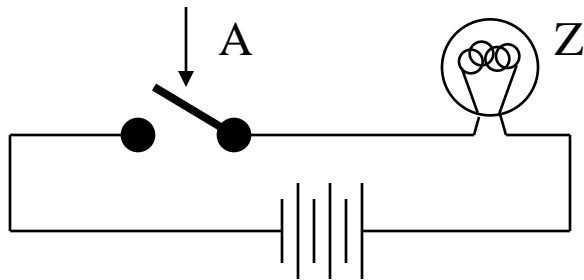
- Began Spring 2009:
 - Xilinx XUPV5 development board (a.k.a ML505)
 - Can enable very aggressive final projects.
 - Project debugging with simulation tools and with in-system hardware debugging tools.



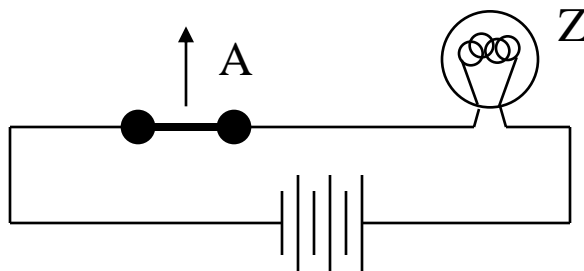
- State-of-the-art LX110T FPGA: ~1M logic gates.
- Interfaces: Audio in/out, digital video, ethernet, on-board DRAM, PCIe, USB, ...

Physical Implementation - Switch

- Implementing a simple circuit (arrow shows action if wire changes to “1”):



close switch (if A is “1” or asserted)
and turn on light bulb (Z)

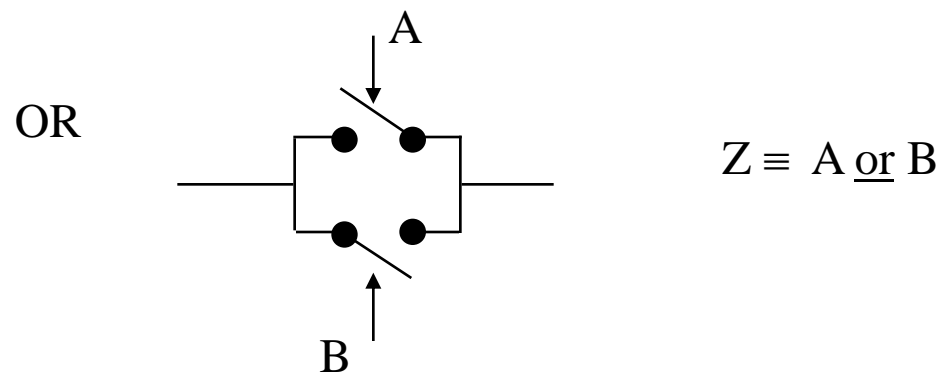
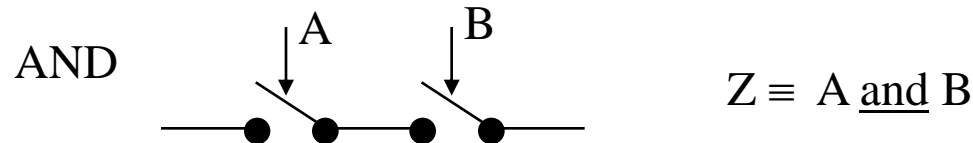


open switch (if A is “0” or unasserted)
and turn off light bulb (Z)

$$Z \equiv A$$

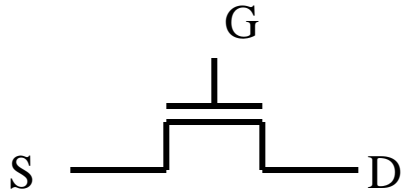
Switches (cont'd)

- Compose switches into more complex ones (Boolean functions):



MOS transistors

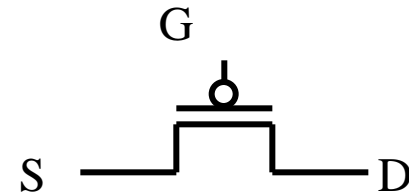
- MOS transistors act as switches as follows:
 - if voltage on gate terminal is (some amount) higher/lower than source terminal then a conducting path is established between drain and source terminals



n-channel

open when voltage at G is low
closes when:

$\text{voltage}(G) > \text{voltage}(S) + \epsilon$



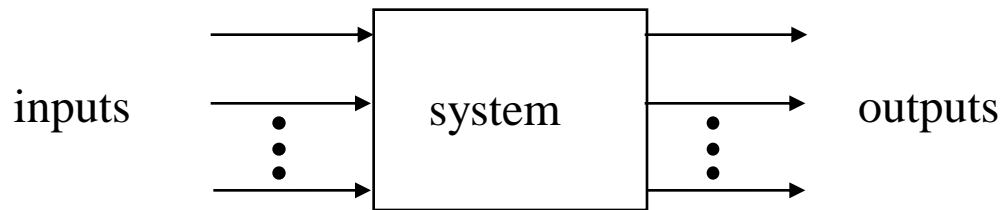
p-channel

closed when voltage at G is low
opens when:

$\text{voltage}(G) < \text{voltage}(S) - \epsilon$

Combinational Logic

- A simple model of a digital system is a unit with inputs and outputs:



- Combinational means "memory-less"
 - a digital circuit is combinational if its output values only depend on its input values

Combinational logic symbols

- Common combinational logic systems have standard symbols called logic gates
 - Buffer, NOT

A  Z



– AND, NAND

A  B Z



– OR, NOR

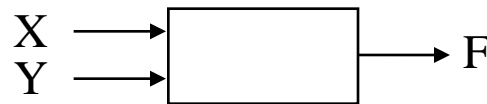
A  B Z



easy to implement
with CMOS transistors
(the switches we have
available and use most)

2-Variable Logic Functions

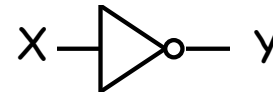
- There are 16 possible functions of 2 input variables:



X	Y	16 possible functions (F0–F15)															
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		<div><div>0</div><div>X and Y</div><div>X</div><div>Y</div><div>X xor Y</div><div>X or Y</div><div>X nor Y</div><div>not (X or Y)</div><div>X = Y</div><div>not Y</div><div>not X</div><div>X nand Y</div><div>not (X and Y)</div><div>1</div></div>															

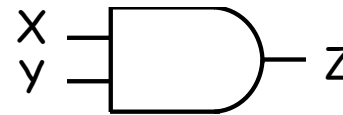
Logic Gates and Truth Tables

- NOT X' \bar{X} $\sim X$



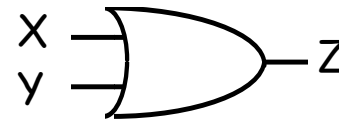
X	Y
0	1
1	0

- AND $X \bullet Y$ XY



X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

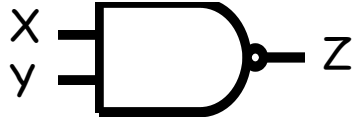
- OR $X + Y$



X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1


Logic Gates and Truth Tables

■ **NAND**



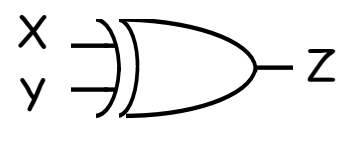
X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

■ **NOR**



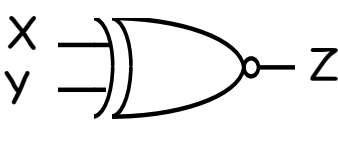
X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	0

■ **XOR**
 $X \oplus Y$



X	Y	Z	$X \text{ xor } Y = X Y' + X' Y$ X or Y but not both ("inequality", "difference")
0	0	0	
0	1	1	
1	0	1	
1	1	0	

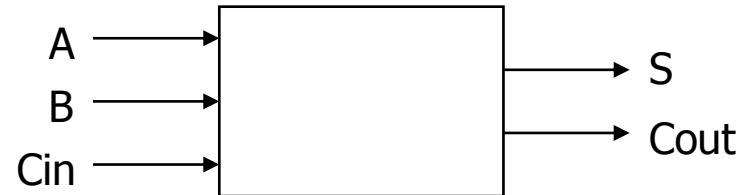
■ **XNOR**
 $X = Y$



X	Y	Z	$X \text{ xnor } Y = X Y + X' Y'$ X and Y are the same ("equality", "coincidence")
0	0	1	
0	1	0	
1	0	0	
1	1	1	

Example: 1 bit adder

- 1-bit binary adder
 - inputs: A, B, Carry-in
 - outputs: Sum, Carry-out



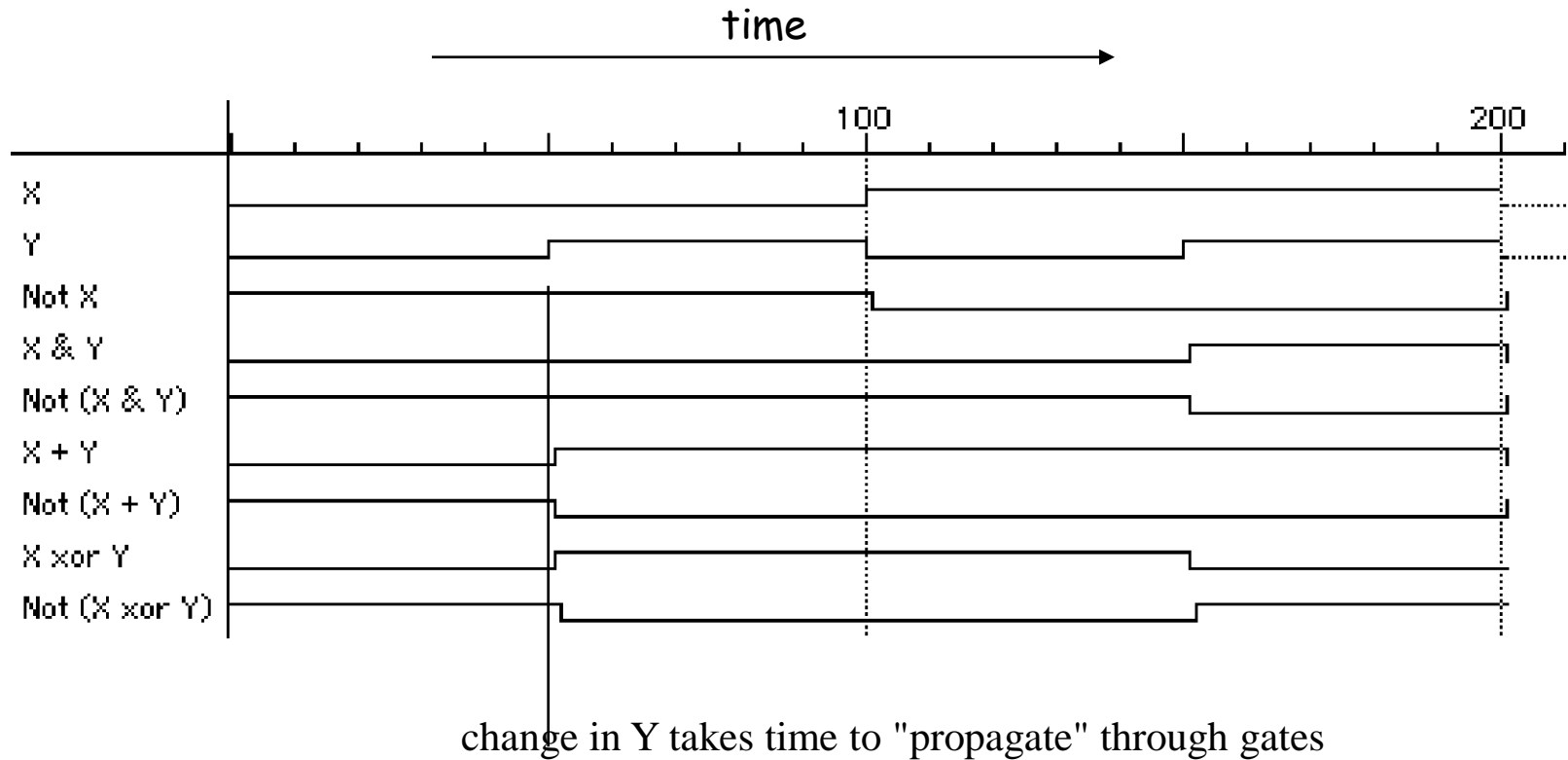
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$S = A' B' Cin + A' B Cin' + A B' Cin' + A B Cin$$

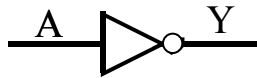
$$Cout = A' B Cin + A B' Cin + A B Cin' + A B Cin$$

Waveform View of Logic Functions

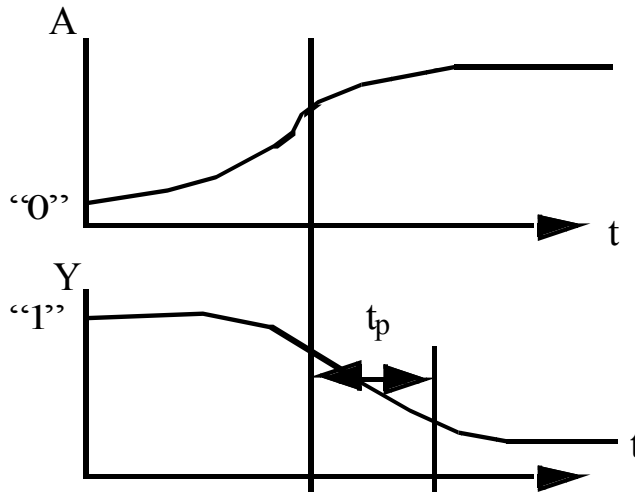
- Just a sideways truth table
 - but note how edges don't line up exactly
 - it takes time for a gate to switch its output!



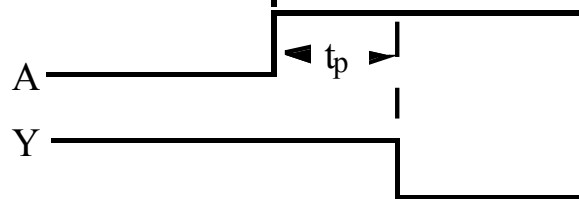
Timing of logic gates



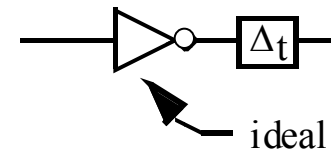
(as seen with oscilloscope)



Delay model



or



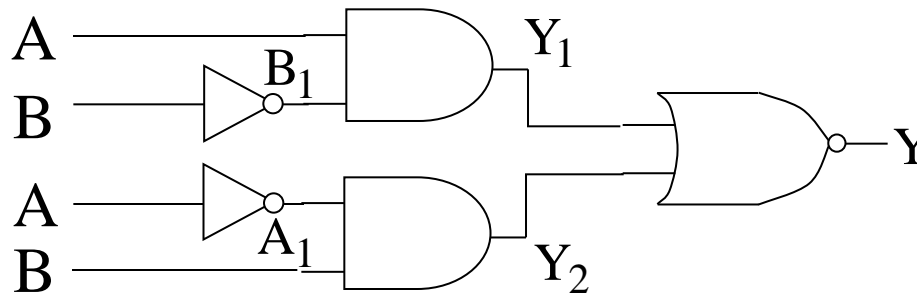
Timing

- Why big deal?
- Dynamic output \neq static output (“glitches”)

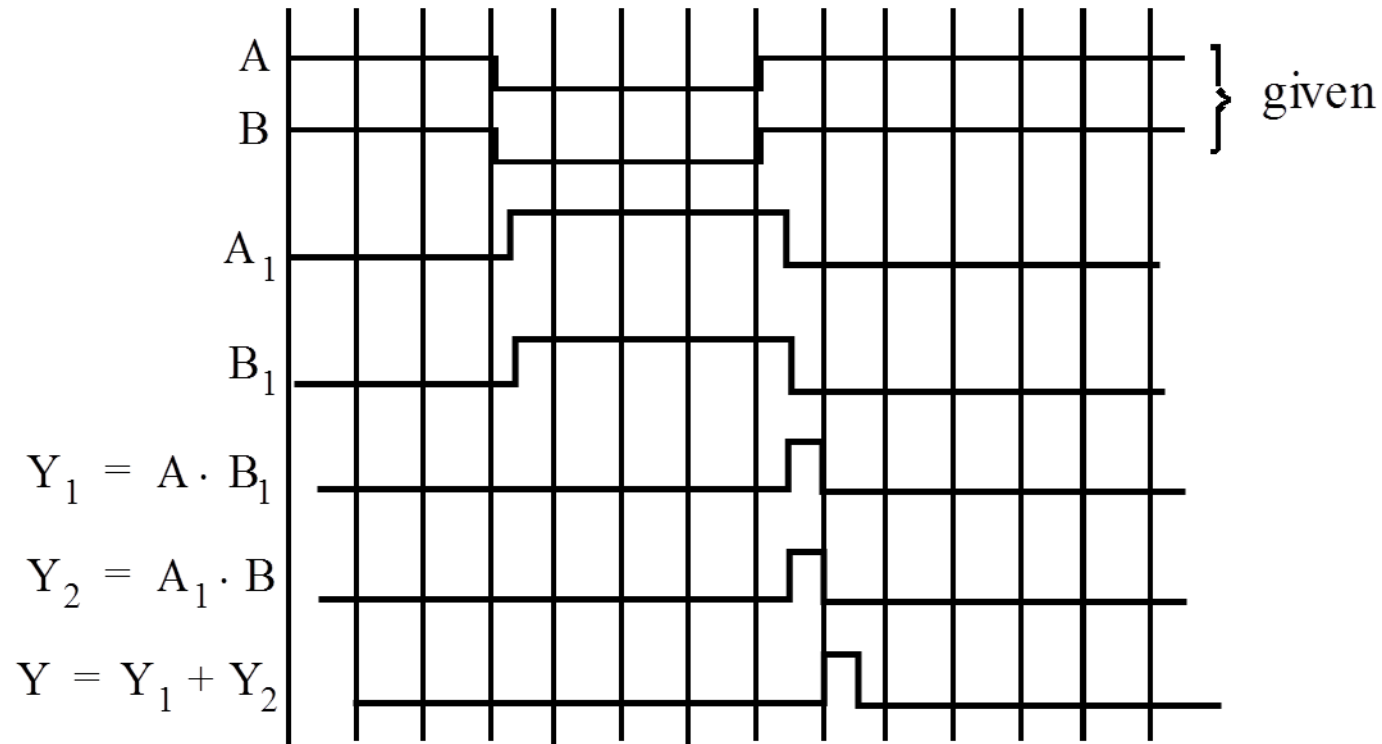
Example: XOR function implemented from AND, OR, INV

idealization: unit delay in each symbol

$$Y = A \cdot \bar{B} + \bar{A} \cdot B$$



Timing Diagram



Static behavior fine $1 \oplus 1 = 0, 0 \oplus 0 = 0$ ✓

Dynamic behavior **glitch** → fundamental feature of switching

Combination vs. Sequential Logic

Combinational

$$\underline{y} = \underline{f}(\text{inputs})$$

n inputs, m outputs

Examples:

Adder: $y[0:3] = a[0:3] + b[0:3]$

memoryless systems

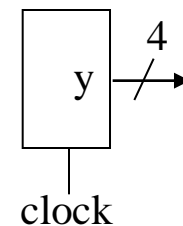
Sequential

$$\underline{y} = \underline{f}(\text{inputs, time})$$

(most useful: clocked logic)

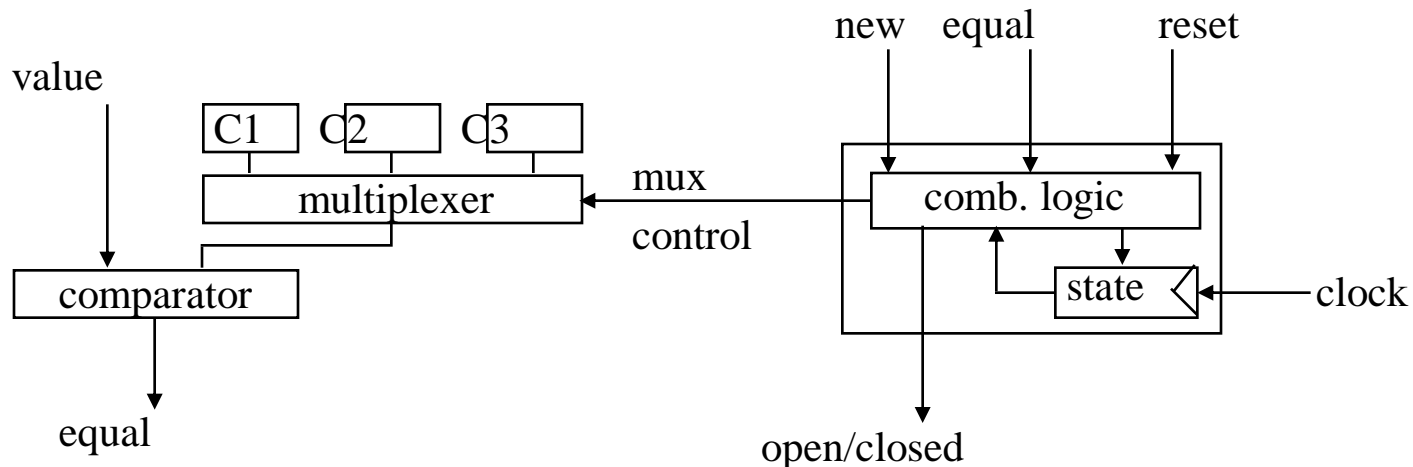
Example: (down) counter

Time (e.g. sec)	Y[0:3]
0	10 ₁₀
1	9
2	8
...	...
10 ₁₀	0



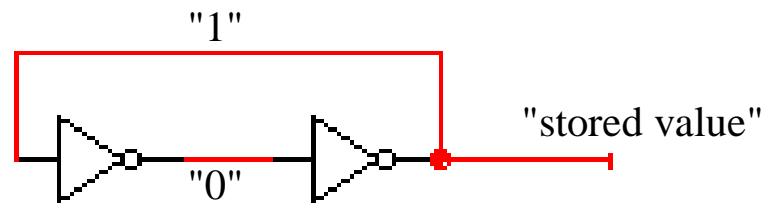
Sequential Circuits

- Circuits with Feedback
 - Outputs = $f(\text{inputs, past inputs, past outputs})$
 - Basis for building "memory" into logic circuits
 - Door combination lock is an example of a sequential circuit
 - State is memory
 - State is an "output" and an "input" to combinational logic
 - Combination storage elements are also memory

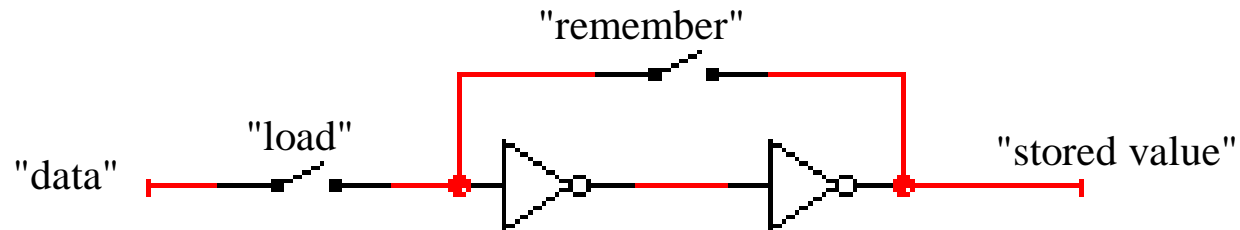


Simple circuits with feedback

- Two inverters form a static memory cell
 - Will hold value as long as it has power applied

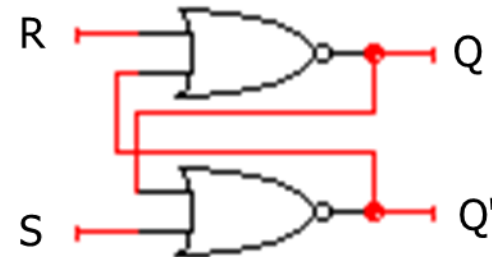
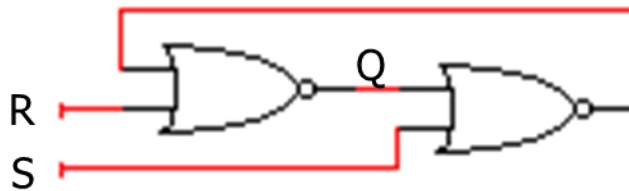


- How to get a new value into the memory cell?
 - Selectively break feedback path
 - Load new value into cell

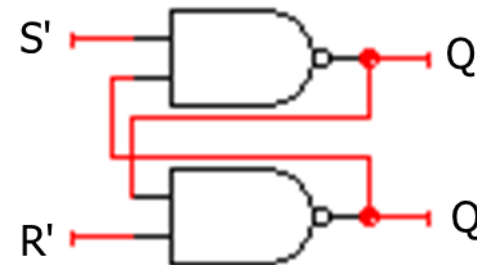
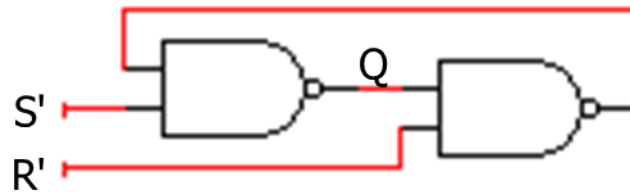


Cross-coupled gates

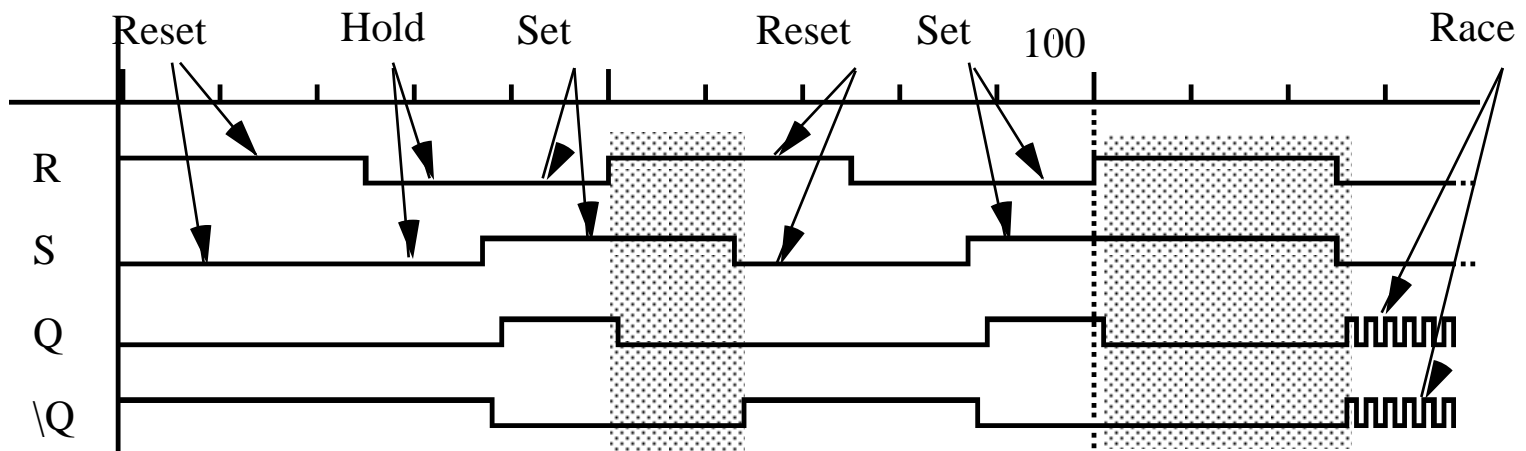
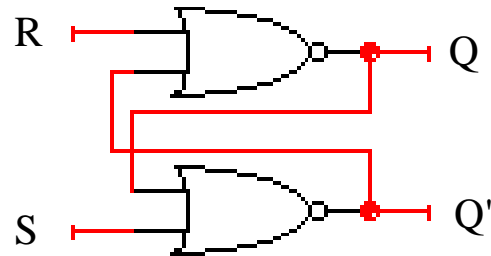
- Cross-coupled NOR gates
 - Similar to inverter pair, with capability to force output to 0 (reset=1) or 1 (set=1)



- Cross-coupled NAND gates
 - Similar to inverter pair, with capability to force output to 0 (reset=0) or 1 (set=0)



Timing behavior



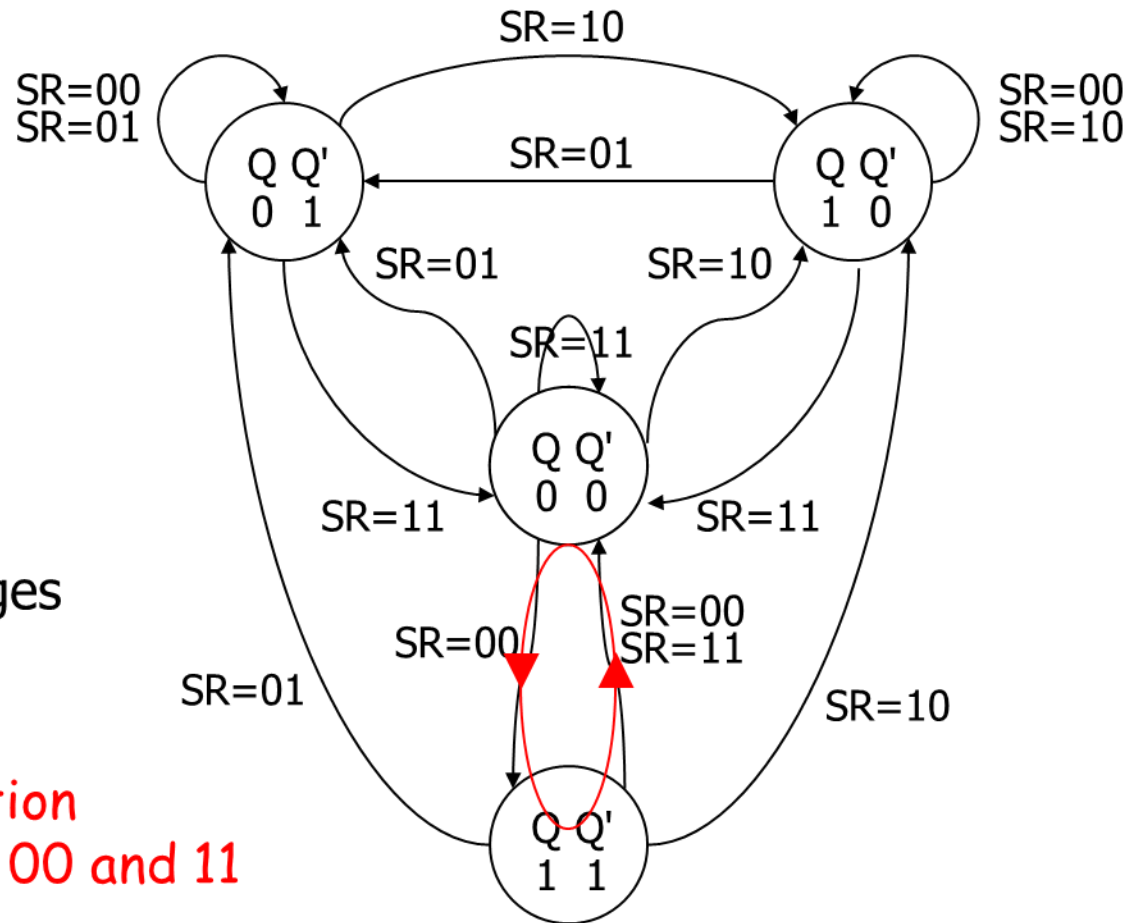
R-S Latch Behavior

S	R	Q
0	0	hold
0	1	0
1	0	1
1	1	unstable

■ State Diagram

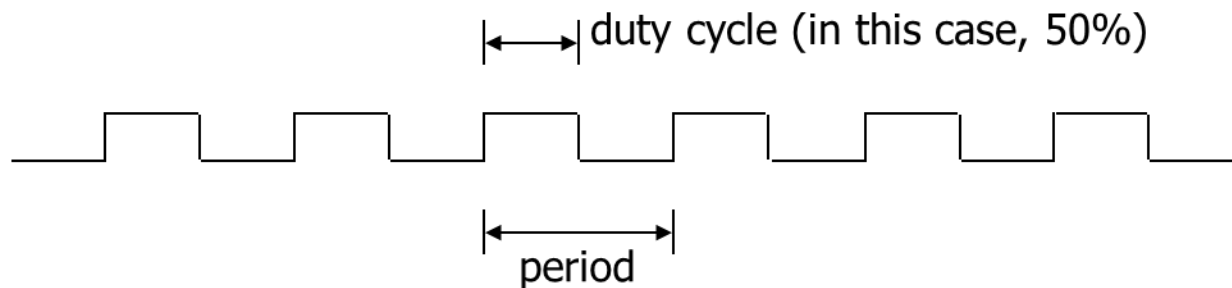
- States: possible values
- Transitions: changes based on inputs

possible oscillation
between states 00 and 11



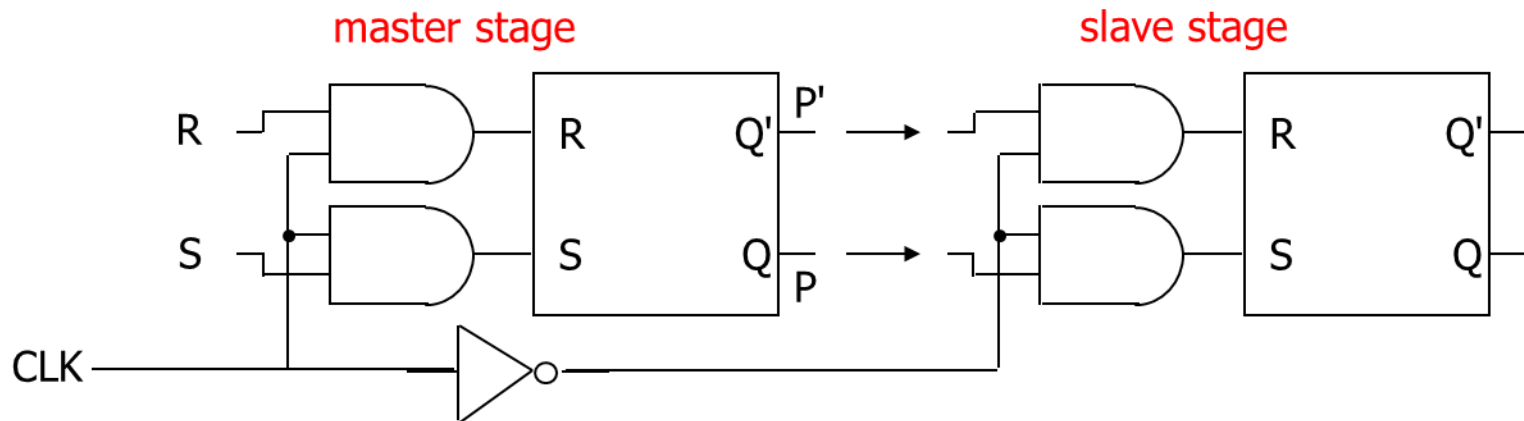
Clocks

- Used to keep time
 - Wait long enough for inputs (R' and S') to settle
 - Then allow to have effect on value stored
- Clocks are regular periodic signals
 - Period (time between ticks)
 - Duty-cycle (time clock is high between ticks - expressed as % of period)



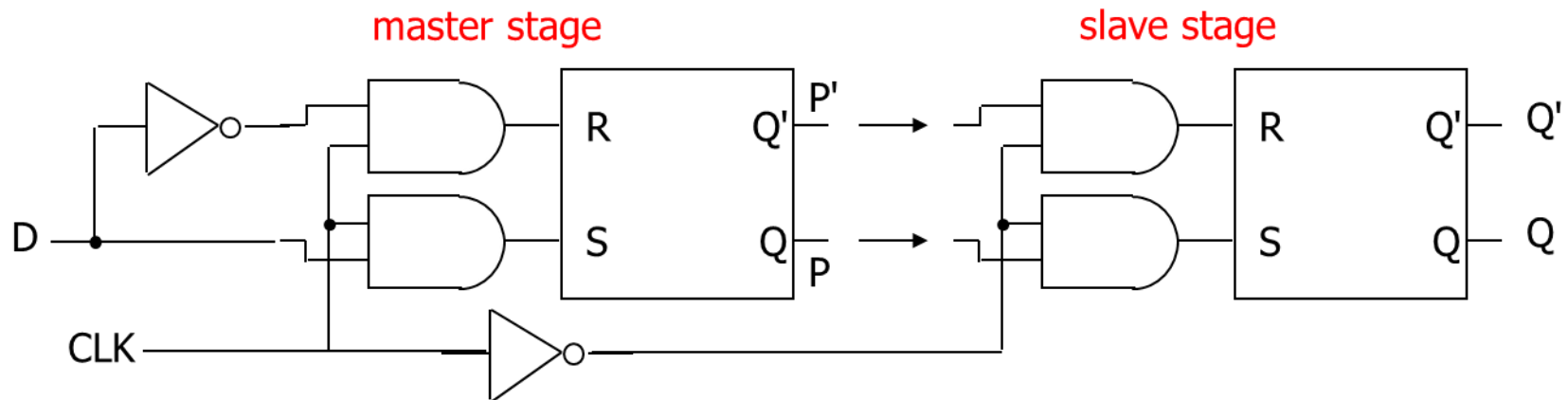
Master-Slave Structure

- Break flow by alternating clocks (like an air-lock)
 - Use positive clock to latch inputs into one R-S latch
 - Use negative clock to change outputs with another R-S latch
- View pair as one basic unit
 - master-slave flip-flop
 - twice as much logic
 - output changes a few gate delays after the falling edge of clock but does not affect any cascaded flip-flops



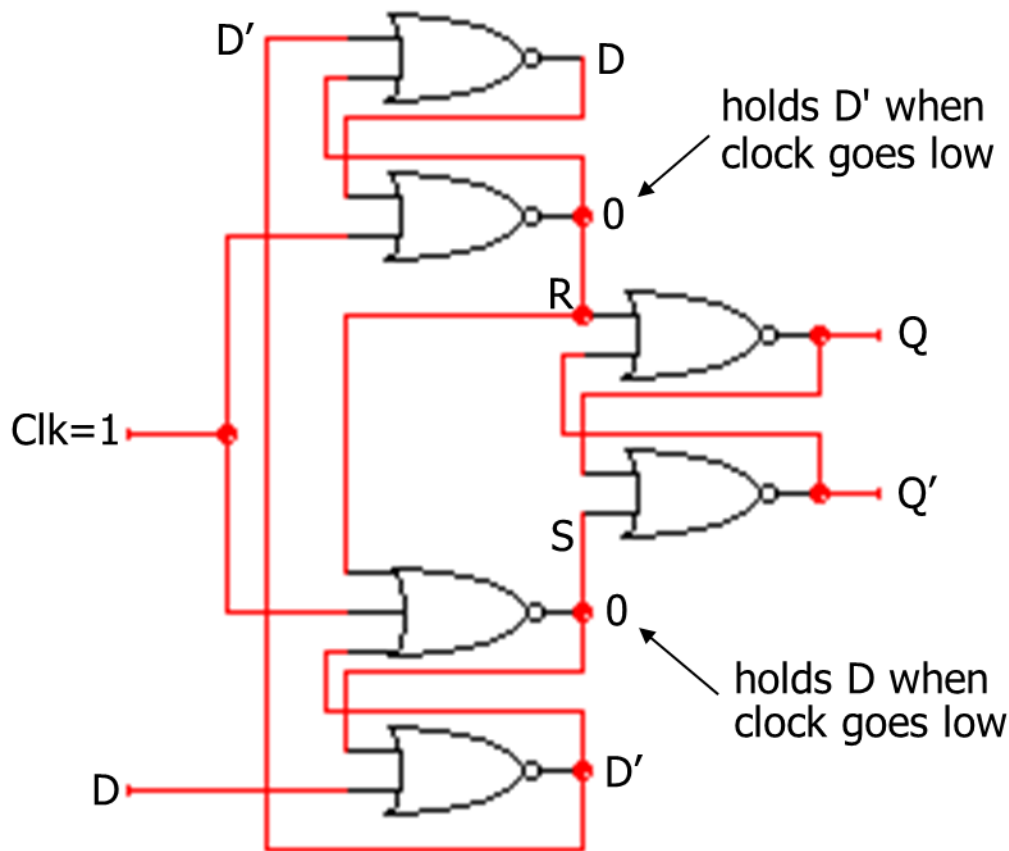
D Flip Flop

- Make S and R complements of each other
 - Eliminates 1s catching problem
 - Can't just hold previous value (must have new value ready every clock period)
 - Value of D just before clock goes low is what is stored in flip-flop
 - Can make R-S flip-flop by adding logic to make $D = S + R' Q$



Edge-triggered Flip Flop

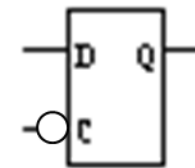
- More efficient solution: only 6 gates
 - sensitive to inputs only near edge of clock signal (not while high)



negative edge-triggered D
flip-flop (D-FF)

4-5 gate delays

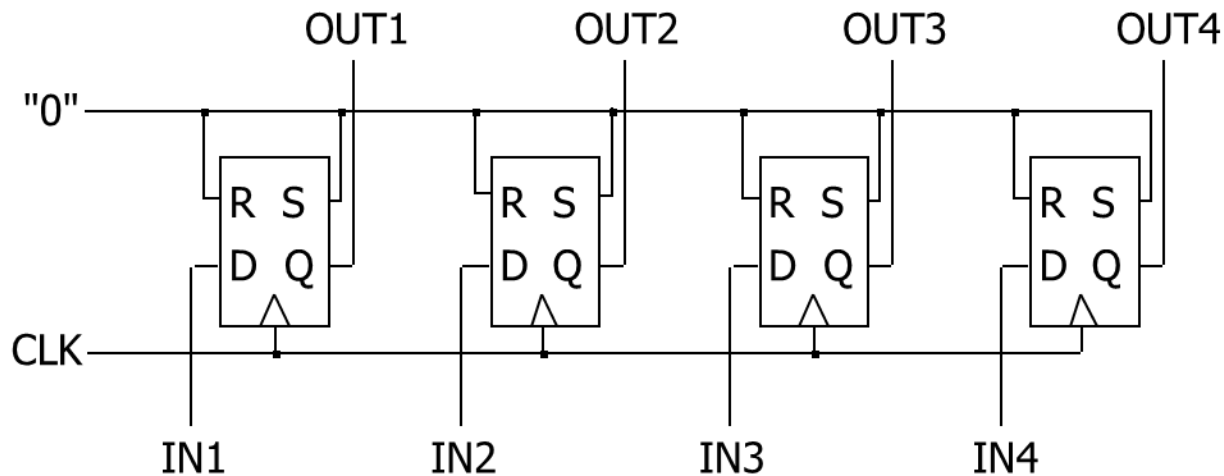
must respect setup and hold time
constraints to successfully
capture input



characteristic equation
 $Q(t+1) = D$

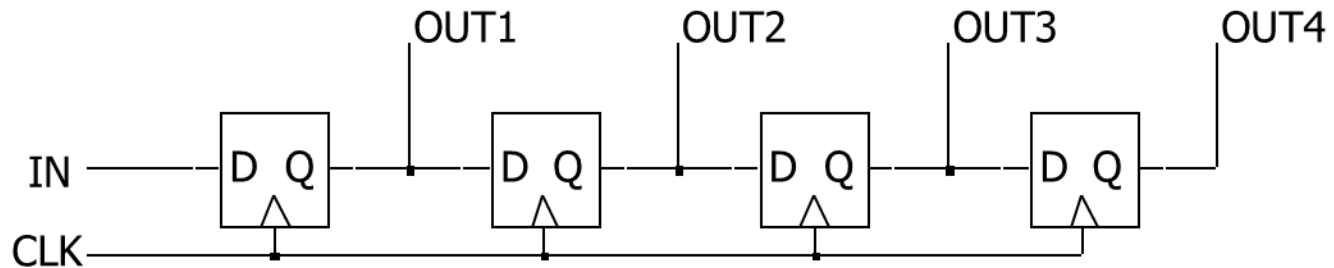
Registers

- Collections of flip-flops with similar controls and logic
 - Stored values somehow related (e.g., form binary value)
 - Share clock, reset, and set lines
 - Similar logic at each stage
- Examples
 - Shift registers
 - Counters



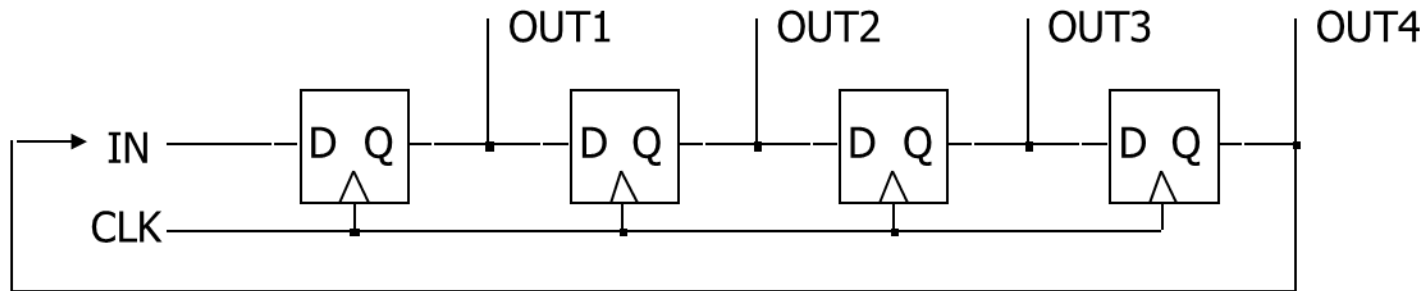
Shift Register

- Holds samples of input
 - Store last 4 input values in sequence
 - 4-bit shift register:

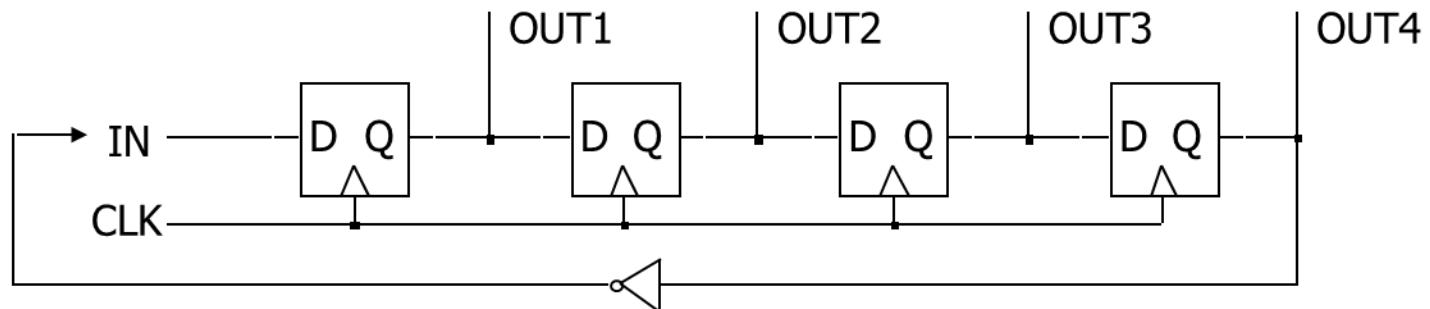


Counters

- Sequences through a fixed set of patterns
 - In this case, 1000, 0100, 0010, 0001
 - If one of the patterns is its initial state (by loading or set/reset)

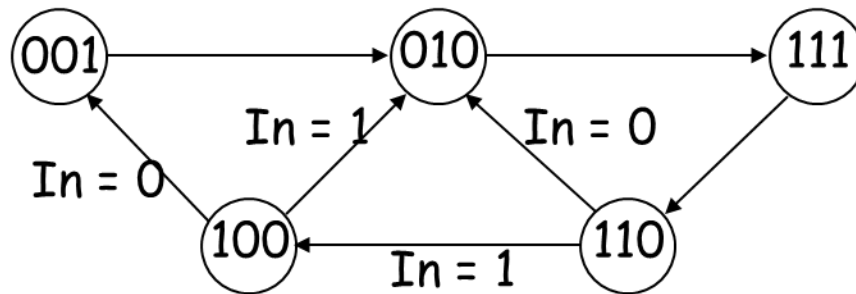


- Mobius (or Johnson) counter (only 1 bit changes at a time)
 - In this case, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000



Finite State Machines

- States: determined by possible values in sequential storage elements
- Transitions: change of state
- Clock: controls when state can change by controlling storage elements

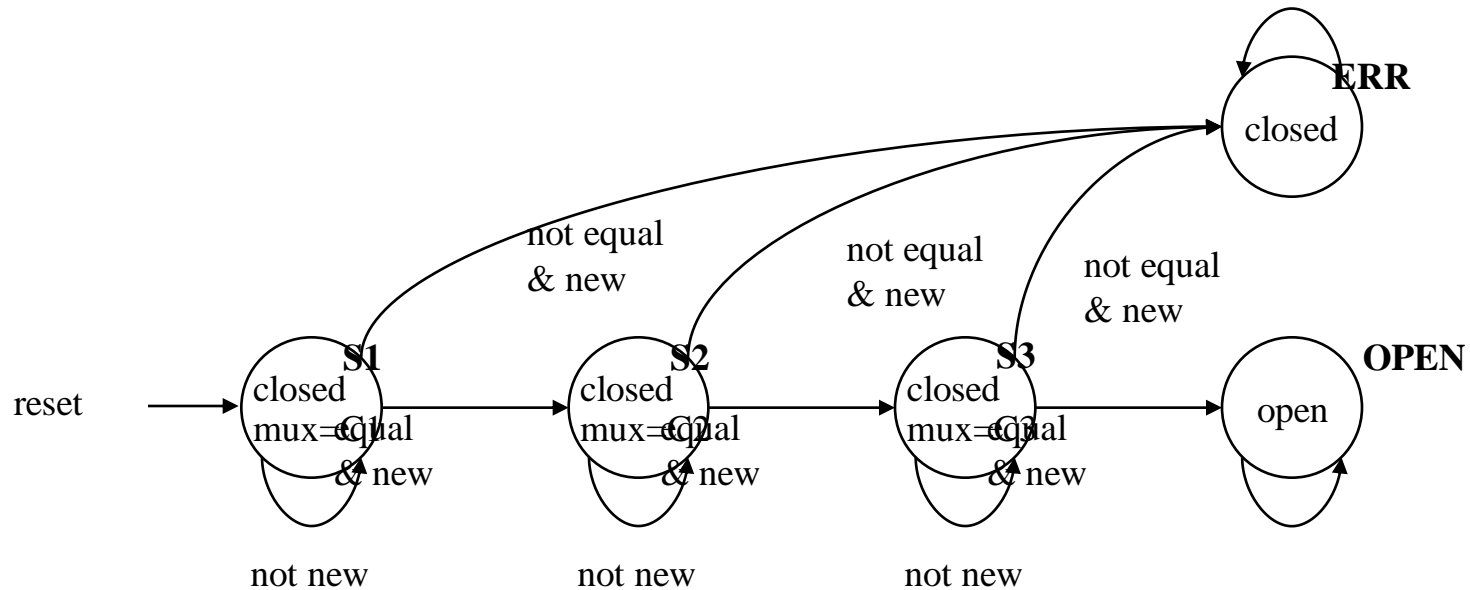


- Sequential Logic
 - Sequences through a series of states
 - Based on sequence of values on input signals
 - Clock period defines elements of sequence

FSM example

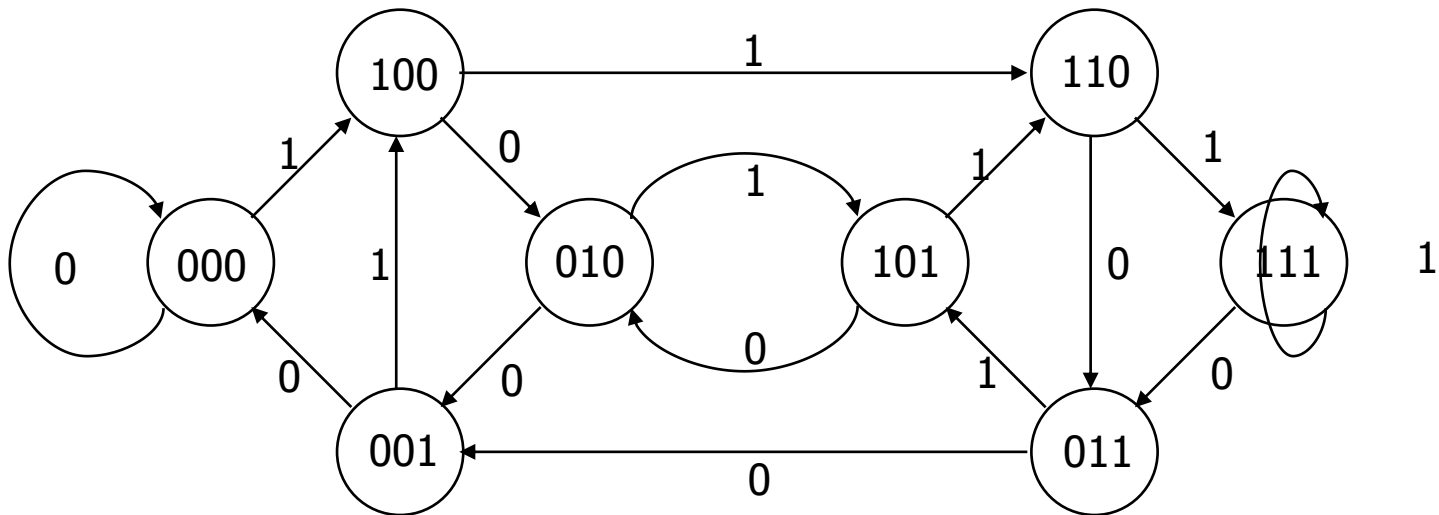
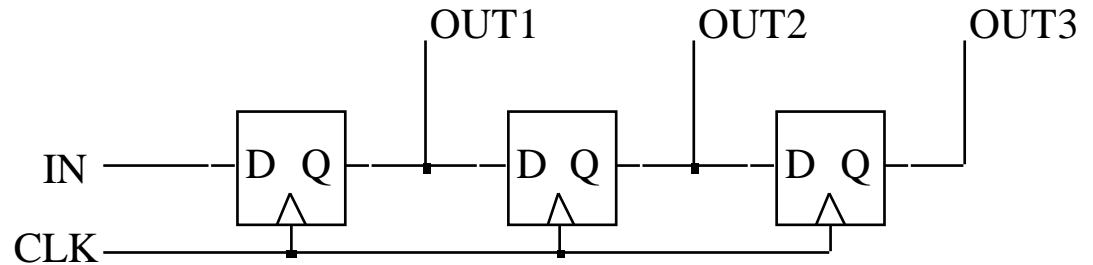
- Door combination lock:
 - punch in 3 values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset
 - inputs: sequence of input values, reset
 - outputs: door open/close
 - memory: must remember combination
or always have it available as an input

FSM Representation



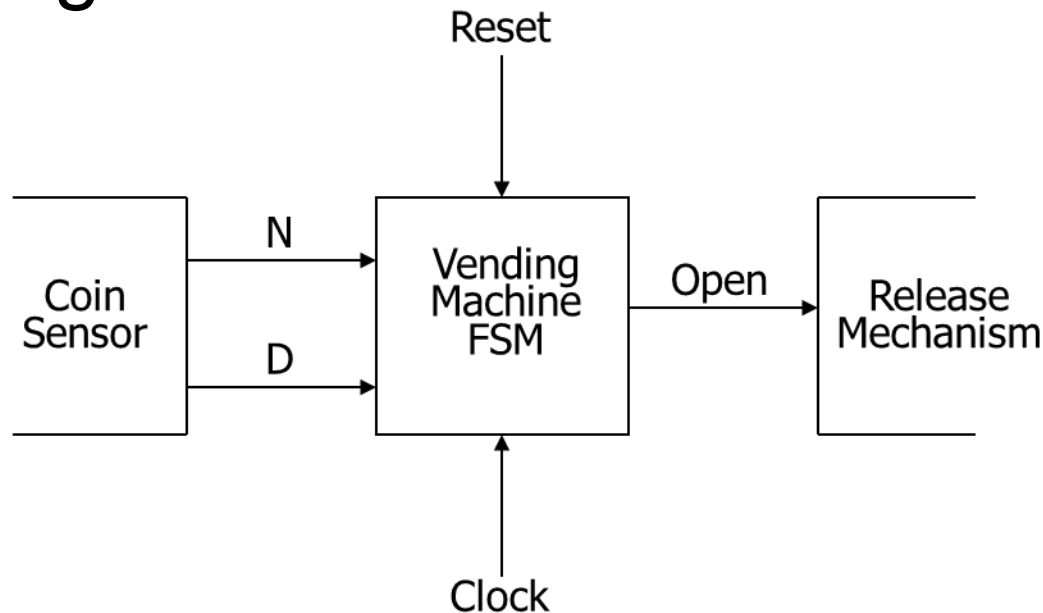
Shift Registers as FSMs

- Shift Register
 - Input value shown on transition arcs
 - Output values shown within state node



FSM Example: Vending Machine

- Release item after 15 cents are deposited
- Single coin slot for dimes, nickels
- No change



Vending Machine

■ Suitable Abstract Representation

■ Tabulate typical input sequences:

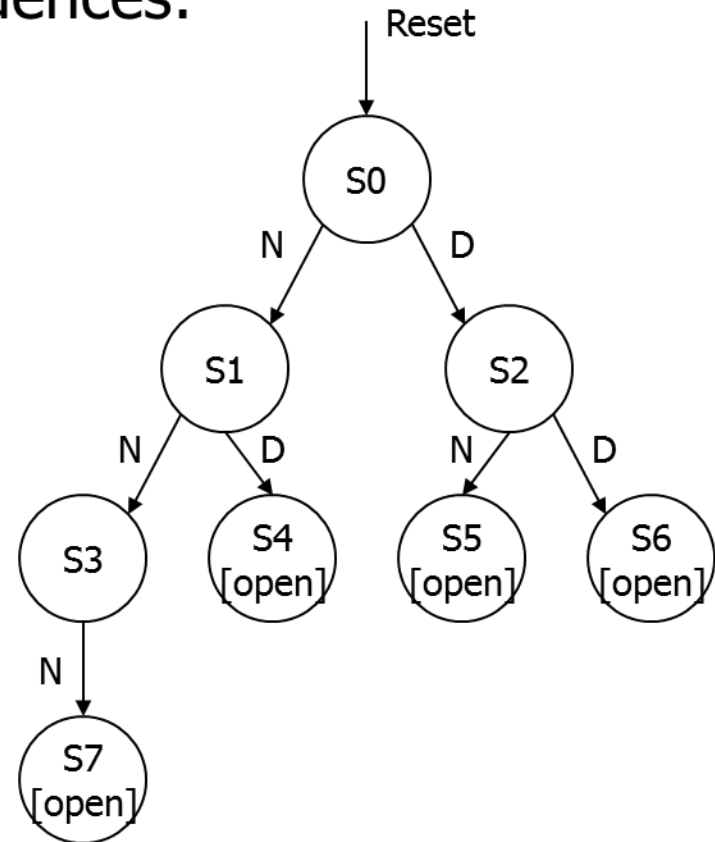
- 3 nickels
- nickel, dime
- dime, nickel
- two dimes

■ Draw state diagram:

- Inputs: N, D, reset
- Output: open chute

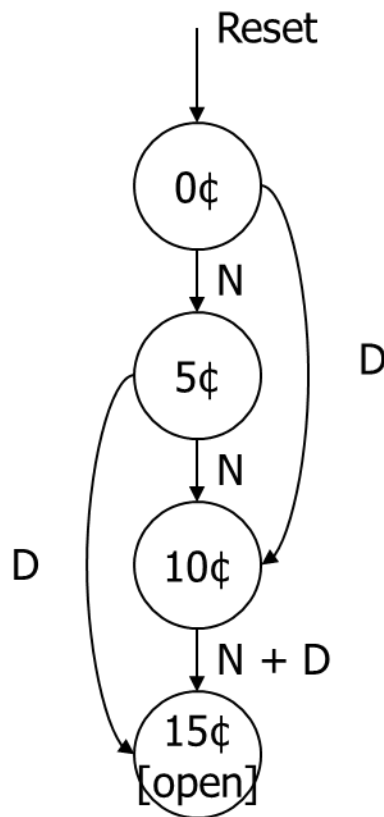
■ Assumptions:

- Assume N and D asserted for one cycle
- Each state has a self loop for $N = D = 0$ (no coin)



Vending Machine

- Minimize number of states - reuse states whenever possible



present state	inputs		next state	output open
	D	N		
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	—	—
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	—	—
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	—	—
15¢	—	—	15¢	1

symbolic state table

Vending Machine

- Uniquely Encode States

present state		inputs		next state		output
Q1	Q0	D	N	D1	D0	open
0	0	0	0	0	0	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	—	—	—
0	1	0	0	0	1	0
		0	1	1	0	0
		1	0	1	1	0
		1	1	—	—	—
1	0	0	0	1	0	0
		0	1	1	1	0
		1	0	1	1	0
		1	1	—	—	—
1	1	—	—	1	1	1

Vending Machine

■ Mapping to Logic

$$D1 = Q1 + D + Q0 N$$

$$D0 = Q0' N + Q0 N' + Q1 N + Q1 D$$

$$\text{OPEN} = Q1 Q0$$

